Best Practice Guide
Modern Accelerators

João Bispo, University of Porto, Portugal
Jorge G. Barbosa, University of Porto, Portugal
Pedro Filipe Silva, University of Porto, Portugal
Cristian Morales, BSC, Spain
Mirko Myllykoski, HPC2N, Sweden
Pedro Ojeda-May, HPC2N, Sweden
Milosz Bialczak, WCSS, Poland
Mariusz Uchronski, WCSS, Poland
Adam Wlodarczyk, WCSS, Poland
Peter Wauligmann, HLRS, Germany
Ezhilmathi Krishnasamy, University of Luxembourg, Luxembourg
Sebastien Varrette, University of Luxembourg, Luxembourg
Sebastian Lührs, JSC, Germany
Hayk Shoukourian (Editor), LRZ, Germany

04-06-2021

Table of Contents

1. Introduction 4
2. Architectures 5
    2.1. GPUs 5
    2.2. FPGAs 7
    2.3. NEC SX-Aurora TSUBASA 9
3. Suitability of accelerators for different HPC codes 11
    3.1. Roofline Model 11
    3.2. Accelerator Suitability 14
        3.2.1. GPUs 14
        3.2.2. FPGAs 16
        3.2.3. Vector Processors 17
    3.3. Summary 18
4. Programming Models and Environment 19
    4.1. Introduction 19
    4.2. CUDA 20
        4.2.1. Vendor libraries 21
        4.2.2. Host, devices and NVIDIA CUDA Compiler 21
        4.2.3. Kernels 22
        4.2.4. Threads and thread blocks 23
        4.2.5. Memory spaces and data transfers 25
        4.2.6. Examples 25
    4.3. OpenCL 29
        4.3.1. Compute units and processing elements 30
        4.3.2. Host and kernels 30
        4.3.3. Work-items and work groups 30
        4.3.4. Memory spaces and data transfers 32
        4.3.5. Host-side API 33
        4.3.6. Terminology comparison 34
        4.3.7. Examples 34
    4.4. SYCL 39
        4.4.1. Overview 39
        4.4.2. Expressing parallelism 39
        4.4.3. Data access and storage 41
        4.4.4. Examples 42
    4.5. OpenACC 44
        4.5.1. Overview 44
        4.5.2. Configuration 44
        4.5.3. Programming Model 45
        4.5.4. Examples 47
    4.6. HIP 49
        4.6.1. HIP API 49
        4.6.2. HIPify Tools 51
        4.6.3. Examples 51
    4.7. HLS 53
        4.7.1. Xilinx Vitis HLS 54
    4.8. NEC Programming Environment 59
        4.8.1. NEC Compilers 59
        4.8.2. NEC MPI 60
        4.8.3. Execution Mode 60
    4.9. Summary 61
5. Performance Analysis and Tuning 63
    5.1. Profiling Tools 63
        5.1.1. nvprof 63
        5.1.2. Visual Profiler 67
        5.1.3. Nsight Compute and Nsight Systems 68

5.1.4. PGI compiler (pgprof) 69
5.2. Tuning for NVIDIA GPUs 72
5.2.1. CUDA Occupancy Calculator 72
5.2.2. Shared Memory Programming 72
5.2.3. Register Usage 74
5.2.4. Cache Configuration 74
5.2.5. Memory Access Through NVLink Interconnect 75
5.2.6. NVIDIA CUDA Streams 76
5.2.7. NVIDIA GPUDirect 77
5.2.8. Thread Blocks in OpenACC Programming Model 79
5.3. NEC SX-Aurora Performance Tuning 79
5.3.1. PROGINF 80
5.3.2. FTRACE 80
5.3.3. veperf 85
5.3.4. Final Recommendations and Guidelines 86
5.4. Performance Analysis with Extrae/Paraver: use case with GROMACS 87
5.5. Debugging 89
5.5.1. NVIDIA Debugging Tools 89
5.5.2. AMD ROCm Debugger 90
5.5.3. TotalView 90
5.5.4. Vitis (HLS) 90
5.6. Summary 90
6. European Systems 92
6.1. JUWELS Booster 92
6.1.1. Programming Environment 93
6.2. Iris Cluster, Luxembourg University 93
6.2.1. Iris Cluster Overview 93
6.2.2. Network 95
6.2.3. Software/Modules Environment 96
6.3. BSC CTE-AMD 97
6.3.1. Programming Environment 98
6.4. Intel Hardware Accelerator Research Program (HARP) 99
6.5. NEC SX-Aurora TSUBASA Prototype System at HLRS 100
7. Brief Description of Used Applications/Benchmarks 102
7.1. HPCG Benchmark 102
7.2. GROMACS 102
A. Acronyms and Abbreviations 103
1. Units 103
2. Acronyms 103
Further documentation 106

1. Introduction

Hardware accelerators are special types of elements designed for boosting the performance of certain application regions requiring large amounts of numerical computations. Several factors contributed to broadening the use and furthering the adoption of these technologies in High-Performance Computing (HPC). One of such is the offered greater computational throughput as compared to stand-alone Central Processing Units (CPUs), which is driven by the highly parallel architectural design of accelerators. This is particularly important in the current era of ever-increasing computational demands featuring high reuse rates of compute-intensive operational patterns. Another contributing factor is that these specialized chips are also capable of delivering much higher compute performance as compared to CPUs under the same power budget, making these technologies even more appealing for system vendors and users. All these led HPC manufacturers and integrators to unleash further the potential of hardware accelerators for delivering the required compute performance more efficiently. In fact, this is one of the main reasons that the current Top500 list [1] continues to be enriched with various accelerated systems.

The next generation of HPC systems will also see a considerable amount of accelerator technology used. As a matter of fact, two out of the three European High-Performance Computing Joint Undertaking (EuroHPC JU) [2] pre-exascale HPC sites have already announced that their supercomputers will be equipped with large amount of Graphics Processing Units (GPUs). Thus, in order to achieve a competitive application performance and to be able to use the underlying hardware infrastructure efficiently, HPC application developers should be familiar with various challenges associated with using and orchestrating vast amounts of accelerator devices while being acquainted with the available ecosystem of the supporting tools.

This Best Practice Guide (BPG) extends the previously developed series of BPGs [3] by providing an update on new accelerator technologies to further support the European HPC user community in achieving outstanding performance records of their large-scale parallel applications. This guide follows the style of the previously published guide on "Modern Processors" [4], by providing a hybrid approach of a field guide and a textbook. The aim of this BPG is not to replace any of the available in depth textbooks and/or documentations of certain tools, but rather to provide a set of best practices that build upon the available literature and the expertise of authors involved to further ease the process of application porting and performance optimisation. This guide showcases the usability and possibilities of further application tuning given a specific accelerator technology, and does not provide any direct comparisons of different accelerator technologies involved. The guide provides a generic overview on various accelerators and their accompanying programming models/environments and thus should be viewed as complementary to the existing in-depth BPGs provided by hardware vendors that are typically specific to their own product.

This BPG starts with the description of the hardware for a selection of relevant accelerator technologies currently deployed at some PRACE sites, namely: GPUs, Field-Programmable Gate Arrays (FPGAs), and vector processors in Section 2. This is followed by the discussion on suitability of these accelerators for different HPC applications in Section 3.

This document then provides information on programming models (e.g. CUDA, SYCL, HIP, etc.) and development environments, as well as outlines some hints and best practices for application porting in Section 4. The strategies for application performance analysis and tuning, as well as the brief overview on the available debugging tools are presented in Section 5.

To further assist the effective usage of the available hardware infrastructure by the European HPC user community, this guide also provides brief information about various flagship and prototype HPC systems that are available at PRACE HPC sites and employ the discussed accelerator technologies in Section 6. Section 7 concludes this BPG by providing a brief description of the case applications used.

2. Architectures

This section introduces the architectures of the accelerator technologies covered in this guide.

2.1. GPUs

Initially, Graphics Processing Units (GPUs) were designed to be specialised electronic circuits for accelerating the production of visual output such as text, digital images, 2D geometric models, 3D models, and video games. In particular, the calculations involving the production of 3D graphics are computationally demanding but have potential for a very high level of parallelism. Hence, GPUs are designed to take advantage of this potential parallelism and this ability to quickly perform highly-parallel computations is exactly the reason why general-purpose GPUs, such as the data-center-optimised NVIDIA Tesla V100 GPU shown in Figure 1, have become so popular in HPC applications as well.

Figure 1. NVIDIA Tesla V100 SXM2 Module with Volta GV100 GPU [16]

When compared to CPUs, GPUs generally allocate more silicon space for units that perform computations and less silicon space to units that direct the computations. This allows GPUs to leverage more computing power from the available silicon space. However, this comes with a price as will be discussed in Section 3.2.1. Both NVIDIA and AMD GPUs follow a semi-modular design where the units performing computations, i.e. processing elements, are grouped into several computing units. Inside each computing unit, the processing elements share certain resources such as schedulers, dispatch units, registers, caches, fast local storage, load/store units, and texture units. Each computing unit contains dozens of processing elements and each GPU die contains dozens or hundreds of computing units. NVIDIA refers to the processing elements as CUDA cores and to the computing units as Streaming MultiProcessors (SMs). Figure 2 shows a block diagram for the NVIDIA Tesla V100 GPU. A full GV100 GPU is composed of 6 GPU Processing Clusters (GPCs) each of which contains 7 Texture Processing Clusters (TPCs). Each TPC contains two SMs. All SMs share a common level-2 cache that is connected through 8 memory controllers to 4 HBM2 memory modules.

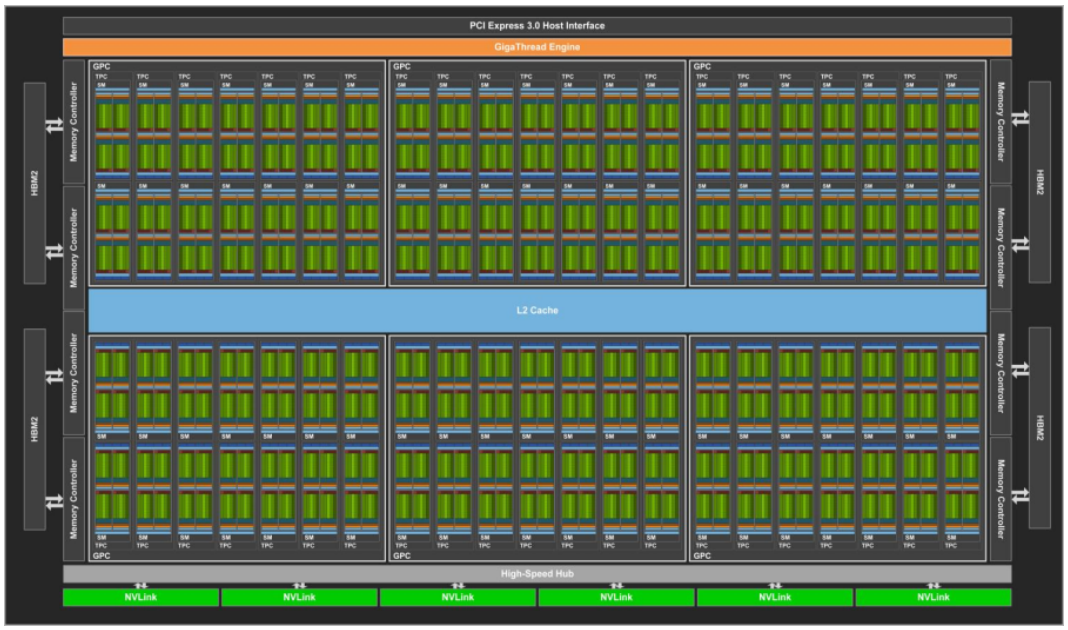

Figure 2. NVIDIA Volta GV100 GPU with 84 SMs [16]

As already mentioned, each computing unit (SM) is composed of several processing elements (CUDA cores) that share resources with each other. Different processing elements are designed to perform different functions. In case of the NVIDIA Volta GV100 GPU (see Figure 3) a SM includes 16 processing elements that can perform integer operations (INT), 16 processing elements that can perform 32-bit floating-point operations (FP32), and 8 processing elements that can perform 64-bit floating-point operations (FP64). Two large processing elements are optimised for specific HPC and AI related operations that may be performed with reduced precision (TENSOR CORE). A set of processing elements form a processing block which in addition to processing elements contains a level-0 instruction cache, a warp scheduler, a dispatch unit, a register file, several load/store units (LS/ST) and several special function units (SFU). All processing blocks share a level-1 instruction cache and a memory that can be dynamically partitioned between a fast local storage and a level-1 data cache. The processing elements in the same processing block are not completely independent of each other due to the fact that they share a scheduler and a dispatch unit. During each instruction issue time, the scheduler picks a set of 32 threads, called *warp*, that is ready to execute an instruction and issues the instruction to a set of suitable CUDA cores [15].

Figure 3. NVIDIA Volta GV100 SM [16]

Figure 4 shows a block diagram for an AMD Instinct MI100 GPU. The GPU is composed of 120 (enhanced) compute units (CUs) that are organised into four compute engines. All compute units share a common level-2 cache that is connected through several memory controllers to several HBM2 memory modules. Figure 5 shows a block diagram for an AMD Instinct MI100 compute unit. Similarly to the NVIDIA GPUs, the scheduler picks a wavefront (a set of 64 work-items, see Section 4.3) that is ready to execute an instruction and issues the instruction to a set of suitable processing elements. For scalar operations, each compute unit contains a scalar register file, a scalar execution unit, and a scalar data cache. For vector operations, each compute unit contains four large vector register files, four vector execution units that are optimised for 32-bit floating-point operations, and a vector data cache. The vector pipelines are 16-wide and each 64-wide wavefront is executed over four cycles. Each compute

unit also contains a matrix core engine that is optimised for specific matrix operations and reduced-precision computations.

Figure 4. AMD Instinct MI100 GPU [17]

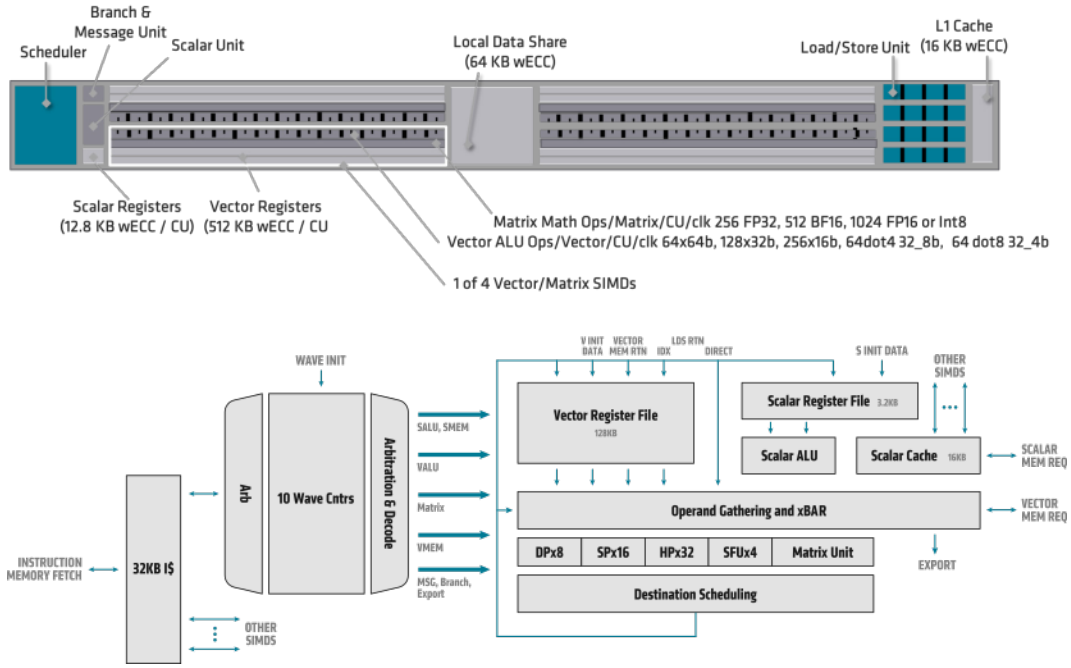

Figure 5. AMD Instinct MI100 computing unit [17]

2.2. FPGAs

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be designed and configured after manufacturing. Field-Programmable is a term from electronics that is applied to a device that can be programmed "in the field", as in outside of the factory.

FPGAs are typically composed of many reconfigurable logic blocks (see Figure 6) which contain the building blocks of integrated circuits, regarding logic (LookUp Tables or LUTs) and memory (Flip-Flops or FFs). These blocks are surrounded by programmable interconnect, and by configuring the contents of the blocks and the interconnect, it is possible to implement any circuit, within resource constraints.

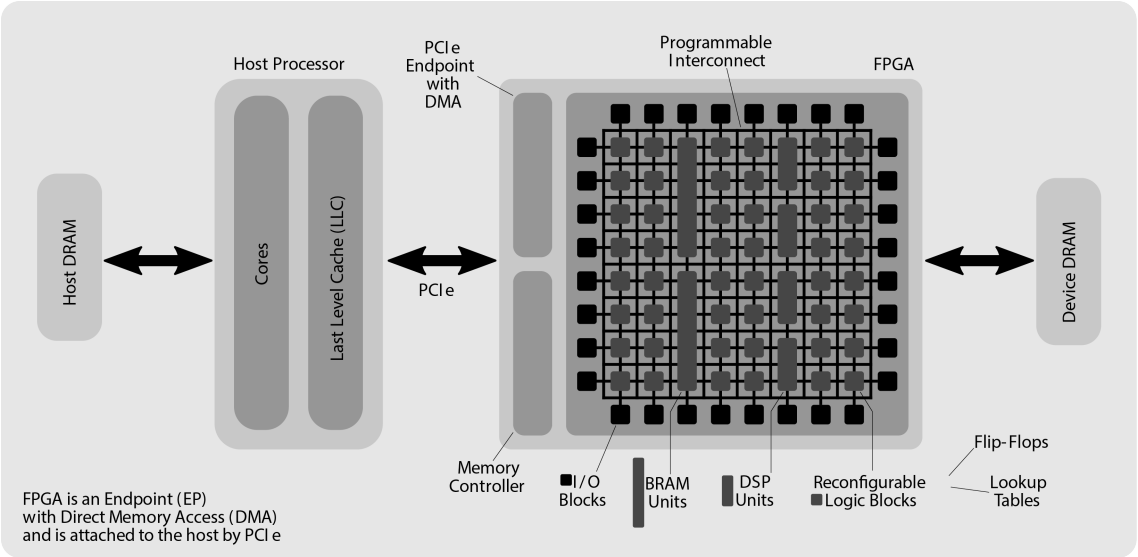

Figure 6. Architectural diagram for a typical modern FPGA expansion card [102]

Two factors led to the initial success of FPGA technology, lower overall costs for low-volume production, and rapid design and prototyping. FPGAs thus became a mainstay of the hardware development community, accompanying development advances in Electronic Design Automation (EDA), such as logic synthesis, which by the end of the 1990s had become necessary for all but the simplest projects. As FPGA usage grew to encompass increasingly compute-intensive applications (mostly due to the Internet-led growth of the communications industry), vendors began to incorporate discrete logic blocks, such as large on-chip memories and microprocessors, thus shifting away from the pure "gate array" [103].

Today's FPGA applications can broadly be grouped in two categories: the more traditional embedded device (now updated by concepts such as edge computing), and HPC acceleration (often also referred to as data centre acceleration). This paradigm treats FPGA platforms as hardware accelerators, just like GPUs or the newer tensor processing devices, which are controlled by a host device. These platforms thus take the form of an expansion card, connected e.g. via Peripheral Component Interconnect Express (PCIe), see Figure 7, although experimental platforms such as the Intel Hardware Accelerator Research Program (HARP) [112] puts in the same chip a tightly connected Intel CPU (e.g. Broadwell Xeon) and FPGA (e.g. Arria 10).

Figure 7. A PCIe FPGA accelerator card [104]

There are two main FPGA companies, Xilinx (recently acquired by AMD [105]) and Intel [106], each one with their corresponding products for HPC. There are also several companies (e.g. BittWare, Terasic) that provide Xilinx and Intel FPGAs packaged as PCIe expansion cards. Since with FPGAs it is possible to have custom circuitry around its input/output interfaces, it is common for FPGA cards to have high-speed Ethernet connectors (e.g. multiple QSFP28 4x 25Gb networking interfaces), which allow for data to be directly delivered from the Ethernet cable to the computation pipeline. Table 1 presents some characteristics of HPC-oriented FPGAs, which currently contain upwards of a million reconfigurable logic elements.

If you are interested in further details about FPGAs at an introductory level, we recommend the "For Dummies" book on the subject, which was written in partnership with Intel [114].

Table 1. Characteristics for a selection of HPC-oriented FPGA cards

	Alveo U250	Alveo U280	BittWare 520N
FPGA Chip	Xilinx UltraScale	Xilinx UltraScale	Intel Stratix 10 GX
Off-chip Memory Capacity	32 GB	32 GB + 8 GB HBM2	32 GB
Internal SRAM Capacity	54 MB	41 MB	32 MB
Network Interfaces	2x QSFP2 (100 GbE)	2x QSFP2 (100 GbE)	4x QSFP2 (100 GbE)
Logic Elements	1.3 million	1.1 million	10.2 million

2.3. NEC SX-Aurora TSUBASA

The SX-Aurora TSUBASA [75] is an accelerator developed by NEC. In contrast to its predecessor, SX-ACE [76], which functioned stand-alone, SX-Aurora complements an x86 host processor. As for most accelerators, the interface between them is PCIe 3.0 [77]. However, unlike NVIDIA’s and AMD’s graphics processing units, NEC's SX-Auroras are vector engines, which feature extra-wide vector instructions instead of warp- or wavefront-based threading models.

Different models with varying numbers of cores exist but the structure of the cores itself is the same in every model. Each core contains several computing units and 64 vector registers that provide 256 × 64 bit [78]. The three Fused Multiply–Add (FMA) pipes support 32-way parallelism which enables $3 \cdot 2 \cdot 32 = 192$ floating point operations per cycle in double precision, as indicated by Figure 8. Depending on the clock speed and the number of cores of a specific model, the total peak performance per device is between 2.15 and 3.07 TFLOPS [79]. Table 2 shows the available SX-Aurora types.

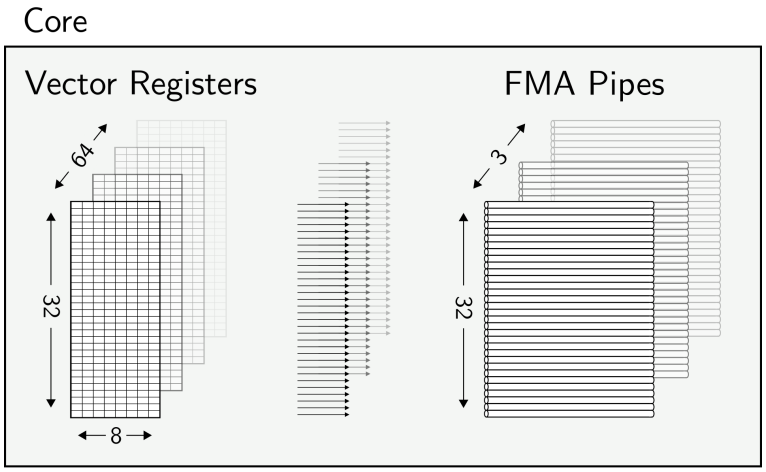

Figure 8. NEC SX-Aurora TSUBASA's vector registers and FMA pipes

Table 2. Available models for NEC SX-Aurora TSUBASA [79]

	Type 10AE	Type 10BE	Type 10B	Type 10CE	Type 20A	Type 20B
Clock speed (GHz)	1.584	1.408	1.400	1.400	1.600	1.600
# of cores per processor	8	8	8	8	10	8
DP performance (TFLOPS)	2.43	2.16	2.15	2.15	3.07	2.45
SP performance (TFLOPS)	4.86	4.32	4.30	4.30	6.14	4.91
Memory bandwidth (TB/s)	1.35	1.35	1.22	1.00	1.53	1.53
Cache capacity (MB)	16	16	16	16	16	16
Memory capacity (GB)	48	48	48	24	48	48

The memory is based on 6 modules of HBM2 (4 modules for Type 10CE). As shown in Figure 9, each module provides a capacity of 8 GB at more than 200 GB/s of bandwidth. In total this accumulates to 48 GB at more than 1,200 GB/s. A shared last level cache (LLC) is located between the cores and HBM2. It offers 16 MB, has a "write-back" policy and a line-size of 128 bytes. Since every core can access the LLC with a bandwidth of more than 400 GB/s, applications should be optimised with respect to cache usage.

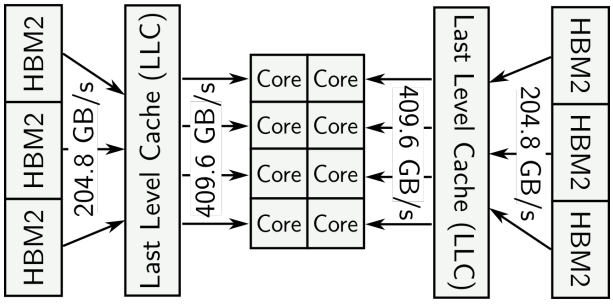

Figure 9. Memory hierarchy of SX-Aurora inspired by [80]

The Vector Engine (VE) requires a Vector Host (VH) in the form of an x86 processor. Message Passing Interface (MPI) based communication can be performed through the vector host or, once a connection has been set up, directly through a PCIe switch to an InfiniBand (IB) network, as shown in Figure 10.

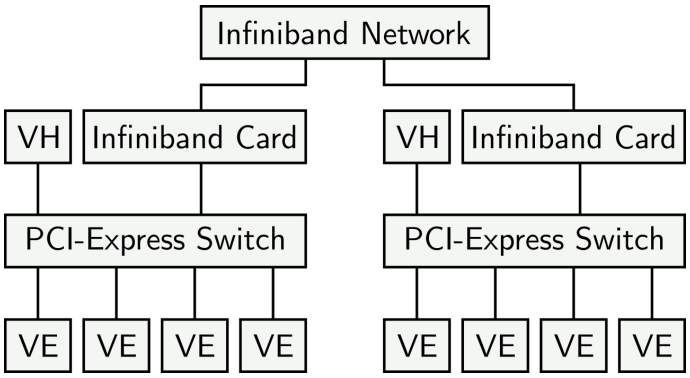

Figure 10. Network integration inspired by [78]

NEC offers different server-packages that include vector engine cards. They range from one vector host with one vector engine (home tower solution) to 8 vector hosts with 64 vector engines (supercomputer). The hardware used for the demonstrations in this guide is presented in Section 6.5, "NEC SX-Aurora TSUBASA Prototype System at HLRS".

3. Suitability of accelerators for different HPC codes

Compared to Central Processing Units (CPUs), accelerator devices, such as GPUs, FPGAs and vector processors, provide both significantly higher instruction throughput and memory bandwidth. In particular, this performance is delivered in a relatively small price and power envelope as shown in Table 3. For example, an NVIDIA Tesla V100 GPU can deliver 0.7 GFLOPS (= 0.7×10^9 Floating Point Operations Per Second (FLOPS)) for each Euro and 28 GFLOPS for each watt of power, whereas a typical 14-core Intel Skylake server CPU can deliver 0.6 GFLOPS for each Euro and 8.6 GFLOPS for each watt of power. However, each type of accelerator favours specific kinds of computation patterns, and it is important to choose the right accelerator for the code that we want to accelerate. If a computation pattern is not a good fit for an accelerator, the most likely result is to see significant performance degradation, when compared with running the same code on a CPU. This section introduces the roofline model, a tool that allows to characterise an application regarding computation and memory access intensity, and provides some remarks about what kind of patterns best fit each kind of accelerator covered in this guide. This section does not aim to provide any comparisons among the accelerator technologies discussed, but rather to discuss their suitability based on the computation and memory access patterns of target applications.

Table 3. Indicative performance, price (year 2020-2021) and power figures for different CPUs, GPUs, FPGAs and vector processors [8, 9, 10, 11, 12, 13] . The number format is indicated in the parentheses (dp = double-precision floating-point format, sp = single-precision floating-point format and int8 = 8-bit integer format)

Device	Throughput (GFLOPS)	Bandwidth (GB/s)	Price (€)	Power (W)
4-core Intel Skylake desktop CPU	200 (dp)	35	300	90
14-core Intel Skylake server CPU	1200 (dp)	100	2000	140
NVIDIA Tesla V100 GPU	7000 (dp)	900	10000	250
Xilinx Alveo U250 FPGA	33300 (int8)	77	6000	225
Xilinx Alveo U280 FPGA	24500 (int8)	460	6000	225
BittWare 520N Stratix10 GX2800 FPGA	9200 (sp)	77	10000	225
NEC SX-Aurora TSUBASA Type 10AE vector processor	2430 (dp)	1350	6000	300

3.1. Roofline Model

The performance figures indicated in Table 3 are only rarely attainable in practice. One of the more powerful tools for understanding the *attainable performance* is the concept of *arithmetic intensity*. Arithmetic intensity is calculated separately for each algorithm and is defined as the ratio between the work (i.e. the number of arithmetical operations performed) and the memory traffic (i.e. the number of bytes of data transferred). This section mainly discusses the work in terms of FLoating-point OPerations (FLOP) but the techniques described in this section can be applied to integer operations, etc. As an example, consider a function that computes a vector-vector addition:

```
void axpy(int n, double alpha, double *x, double *y)
{
    for (int i = 0; i < n; i++)
        y[i] = y[i] + alpha * x[i];
}
```

The total number of floating-point operations performed is 2 × n (one multiplication and one addition per row) and the total number of bytes transferred is 3 × 8 × n (two 8-byte loads and one 8-byte store per row). Hence, the arithmetic intensity of the function is (2 × n) FLOP / (3 × 8 × n) Byte = 1/12 FLOP/Byte. Note that each FMA operation counts as two floating-point operations.

The arithmetic intensity of an algorithm is not always trivial to compute. For example, a certain fraction of the data is frequently stored in a cache or a specialised memory region that is significantly faster than the main memory of the device. It is thus common to compute the arithmetic intensity only with respect to the main memory data transfers, i.e. cached data transfers are ignored in the calculations. Figure 11 gives indicative arithmetic intensities for vector-vector addition (AXPY), Sparse Matrix-Vector Multiplication (SpMV), stencil codes, Fast Fourier Transformation (FFT), GEneral Matrix to Matrix Multiplication (GEMM), particle codes, and Deep Neural Networks (DNN). In general, the arithmetic intensity increases as more data reuse is introduced. The vector-vector addition operation does not involve a lot of data reuse as each row of the vector *x* is accessed only once and each row of the vector *y* is accessed twice. On the other hand, matrix-matrix multiplication, particle codes and deep neural networks are usually implemented in such a way that the data is partitioned into small batches, piece-by-piece loaded to caches or specialised memory regions, and then repeatedly accessed.

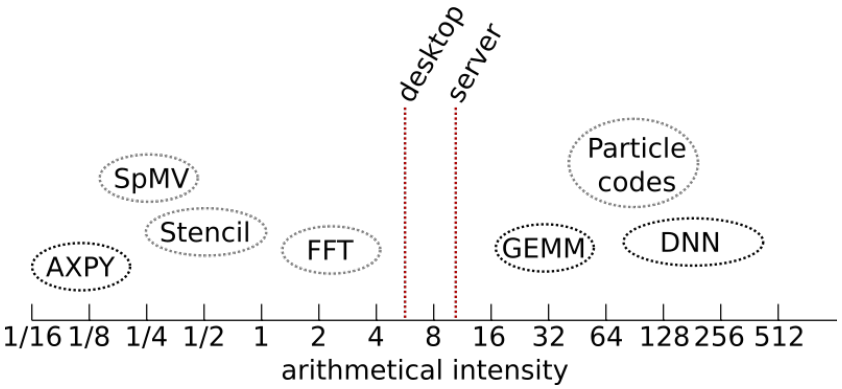

Figure 11. Indicative arithmetic intensities for various algorithms and ridge points for the two CPUs included in Table 3

The arithmetic intensity of an algorithm itself does not provide any prediction for the attainable performance on a given device. In order to connect the performance figures shown in Table 3 to the arithmetic intensities shown in Figure 11, we must calculate a *ridge point* (a.k.a. an optimal arithmetic intensity) for each device. The ridge point of a device is defined as the ratio between the peak performance (i.e. the number of arithmetic operations the device can perform per second) and the peak bandwidth (i.e. the number of bytes of data the device can transfer per second). Figure 11 gives ridge points for the two CPUs included in Table 3. For example, the desktop CPU can perform 200×10^9 FLOPS and transfer 35×10^9 Byte/s, and thus has the ridge point at 5.7 FLOP/Byte. Similarly, the server CPU can perform 1.2×10^{12} FLOPS and transfer 100×10^9 Byte/s, and thus has the ridge point at 12 FLOP/Byte.

Algorithms that have lower arithmetic intensity than the ridge point of a device are said to be *memory bound* on the device as the attainable performance is always going to be limited by the peak bandwidth. More precisely, for memory-bound algorithms, there exists a linear relation between the arithmetic intensity and the attainable performance (attainable performance = peak bandwidth × arithmetic intensity) as visualised in Figure 12. Algorithms that have higher arithmetic intensity than the ridge point of a device are said to be *compute bound* on the device as the attainable performance is always going to be limited by the peak performance. When the attainable performance is plotted against the arithmetic intensity, as done in Figure 12, the end result is a predictive model that resembles a roofline. Therefore, it is not surprising that this model is called the *roofline model* [14] .

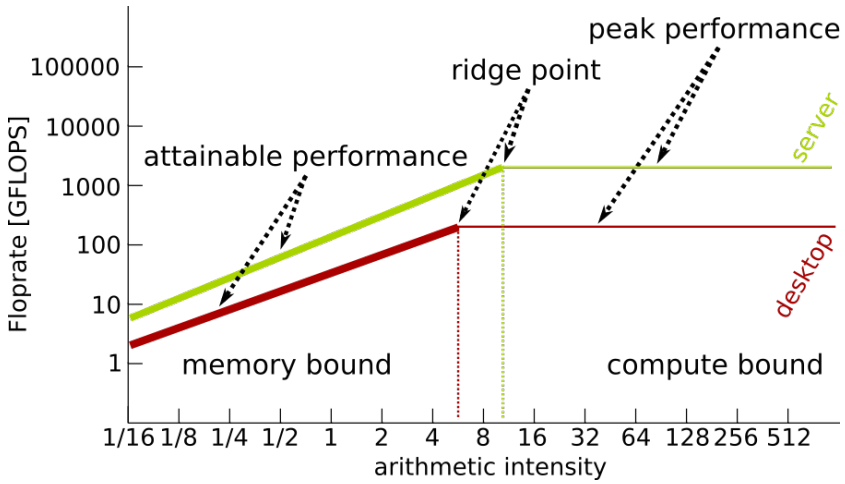

Figure 12. Obtainable performance for the two CPUs included in Table 3

Figure 13 shows CPU roofline models for various algorithms. We can see that vector-vector addition, sparse matrix-vector multiplication, stencil codes and fast Fourier transformation are memory bound on both CPUs. For example, for the desktop CPU, the roofline model predicts that the vector-vector addition can reach the performance of 35 GB/s × 1/12 Flop/Byte = 2.9 GFLOPS. Similarly, for the server CPU, the roofline model predicts the performance of 100 GB/s × 1/12 FLOP/Byte = 8.3 GFLOPS. On the other hand, matrix-matrix multiplication, particle codes and deep neural networks are compute bound on both CPUs. That is, the roofline model predicts that all three algorithms can reach the peak performance on both CPUs. As a general rule, it is advisable to strive to increase the arithmetic intensity of an algorithm because this often leads to a higher performance as predicted by the roofline model. Usually this means that an algorithm should attempt to utilise on-chip memory as much as possible as this reduces the number of main memory transfers. See Section 4.2, “CUDA”, Section 4.3, “OpenCL” and Section 5.2.2, “Shared Memory Programming” for further technical information.

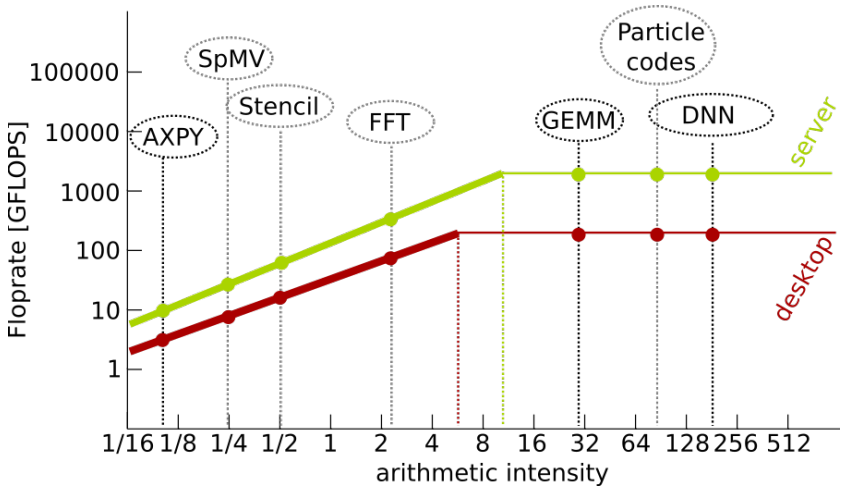

Figure 13. Roofline models for the two CPUs included in Table 3

It should be noted that several additional factors affect the performance of an algorithm (e.g. NUMA configuration, cache hierarchy, cache size, software prefetching, vectorisation, and instruction-level parallelism) and a prediction provided by a roofline model is not therefore expected to tell the complete story. Furthermore, even if the roofline model accounted for all relevant factors, a code usually requires a lot of additional tuning before it can reach the performance predicted by the roofline model. However, the obtained performance of an algorithm cannot exceed a prediction provided by a correctly formed roofline model.

3.2. Accelerator Suitability

3.2.1. GPUs

Let us first consider the roofline model and investigate how different algorithms are expected to perform on a NVIDIA Tesla V100 GPU (see Figure 14). Since the V100 GPU can perform 7×10^{12} FLOPS and transfer 900×10^{9} Byte/s, its ridge point is at 7.8 FLOP/Byte. This means that the V100 GPU is balanced very similarly to the desktop CPU (5.7 FLOP/Byte) whereas the server CPU (12 FLOP/Byte) is better balanced for algorithms that have higher arithmetic intensity. The roofline model predicts that the vector-vector addition operation can reach the performance of 900 GB/s × 1/12 FLOP/Byte = 75 GFLOPS. This is far from the peak performance of the V100 GPU (7000 GFLOPS) but much higher than either CPU can achieve (2.9 GFLOPS and 8.3 GFLOPS). This is a very important observation to internalise since some algorithms can be faster on a GPU because the GPU has a much higher peak performance and some algorithms can be faster on a GPU because the GPU has a much higher peak bandwidth. For some algorithms, both factors are relevant.

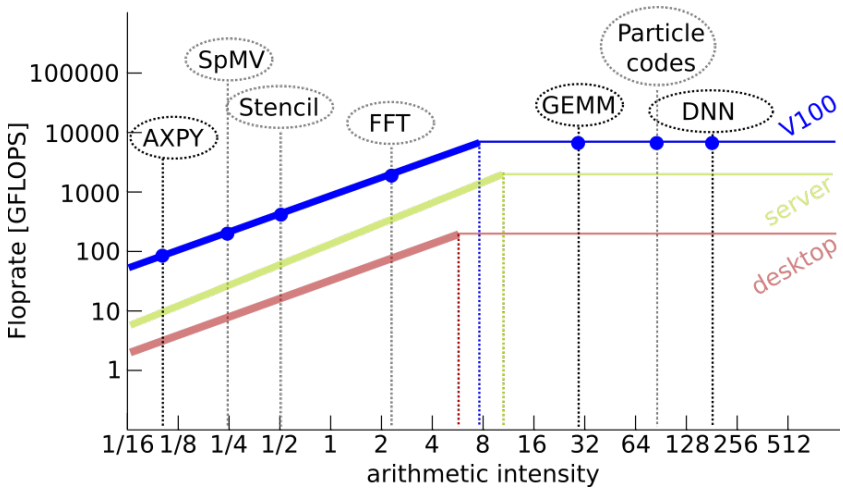

Figure 14. Roofline models for the NVIDIA Tesla V100 GPU. The CPU roofline models are included for scale only.

A modern GPU consists of thousands of cores that are connected to several memory controllers and memory banks, as discussed in Section 2.1. For complex scheduling-related reasons, the cores should be oversubscribed with an excess number of threads. This increases the *occupancy* of the cores which further helps keeping the computation pipelines full and reduces the effects of memory access latencies (see Section 5.2.1, “CUDA Occupancy Calculator”). Therefore, as visualised in Figure 15, the number of threads running on a GPU must be extremely large. Otherwise only a fraction of the attainable performance can be achieved. If an algorithm cannot be parallelised with a sufficient number of threads, then a GPU is not a suitable computational resource for the application.

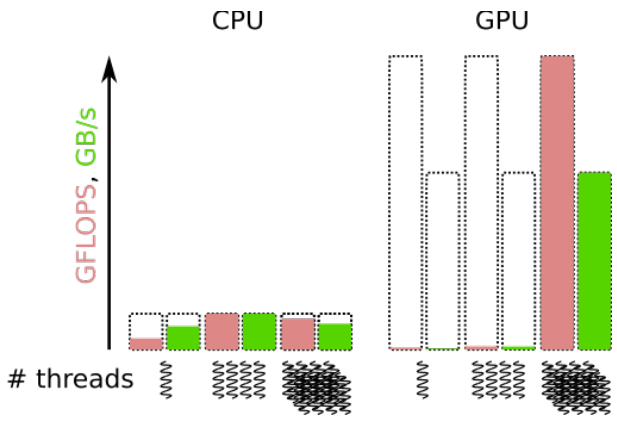

Figure 15. An illustration of the number threads required for optimal performance on a GPU

A GPU is typically connected to a CPU (and the RAM) through a PCIe bus. The bandwidth of the PCIe bus is very limited when compared to the memory bandwidth of the internal memory of a GPU (V100: 900 GB/s). For example, a 16-line PCIe 3.0 bus has a bandwidth of 32 GB/s (16 GB/s dual simplex). If we assume that both vectors in a vector-vector addition operation are stored in RAM instead of GPU's internal memory, then the roofline model must be re-calibrated as visualised in Figure 16. In this case, the roofline model predicts that the vector-vector addition operation can reach the performance of 32 GB/s × 1/12 FLOP/Byte = 2.7 GFLOPS. The conclusion is that GPUs should be used only in applications where the data is accessed several times after it has been transferred to GPU's internal memory. The same applies to most accelerator devices that are connected to the main memory though a PCIe bus.

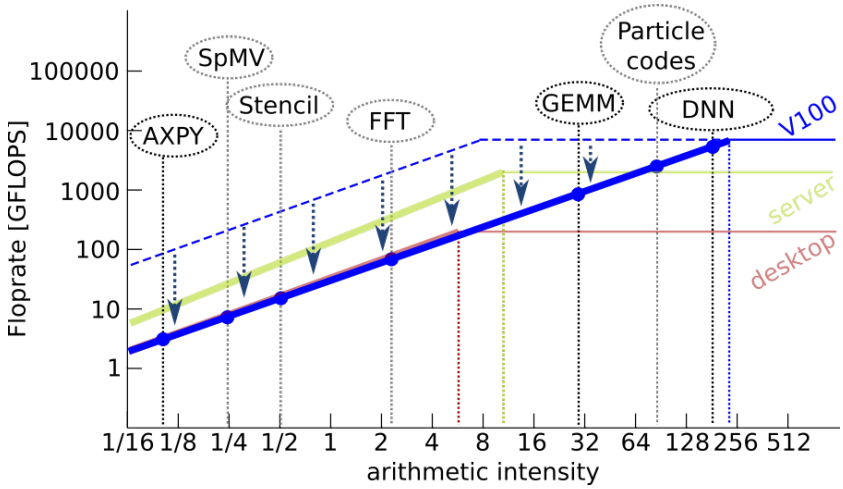

Figure 16. Roofline models that assume that the data is accessed through a PCIe bus. The CPU roofline models are included for scale only

The final major issue to consider is the manner in which the threads interact with each other. As already mentioned in Section 2.1, “GPUs”, the cores are not fully independent of each other. Instead, on the hardware level, a set of cores executes a common instruction together in a manner similar to a vector processor. This is not immediately visible to a programmer but it does affect the performance. For example, a modern NVIDIA GPU divides the threads to groups of 32 threads called warps. If one or more threads within a warp diverge from the rest of the warp at a branching point, then the overall cost of this divergence is almost the same as if all threads within the warp had executed all involved paths. Even if the probability of a thread within a warp diverging from the rest of the warp is very low, the overall effect can be quite large as shown in Figure 17. The depths of the diverging execution paths (i.e. how many instructions each path contains), and the type of operations performed within the paths, affect the overall cost. For example, threads that are inactive in an execution path do not trigger any memory transfers, but threads within a warp should access the memory in a *coalescing* manner. That is, when a warp accesses the memory, the resulting access pattern should target as few continuous chunks of memory as possible. Therefore, GPUs are not suitable for applications that involve a lot of diverging execution paths.

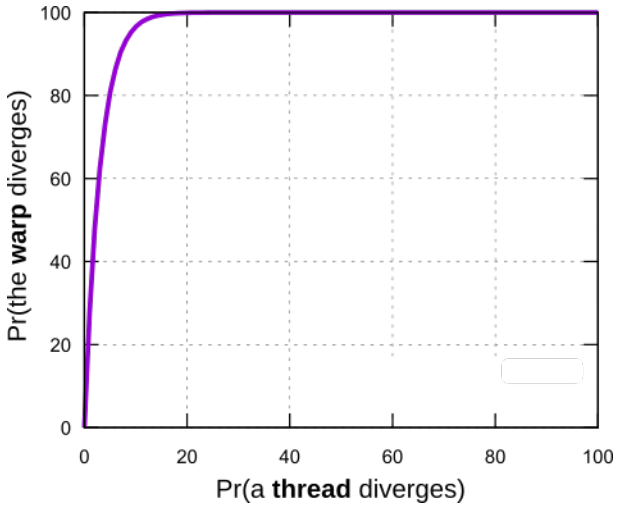

Figure 17. The probability that a warp diverges as a function of the probability that a thread within the warp diverges, i.e, Pr(the warp diverges) = [1 - (1 - Pr(a thread diverges) / 100%)^{32}] × 100%. For example, if a thread diverges with the probability of 1%, then the warp diverges with the probability of [1 - (1 - 0.01)^{32}] × 100% = 27.5%

3.2.2. FPGAs

Since FPGAs do not have a fixed architecture, theoretically they can implement any model of computation, including the same models as CPUs and GPUs. However, reconfigurability has an overhead such that, for a given design, when compared with a fixed architecture (e.g. implemented as an ASIC), it will most likely use more physical resources, and obtain lower clock speeds.

The roofline model in Figure 18 shows that FPGAs are more suited for workloads with high arithmetic intensity. On the one hand, this is due to the high number of available resources that can be potentially used to perform computation. On the other hand, historically FPGAs have had low internal bandwidth for its larger pool of RAM when compared with other accelerators, which pushes them towards the end of the arithmetic intensity axis (very high-bandwidth custom memories can be instantiated inside the FPGAs, but they are extremely limited in size). However, the memory issue is currently being addressed, for instance, with the use of High Bandwidth Memories (HBM), as can be seen in the case of the Alveo U280, which in the graph is the FPGA that is further left in the arithmetic intensity axis.

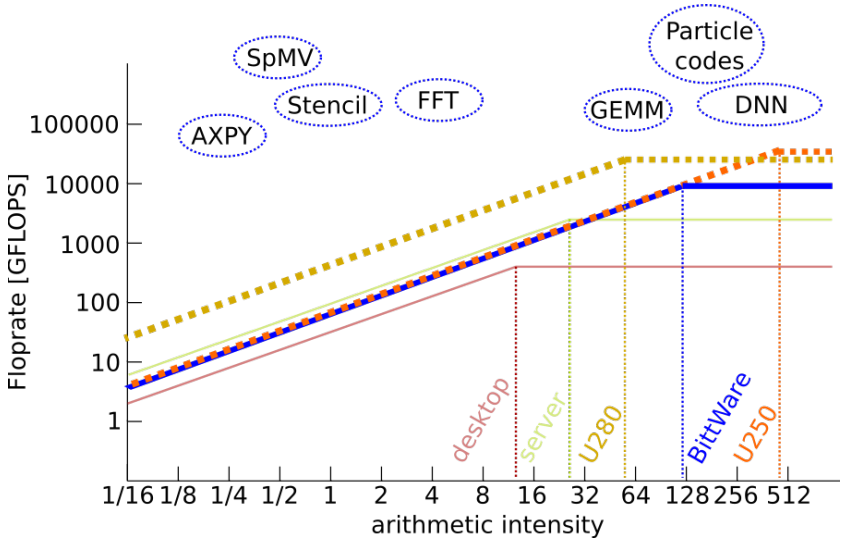

Figure 18. Roofline models for the Alveo U250, U280 and BittWare 520N Stratix10. The CPU roofline models are included for scale only

Please note that the FLOP rates presented for FPGAs are measured not only in single precision (e.g. BittWare), as the CPU models in the graph, but can also appear as integer operations (e.g. U250, U280). One reason for this is that FPGAs are commonly used for custom-precision computations, where the computational units are designed to implement just enough bits to carry out the computations at a sufficient accuracy.

FPGAs excel at streaming computation[1] and deep pipelines, and can provide a computational edge if an algorithm can take advantage of these paradigms. Even when FPGAs are not able to achieve the performance of other accelerators such as GPUs, they can usually provide solutions that consume less energy overall, a pressing issue for HPC centers..

If loop iterations are independent, it is possible to build deep, custom pipelines that process the several iterations simultaneously, in lock-step, and after the first iteration finishes, the time to finish the next one usually only depends on the latency of the slower stage in the pipeline. Pipelines can be replicated for extra parallelism (resources allowing), and coupled with streaming, FPGAs can deliver highly-efficient designs that do not need to decode instructions at each clock cycle, as it happens in CPUs and GPUs.

3.2.3. Vector Processors

Vector Processors exist in many different varieties. Most modern CPUs support vector computing are based on the form of SIMD instructions. The way GPUs organise their workflow in warps (NVIDIA) or wavefronts (AMD) can also be interpreted as vector processing. However, in this section we consider classical vector computing architectures, in particular NEC SX-Aurora TSUBASA. Vector supercomputers have been popular in the 1970s and 1980s but NEC has been the only vendor to develop new vector architectures for supercomputing in recent years.

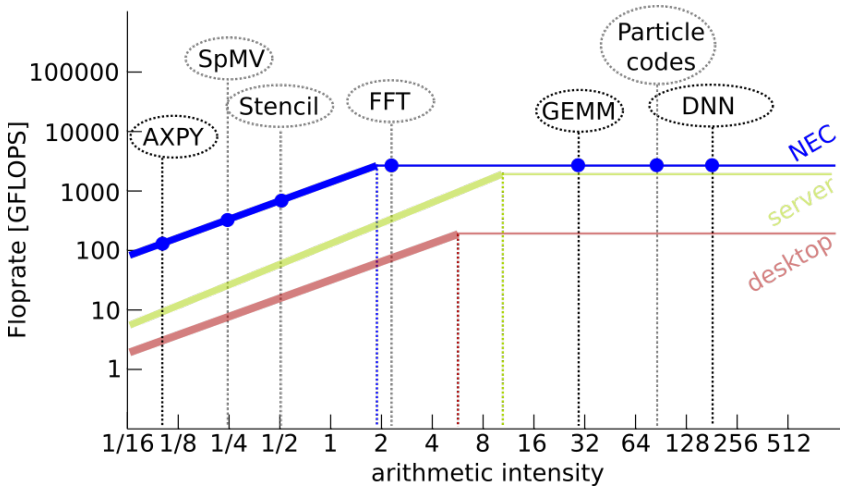

Figure 19. Roofline models for the NEC SX-Aurora TSUBASA Type 10AE. The CPU roofline modes are included for scale only

The roofline model in Figure 19 shows that SX-Aurora TSUBASA excels for workloads with low arithmetic intensity. It shares this property with GPUs because both architectures use on-chip HBM2. Sparse matrix-vector multiplications and stencil operations are often representative for traditional HPC tasks, such as CFD simulations. These are therefore a great fit for modern vector processors. Naturally, SX-Aurora can also perform workloads with high arithmetic intensity and TensorFlow, for example, was optimised by NEC for the vector computers [93]. Nevertheless, the largest supercomputer that currently features SX-Auroras [94] is #33 (HPL) and #10 (HPCG) on the Top500 list, which indicates that NEC's newest vector processors are more competitive for memory-intensive workloads.

Vector processors inherently favour uniform and contiguous memory access patterns because vector instructions can hardly be generated for randomly scattered data. Considering numerical simulation in this regard, the NEC SX-

[1] In streaming computation, data is directly given to computational units at the beginning of each iteration, instead of being randomly accessed from a pool of memory.

Aurora is most likely to be better suited for applications featuring structured grids in comparison to unstructured grids. For more information concerning vector processors please refer to Sections 2.3, 4.8, and 5.3.

3.3. Summary

The following bullet list highlights the most important aspects discussed in this section.

- A prediction provided by a roofline model is an approximation, and provides only an upper bound to attainable performance.
- In general, a high arithmetic intensity leads to higher instruction throughput but not necessarily to an overall higher performance. A low arithmetic intensity usually implies that the performance is going to be limited by the available memory bandwidth.
- Memory accesses that pass through the PCIe bus are generally slower than memory accesses that target the accelerator's internal memory. This effect is particularly strong if the algorithm has a low arithmetic intensity.
- GPUs and vector processors rely on (implicit) vector operations. The involved arithmetic operations and memory accesses should be vectorisable (and coalescing).
- GPUs require an extremely high level of concurrency (thousands of threads) for optimal performance.
- GPUs and NEC vector processors are similar with respect to suitability because they both rely on PCIe, HBM2 and SIMD technology. The main difference is that GPUs tend to be more expensive but also provide more peak performance.
- FPGAs excel at streaming computation and deeply pipelined algorithms, and are more suited for workloads with high arithmetic intensity.
- FPGAs can usually provide solutions that consume overall less energy.

4. Programming Models and Environment

This section provides an overview of the available software ecosystems for different accelerator technologies. In the first section, we discuss the portability of the programming models across the accelerators, the compatibility among programming models, and their advantages and disadvantages. Then, in the remaining sections, we present an introduction to each programming model, with some useful hints and with complete examples that can be compiled to execute on the target device. Please note that only basic concepts are introduced here, which may not be sufficient for becoming proficient on these programming models. The reader should refer to specialised documentation of each programming model.

4.1. Introduction

The languages and programming models presented here are used to program the three types of device architectures, discussed in Section 2, “Architectures”, namely, GPUs, FPGAs and the NEC accelerators. Table 4 presents the programming languages that are commonly used for developing programs for accelerators (C/C++ and Fortran), and the portability of each programming model across the accelerator types. The host processor, CPU, is also included to show that programs written for accelerators that use OpenCL, SYCL, or OpenACC can run on the host without modifications. This is an important characteristic for performing program validation and correctness tests.

CUDA code can only run on NVIDIA GPUs and its main advantage is to explore better the characteristics of these devices, which allows to achieve higher performance rates as compared with other tools running on NVIDIA devices. OpenCL was developed with the purpose of portability, which is its main advantage, and can be used to program all the devices but the NEC accelerator. Note that code that has been optimised for one architecture does not necessarily run well on a different architecture. SYCL runs over the OpenCL runtime and therefore provides the same functionality and performances. It is mainly a C++ wrapper that encapsulates the verbose syntax of OpenCL. OpenACC is a pragma based approach that does not require rewriting existing sequential base code to run on a GPU accelerator. The high level approach and portability are its main advantages for the programmer, however imposing limitations on the attainable performance. HIP is the programming model developed by AMD for their GPUs, although it can also run on NVIDIA GPUs. The main advantage of HIP is the attainable higher performance on AMD GPUs. HLS refers to a higher-level approach to FPGA programming, which includes using languages such as C++ or OpenCL to generate hardware modules. It is only applicable for FPGA programming, which is programmed at the circuit level. The NEC device is a vector processor and does not need any particular programming model. The NEC compiler generates the executable code from C++ or Fortran programs, with no specific modifications, introducing vector instructions when adequate. There is another programing model, OpenMP, which provides extensions to program GPUs, but its main target and features are for exploiting parallelism on multicore CPUs, and therefore it is not considered here.

Table 4. Programming languages and programming models portability matrix

	C/C++	Fortran	CUDA	OpenCL	SYCL	OpenACC	HIP	HLS
x86 CPU	Yes	Yes	No	Yes	Yes	Yes	No	No
NVIDIA GPU	No	No	Yes	Yes	Yes	Yes	Yes	No
AMD GPU	No	No	No	Yes	Yes	Yes[a]	Yes	No
FPGA	No	No	No	Yes	Yes	No	No	Yes
NEC	Yes	Yes	No	No	No	No	No	No

[a] As of 2020, the compiler support is still immature.

Table 5 presents a compatibility matrix between the various programming models, where one can see the degree of adaptation required to transform a program written in one programming model to another. *Complete rewrite* indicates that the kernel to accelerate and the main code need to be rewritten. *Keep structure* indicates that the kernel needs to be rewritten but the overall design and structure of the code can be kept (i.e. an automatic conversion is theoretically possible but no good tool exists). *Add code* indicates that new code needs to be written. *Add pragmas* indicates that pragmas need to be added. *Convert tool* indicates that a conversion tool exists. *Convert macros* indicates that a macro-based conversion tool exists. From this table, one can observe that a C/C++ program

can be the base for any of the accelerators programming models. The other transformations can be achieved by keeping the structure of the code, or require a complete rewrite. The HLS methodology is particular to FPGA programming and there is no equivalence to the other devices.

Table 5. Programming model compatibility matrix

From \ To	CUDA	OpenCL	SYCL	OpenACC	HIP	HLS
C/C++	Add code	Add code	Add code	Add pragmas	Add code	Add code
Fortran	Add code	Complete rewrite	Complete rewrite	Add pragmas	Complete rewrite	Complete rewrite
CUDA		Keep structure	Keep structure	Complete rewrite	Convert tool	Complete rewrite
OpenCL	Keep structure		Keep structure	Complete rewrite	Keep structure	Keep structure
SYCL	Keep structure	Keep structure		Complete rewrite	Keep structure	Keep structure
OpenACC	Complete rewrite	Complete rewrite	Complete rewrite		Complete rewrite	Complete rewrite
HIP	Convert macros	Keep structure	Keep structure	Complete rewrite		Complete rewrite

As OpenCL supports most considered architectures, Table 6 presents an OpenCL portability matrix between the various architectures. *Kernel redesign* indicates the kernels (see Section 4.3.2, “Host and kernels”) must be re-designed for optimal performance. *Minor modifications* indicates the kernels require minor modifications for optimal performance.

Table 6. OpenCL portability matrix

From \ To	x86 CPU	NVIDIA GPU	AMD GPU	FPGA
x86 CPU		Kernel redesign	Kernel redesign	Kernel redesign
NVIDIA GPU	Kernel redesign		Minor modifications	Kernel redesign
AMD GPU	Kernel redesign	Minor modifications		Kernel redesign
FPGA	Kernel redesign	Kernel redesign	Kernel redesign	

OpenACC is also easily portable between the most common of considered architectures. In most cases porting an application from one architecture to another consists of changing the compiler flag, e.g. for the NVidia compiler, *nvc -acc=gpu* into *nvc -acc=multicore*, which will make the compiler parallelise a code on CPU instead of NVIDIA GPU. There is a possibility of tuning some other flags to improve the performance on targeted architecture, e.g. *nvc -add=gpu -gpu=cc70*, which will parallelise code on GPU and target the specific GPU microarchitecture, Volta in this case. Most compilers that support NVIDIA or AMD GPUs also provide CPU support. Porting code from NVIDIA GPU to AMD GPU requires a change of compilation flags and change the compiler itself, i.e. from *nvc* to *gcc*.

4.2. CUDA

The *Compute Unified Device Architecture* (CUDA) [15] is a proprietary parallel computing platform and an API created by NVIDIA. Since CUDA is being developed by NVIDIA themselves and only supports NVIDIA GPUs, CUDA can be used to access the entire feature set of modern NVIDIA GPUs. Other programming models, such

as OpenCL (see Section 4.3), HIP (see Section 4.6), OpenACC (see Section 4.5) and OpenMP, implement only a subset of this feature set. For example, the pragma-based approaches (OpenACC and OpenMP) do not allow a programmer to directly access the shared memory (see Section 4.2.5). CUDA can be used:

- directly through CUDA C/C++ and CUDA Fortran;
- directly through wrappers (Python, Perl, Fortran, Java, Ruby, Lua, etc);
- indirectly through compiler directives (OpenACC, OpenMP); and
- indirectly through other computational interfaces (OpenCL, DirectCompute, OpenGL, etc).

In this subsection, we will only discuss CUDA C/C++. However, CUDA Fortran and the various wrappers are built around the same core concepts.

4.2.1. Vendor libraries

The complete CUDA SDK comes with a large set of vendor-optimised libraries [18], most notably:

- CUDA Runtime library (CUDART), basic functionality.
- CUDA Basic Linear Algebra Subroutines (cuBLAS).
- CUDA Fast Fourier Transform library (cuFFT).
- CUDA Random Number Generation library (cuRAND).
- CUDA Sparse Matrix library (cuSPARSE).
- CUDA Direct Linear Solvers library (cuSOLVER).
- CUDA Tensor Linear Algebra library (cuTENSOR).
- CUDA Deep Neural Network library (cuDNN).

Also, many 3rd party libraries use these vendor libraries to accelerate their performance. In most cases, a reader is recommended to use the vendor-optimised libraries instead of implementing their own.

4.2.2. Host, devices and NVIDIA CUDA Compiler

CUDA comes with its own compiler called *nvcc*. A part of the program is always executed on the *host*, i.e. on the CPU. The nvcc compiler passes this part of the program to the host C++ compiler (e.g. GNU g++). The part of the program that is meant to be executed on a *device*, i.e. on a GPU, is written in an extended dialect of C++ and compiled to *PTX* (Parallel Thread Execution) code. The PTX code is either compiled to an architecture-specific binary and/or kept as is for later compilation by the graphics driver. The nvcc compiler links (using nvlink linker program) the host-side code and the device-side code into a single binary or a library.

In order to better understand what this means in practice, consider the following "Hello world" program (hello.cu):

```
#include <stdio.h>

__global__ void say_hello()
{
    printf("A device says, Hello world!\n");
}

int main()
```

```
{
    printf("The host says, Hello world!\n");
    say_hello<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

On a linux machine, the above "Hello world" example is compiled and executed as follows:

```
$ nvcc -o hello hello.cu
$ ./hello
The host says, Hello world!
A device says, Hello world!
```

It is also possible to pass additional arguments to the host compiler (g++ in this example):

```
$ nvcc -o hello hello.cu -Xcompiler="-Wall"
```

This passes the *-Wall* flag to the host compiler and causes the host compiler to print additional warnings relating to the host-side code. Thus, it is suggested to use this flag when generating a high quality code.

Each NVIDIA GPU has a *compute capability* classification that describes the capabilities of the GPU. For example, NVIDIA Tesla V100 GPU has the compute capability of 7.0. The following command embeds the resulting program binary with pre-compiled binary code that is compatible with compute capability 5.0 and 6.0 (second and third row), and PTX and pre-compiled binary code that is compatible with compute capability 7.0 (fourth row):

```
$ nvcc -o hello hello.cu \
    -gencode arch=compute_50,code=sm_50 \
    -gencode arch=compute_60,code=sm_60 \
    -gencode arch=compute_70,code=\'compute_70,sm_70\'
```

If the GPU has a higher compute capability than 7.0, then the graphics driver compiles the 7.0-optimised PTX code to binary code that is optimised for the compute capability of the GPU. However, the optimised binary code cannot support features that are not present in the 7.0-optimised PTX code. More information can be found from the nvcc documentation [19].

4.2.3. Kernels

The device-side code is written inside special functions called *kernels*. A kernel is declared with the *__global__* keyword and its return type is always *void*. In the previous "Hello world" example, the *main* function is executed on the host and the *say_hello* kernel is executed on a device:

```
__global__ void say_hello()
{
    printf("A device says, Hello world!\n");
}
```

The host *issues* the *say_hello* kernel to the default device (one device is always active for each host thread) as follows:

```
say_hello<<<1,1>>>();
```

This places an instance of the *say_hello* kernel into a command queue called *stream* (so-called null stream in this example since no specific stream is specified). The kernel is not guaranteed to be executed at this point, instead the CUDA runtime system will launch the kernel on the device when it becomes possible. All kernels that are placed into a stream are executed in the order they are issued and two kernels in the same stream cannot execute concurrently. We will return to the <<< ... , ... >>> in the next subsection. For now, it is sufficient to know that the number of threads executing the kernel is set to one.

When the CUDA runtime system finally launches the kernel, it creates a set of threads (one thread in this example) that all enter a kernel from the beginning of the body of the kernel function. That is, the threads are not spawned in a fork-join style inside the kernel (except when a kernel issues a child kernel; see CUDA Dynamic Parallelism API). Finally, the *cudaDeviceSynchronize()* function call causes the host thread to wait until the (null) stream is empty, i.e. the kernel has been executed on the default device. If the *cudaDeviceSynchronize()* function call is left out, then it is possible that the program exits before the CUDA runtime system has launched the kernel. In that case, only the host-side *printf()* function call gets executed.

4.2.4. Threads and thread blocks

As mentioned in the previous subsection, all threads enter a kernel from the beginning of the body of the kernel function. The number of threads executing a kernel is usually very large as a typical GPU can consist of thousands of CUDA cores and each CUDA core can execute several threads concurrently. In the same manner as the CUDA cores are divided into SMs, the threads are divided into *thread blocks*. Each thread block is mapped to one SM while one SM can have multiple thread blocks mapped to it. Threads that belong to the same thread block are active concurrently and are able to communicate with each other. Threads that belong to different thread blocks are not guaranteed to be active concurrently.

All thread blocks together form a *grid* that can be either one-, two-, or three-dimensional; see Figure 20. The dimensions of the grid are given by the built-in kernel variables *gridDim.x*, *gridDim.y* and *gridDim.z*. Each thread block is given an unique *thread block index* that can be accessed through the built-in kernel variables *blockIdx.x*, *blockIdx.y* and *blockIdx.z*.

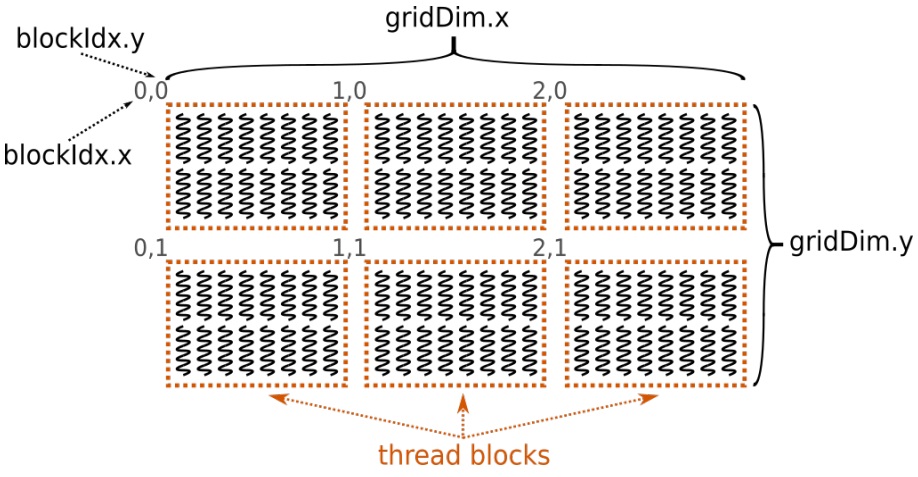

Figure 20. A grid that contains 6 thread blocks, 3 thread block in the x dimension and 2 thread block in the y dimension

Likewise, a thread block can be either one-, two-, or three-dimensional; see Figure 21. The dimensions of the thread block are given by the built-in kernel variables *blockDim.x*, *blockDim.y* and *blockDim.z*. Each thread is given a *thread index* that can be accessed through the built-in kernel variables *threadIdx.x*, *threadIdx.y* and *threadIdx.z*.

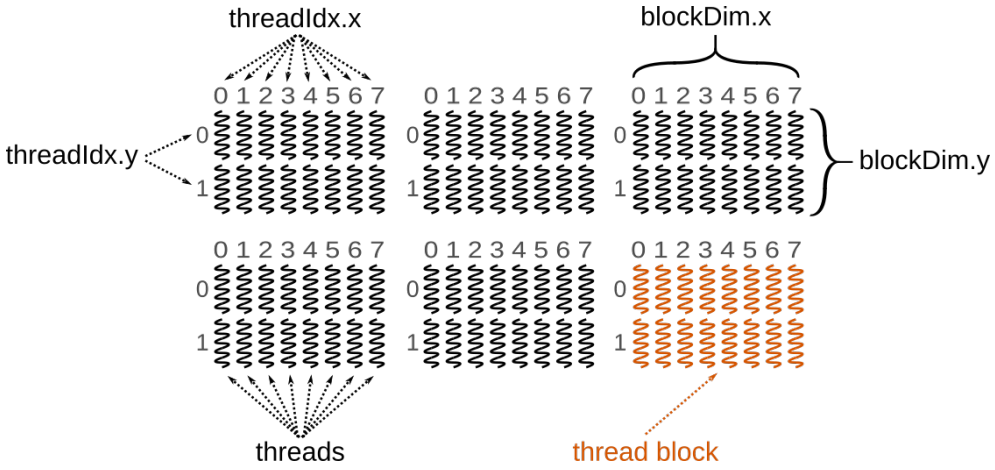

Figure 21. A grid that contains 6 thread blocks with 16 threads in each, 8 threads in the x dimension and 2 threads in the y dimension. Note that the thread blocks are small for illustrational purposes only

The dimensions of the grid and the thread blocks are given when a kernel is issued:

```
kernel_name<<<blocks, threads, smem, stream>>>(...);
```

Above, the argument *blocks* is of type *dim3* and specifies the dimensions of the grid (*blocks.x* × *blocks.y* × *blocks.z* thread blocks). The argument *threads* is of type *dim3* and specifies the dimensions of the thread blocks (*threads.x* × *threads.y* × *threads.z* threads per thread block). The argument *smem* is the number of bytes of shared memory allocated (see the next subsection) and the argument *stream* is the stream to which the kernel is placed. The following example launches a kernel with 96 threads:

```
#include <stdio.h>

__global__ void say_hello()
{
    printf("Thread (%d,%d) in block (%d,%d) says, Hello world!\n",
        threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y);
}

int main()
{
    dim3 threads(8, 2);
    dim3 blocks(3, 2);
    say_hello<<<blocks, threads>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

As illustrated in Figure 21, the threads are organised into 6 thread blocks, 3 thread blocks in the *x* dimension and 2 thread blocks in the *y* dimension. Each thread block contains 16 threads, 8 threads in the *x* dimension and 2 threads in the *y* dimension. Note that the thread blocks should be significantly larger in practice. In particular, it is generally recommended that the thread block's *x* dimension is a multiple of 32. The program outputs:

```
Thread (0,0) in block (2,1) says, Hello world!
Thread (1,0) in block (2,1) says, Hello world!
Thread (2,0) in block (2,1) says, Hello world!
...
```

```
Thread (5,1) in block (0,0) says, Hello world!
Thread (6,1) in block (0,0) says, Hello world!
Thread (7,1) in block (0,0) says, Hello world!
```

Threads that belong to the same thread block can synchronise their execution with the *__syncthreads()* barrier function. Threads that belong to different thread blocks are synchronised only when a kernel launch begins and ends.

4.2.5. Memory spaces and data transfers

A CUDA program has six primary memory spaces:

- *Host memory* is accessible by the host and is located in the main memory of the computer (RAM). If a host memory buffer is allocated as *pinned memory*, then a device can access the buffer through a PCIe bus. The host manages the host memory. The bandwidth of the host memory is usually relatively narrow, especially when it is accessed through a PCIe bus.
- *Global memory* is accessible by all threads in all thread blocks and is located in the internal memory of a device (VRAM). The host manages the global memory. The bandwidth of the global memory is relatively wide but the amount of global memory that is available on a device is usually measured in tens of gigabytes.
- *Shared memory* is accessible by threads that belong to the same thread block and is located in the SM the thread block is mapped to. A shared memory buffer is automatically allocated for the livetime of the thread block. The shared memory allocation size is either static or dynamic (specified when a kernel is issued). The bandwidth of the shared memory is extremely wide but the amount of shared memory that is available is usually measured in hundreds of kilobytes per SM.
- *Constant memory* is readable by all threads in all thread blocks and is located in the internal memory of a device behind a read-only cache. The host manages the constant memory. Contemporary NVIDIA GPUs contain 64KB of constant memory.
- *Texture memory* is readable by all threads in all thread blocks and is located in the internal memory of a device behind a read-only cache. Texture memory offers additional addressing modes and data filtering for some specific data formats (see [15]). The host manages the texture memory. Texture memory is a part of the global memory.
- *Local memory* is accessible by a single thread and is located in the internal memory of a device. Local memory is a part of the global memory.

The CUDA memory model is *relaxed*. Threads that belong to the same thread block can use various memory fence commands or the *__syncthreads()* barrier function to order their memory transfers. Threads that belong to different thread blocks cannot control the order of the memory transfers. CUDA provides a set of atomic memory access functions but these functions only guarantee that the operation itself is atomic. All memory transfers are completed before a kernel launch ends.

The host allocates both the host memory and the global memory. The host also initialises the necessary data transfers between the host memory and the global memory. The data transfers can be blocking, in which case the host waits until the null stream is empty before starting the transfer, or asynchronous, in which case the data transfer is placed into a stream. Alternatively, the host can allocate a buffer of *managed memory* that is transparently accessible from both the host and a device. It should be noted that the device should be of the Kepler architecture or newer (preferably Pascal or newer). The necessary data transfers are triggered automatically when either the host or the device accesses a memory page that does not currently reside in the host memory or the global memory, respectively.

4.2.6. Examples

Let us consider the operation of computing a matrix-vector multiplication $y = Ax$. Since each row of the product vector y can be computed independently, we can parallelise the operation in a trivial manner by mapping one thread to each row of the vector. The corresponding kernel looks like this:

```
// a kernel that performs a matrix-vector multiplication y = A * x, where
// the matrix A has m rows and n columns
__global__ void gemv_kernel(
    int m, int n, int ldA, double const *A, double const *x, double *y)
{
    // we are assuming that each row of the vector y gets its own thread
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    // threads that do not contribute to the final result skip to the
    // end of the kernel
    if (thread_id < m) {

        //
        // loop over the corresponding row of the matrix A and the
        // vector x
        //
        // |y_0|   |A_00 A_01 A_02 .... |   |x_0|
        // |y_1|   |A_10 A_11 A_12 .... |   |x_1|
        // |y_2| = |A_20 A_21 A_22 .... | * |x_2|
        // |...|   |.... .... .... .... |   |...|
        // |...|   |.... .... .... .... |   |...|
        //
        // y_k = A_k0 * x_0 + A_k1 * x_1 + A_k2 * x_2 + ...
        //
        double y_k = 0.0;
        for (int i = 0; i < n; i++)
            y_k += A[i*ldA+thread_id] * x[i];

        // store the result to the vector y
        y[thread_id] = y_k;
    }
}
```

That is, each thread calculates a unique global thread index *thread_id* by using the thread block index, the thread block size and the thread index. We are assuming that the number of threads entering the kernel is larger than or equal to the number of rows in the product vector *y*. This means that the kernel does not need to loop over the rows of the matrix *A*. However, the excess threads (each thread block contains the same number of threads) must skip all computations and continue directly to the end of the kernel. The threads that are meant to be active then loop over the corresponding row of the matrix, accumulate the dot product to an intermediate variable *y_k*, and store the result *y_k* to the product vector *y*. Note that threads that have adjacent global thread indices access adjacent 64-bit words. This makes the memory access pattern coalescing (see Section 3.2.1, “GPUs”) as threads that belong to the same warp access neighbouring 64-bit words.

The matching host-side code looks like this:

```
#include <stdlib.h>

__global__ void gemv_kernel(
    int m, int n, int ldA, double const *A, double const *x, double *y);

int main(int argc, char **argv)
{
    int m = 1000, n = 1000, ldA = 1008;
    double *A, *y, *x;          // host memory pointers
    double *d_A, *d_y, *d_x;    // global memory pointers
    srand(time(NULL));
```

```
    // allocate host memory for the matrix A and the vectors y and x

    A = (double *) malloc(n*ldA*sizeof(double));
    y = (double *) malloc(m*sizeof(double));
    x = (double *) malloc(n*sizeof(double));

    // initialise host memory

    for (int i = 0; i < n; i++) {
        x[i] = 2.0*rand()/RAND_MAX - 1.0;
        for (int j = 0; j < m; j++)
            A[i*ldA+j] = 2.0*rand()/RAND_MAX - 1.0;
    }

    // allocate global memory

    cudaMalloc(&d_A, n*ldA*sizeof(double));
    cudaMalloc(&d_y, m*sizeof(double));
    cudaMalloc(&d_x, n*sizeof(double));

    // copy the matrix A and the vector x from the host memory to the
    // global memory

    cudaMemcpy(d_A, A, n*ldA*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_x, x, n*sizeof(double), cudaMemcpyHostToDevice);

    // launch the kernel

    dim3 threads = 128;
    dim3 blocks = (m+threads.x-1)/threads.x;
    gemv_kernel<<<blocks, threads>>>(m, n, ldA, d_A, d_x, d_y);

    // copy the vector y from the global memory to the host memory

    cudaMemcpy(y, d_y, m*sizeof(double), cudaMemcpyDeviceToHost);

    // free the allocated memory

    free(A); free(y); free(x);
    cudaFree(d_A); cudaFree(d_y); cudaFree(d_x);

    return EXIT_SUCCESS;
}
```

The host allocates the required memory from both the host memory (*malloc*) and the global memory (*cudaMalloc*). Note that the *cudaMalloc* function returns a memory buffer that is optimally aligned for both scalar and vector data types. In order to preserve this alignment, the leading dimensions of the matrix (*ldA*) should be a multiple of the cache line width (in bytes). In the case of the contemporary NVIDIA GPUs, the level-1 cache line is 128 bytes and the level-2 cache line is 32-bytes. The leading dimension 1008 therefore aligns the memory perfectly for the level-1 cache ($1008 \times 8 = 63 \times 128$). The matrix *A* and the right-hand side vector *x* are then copied from the host memory to the global memory, and the kernel *gemv_kernel* is issued. The thread block size is set to 128 and the number of thread blocks is calculated such that the total number of threads is larger than or equal to the number of rows in the product vector *y*. Finally, the product vector *y* is copied from the global memory to the host memory. Since the kernel is issued to the null stream and the data transfers are blocked (and synchronised with the null stream), the data transfers and the kernel launch are guaranteed not to overlap.

In the above example, the number of active threads is limited by the height of the matrix *A*. If the matrix is not large enough, it is possible that the kernel launch does not generate enough threads to saturate the CUDA cores. Since the matrix-vector multiplication is memory bound (see Section 3.1, “Roofline Model”), this low saturation will manifest itself as reduced memory throughput and thus lower performance. The solution to this problem is to introduce more threads per row of the matrix as shown below:

```
// fix thread block dimensions so that blockDim.x = blockDim.y = warp size
#define THREAD_BLOCK_SIZE 32

// a kernel that performs a matrix-vector multiplication y = A * x, where
// the matrix A has m rows and n columns
__global__ void gemv_kernel(
    int m, int n, int ldA, double const *A, double const *x, double *y)
{
    // two-dimensional shared memory array for storing the partial dot
    // products
    __shared__ double tmp[THREAD_BLOCK_SIZE][THREAD_BLOCK_SIZE];

    // we are using grid's and threads blocks's x dimensions for mapping
    // the threads to the rows of the matrix
    int thread_id = blockIdx.x * THREAD_BLOCK_SIZE + threadIdx.x;

    double y_k = 0.0;
    if (thread_id < m) {

        // Loop over the corresponding row of the matrix A and the
        // vector x. Note that each thread computes only a part of the
        // dot product.
        for (int i = threadIdx.y; i < n; i += THREAD_BLOCK_SIZE)
            y_k += A[i*ldA+thread_id] * x[i];
    }

    // each thread stores its partial dot product to the shared memory
    // array
    tmp[threadIdx.x][threadIdx.y] = y_k;

    // each thread waits until all threads have done the same
    __syncthreads();

    // threads that share a common index in the y dimension sum together
    // the partial dot products (note the swapped x-y dimensions)
    int active = THREAD_BLOCK_SIZE/2;
    while (0 < active) {
        if (threadIdx.x < active)
            tmp[threadIdx.y][threadIdx.x] +=
                tmp[threadIdx.y][threadIdx.x + active];
        active /= 2;

        __syncthreads();
    }

    // one thread in the y dimension stores the final dot product
    if (thread_id < m && threadIdx.y == 0)
        y[thread_id] = tmp[threadIdx.x][0];
}
```

The above example introduces two primary changes:

- Each row of the matrix *A* has *THREAD_BLOCK_SIZE* threads mapped to it. Specifically, the thread block's x dimension is used for calculating the corresponding row of the product vector (*thread_id*) and the thread block's y dimension is used for looping over the corresponding row of the matrix (*for* loop). In this example, this means that the number of active threads is increased by a factor of 32 when compared to the first example.

- After each thread has finished computing its own part of the dot product, it stores the partial result to a shared memory array *tmp* and waits until all other threads in the thread block have done the same (*__syncthreads()*). After this, threads that share a common index in the y dimension perform a pairwise tree-like summation to obtain the final dot product. At the beginning, half of the threads are active as indicated by the *active* variable. Each thread that is active during an iteration of the *while* loop sums two partial results together and waits until all other threads in the thread block are ready. At his point, the number of active threads is also halved. Finally, one thread in the y dimension stores the final dot product to the product vector *y*.

Note that the above example is not meant to be the final completely-optimised version of the kernel. It is simply meant to illustrate how multidimensional thread blocks and shared memory can be used to improve a code. More information relating to tuning and optimisation of CUDA programs can be found from Section 5.2.

On the host side, we must update the thread block dimensions:

```
dim3 threads(THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE);
dim3 blocks((m+THREAD_BLOCK_SIZE-1)/THREAD_BLOCK_SIZE, 1);
gemv_kernel<<<blocks, threads>>>(m, n, ld_dA, d_A, d_x, d_y);
```

Figure 22 demonstrates the performance difference between the two example codes. Note that the difference is largest when the matrix is relatively small. This agrees with the earlier conclusions.

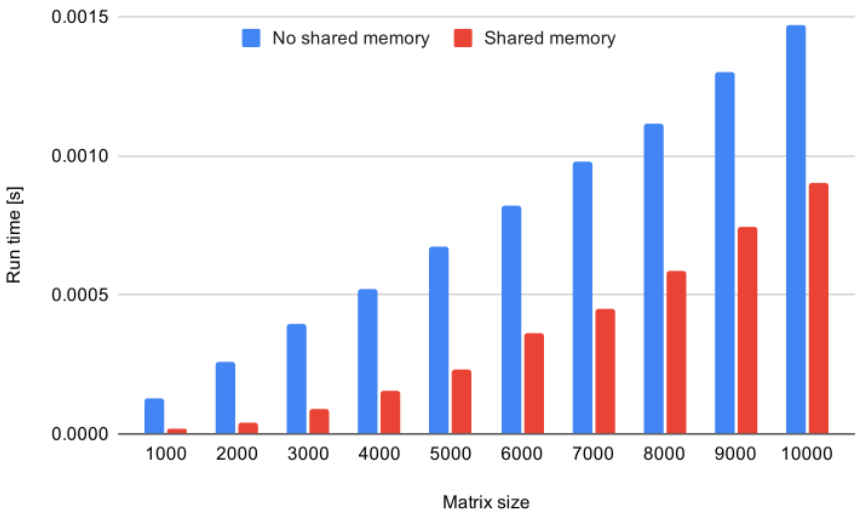

Figure 22. A comparison between the two example codes on a NVIDIA Tesla V100 GPU

4.3. OpenCL

Open Computing Language (OpenCL) [20] is an open standard and framework for developing parallel software for heterogeneous platforms that consist of CPUs, GPUs, Digital Signal Processors (DSPs), FPGAs, vector processors and other hardware accelerators. As the CUDA and OpenCL programming models are very similar, this section includes a terminology translation table.

4.3.1. Compute units and processing elements

OpenCL divides the computer hardware into a *host* and one or more OpenCL *devices*. In most cases, the main CPU of the computer acts as the host. An OpenCL device can be either a CPU, a GPU, a DSP, an FPGA, a vector processor or some other hardware accelerator. The devices are intended to perform the bulk of the computations whereas the host is responsible for managing the devices and submitting commands to them. A device consists of one or more *computing units* which are further divided into one or more *processing elements*. Computations on a device occur within the processing elements. The processing elements within a computing unit can either act together as a SIMD unit or separately as Single Program Multiple Data (SPMD) units.

4.3.2. Host and kernels

An OpenCL program is divided into a *host program* that executes on the host and *kernels* that execute on one or more devices. The kernels are written in *OpenCL C* which is an extended dialect of C99. A kernel is declared with the *__kernel* keyword and its return type is always *void*:

```
__kernel void say_hello()
{
    printf("Device says, Hello world!\n");
}
```

The kernels are stored either as string(s) that are compiled during the execution or as pre-compiled binaries that are loaded during the execution. The computations are defined as "work" that occurs through *work-items* that execute in *work groups*. That is, the OpenCL programming model refers to threads as work-items.

The host places a kernel call into a *command-queue* that is associated with a device. The kernel is not guaranteed to be executed when it is placed into the command-queue, instead the OpenCL runtime system will launch the kernel on the associated device when it becomes possible. All kernels that are placed into a command-queue are executed in the order they are enqueued and two kernels in the same command-queue cannot execute concurrently. When the OpenCL runtime system finally launches the kernel, it creates a set of work-items that all enter a kernel from the beginning of the body of the kernel function. That is, the work-items are not spawned in a fork-join style inside the kernel (except when a kernel enqueues a child kernel; see OpenCL 2.0 device-side enqueue).

4.3.3. Work-items and work groups

All work-items from an *index space* called NDRange that can be either one-, two-, or three-dimensional. The dimensionality index space is given by the built-in kernel function *get_work_dim()*. The size of the index space in a specific dimension is given by the built-in kernel function *get_global_size(uint dimindx)*. Each work-item is given a unique *global index* in each dimension that can be accessed through the built-in kernel function *get_global_id(uint dimindx)*.

A work group can be either one-, two-, or three-dimensional; see Figure 23. The number of work groups in a specific dimension is given by the built-in kernel function *get_num_groups(uint dimindx)*. Each work group is given a unique *group index* in each dimension that can be accessed through the built-in kernel function *get_group_id(uint dimindx)*.

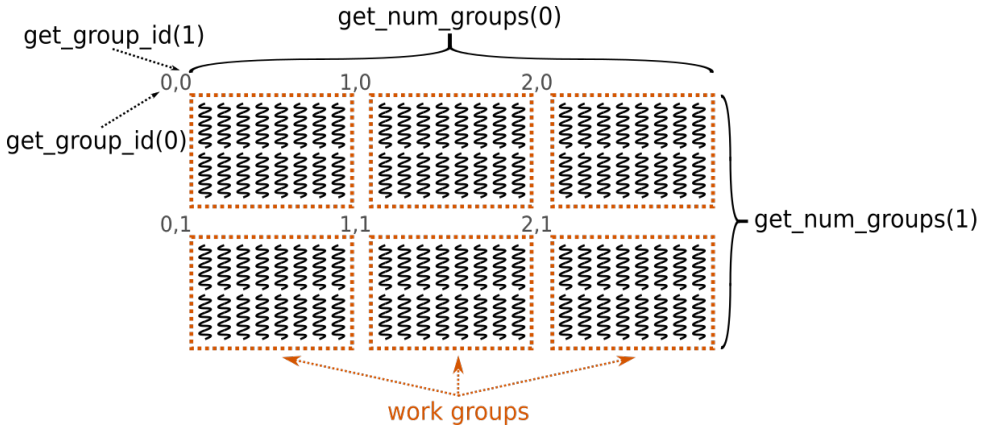

Figure 23. An index space that contains 6 work groups, 3 work groups in the first dimension and 2 work groups in the second dimension

The size of the work group in a specific dimension is given by the built-in kernel function *get_local_size(uint dimindx)*; see Figure 24. Each work-item is given a *local index* in each dimension that can be accessed through the built-in kernel function *get_local_id(uint dimindx)*.

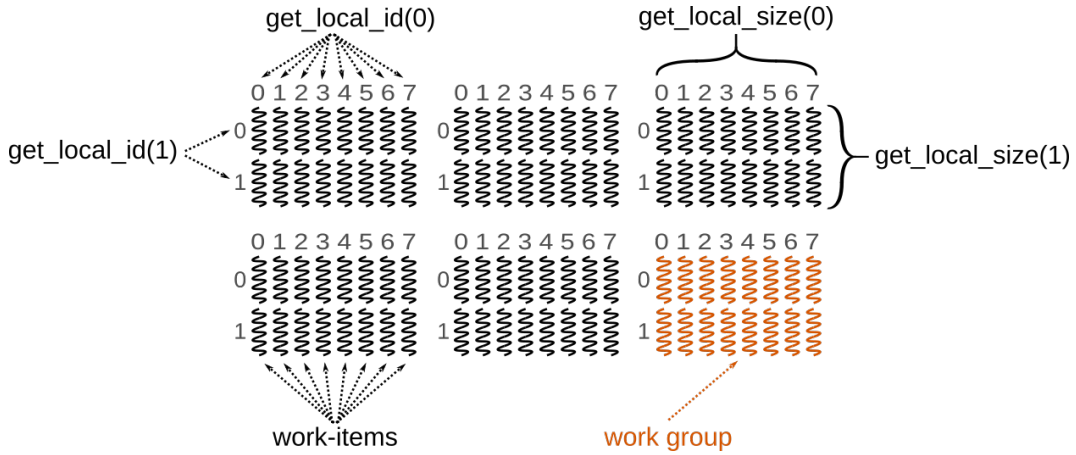

Figure 24. An index space that contains 6 work groups with 16 work-items in each, 8 work-items in the first dimension and 2 work-items in the second dimension. Note that the work groups are small for illustrational purposes only

The dimensions of the index space and the work groups are given when a kernel is enqueued:

```
kernel_name(cl::EnqueueArgs(queue, global, local), ...);
```

Above, the argument *global* is of type *cl::NDRange* and specifies the dimensions of the index space (*global(0)* × *global(1)* × *global(2)* work-items). The argument *local* is of type *cl::NDRange* and specifies the dimensions of the work groups (*local(0)* × *local(1)* × *local(2)* work-items per work group). The argument *queue* is the command-queue to which the kernel is placed. The following example enqueues a kernel with 96 work-items:

```
const std::string kernelSource {
R"CLC(
__kernel void say_hello()
{
    printf("Work-item (%d,%d) in work group (%d,%d) says, Hello world!\n",
        get_local_id(0), get_local_id(1), get_group_id(0), get_group_id(1));
}
)CLC" };
```

```
int main()
{
    ...

    cl::NDRange global(8 * 3, 2 * 2);   // index space dimensions
    cl::NDRange local(8, 2);            // work group dimensions
    say_hello(cl::EnqueueArgs(queue, global, local));

    ...
}
```

As illustrated in Figure 24, the work-items are organised into 6 work groups, 3 work groups in the first dimension and 2 work groups in the second dimension. Each work group contains 16 work-items, 8 work-items in the first dimension and 2 work-items in the second dimension. Note that the work groups should be significantly larger in practise. The program outputs:

```
Work-item (0,0) in work group (2,1) says, Hello world!
Work-item (1,0) in work group (2,1) says, Hello world!
Work-item (2,0) in work group (2,1) says, Hello world!
...
Work-item (5,1) in work group (0,0) says, Hello world!
Work-item (6,1) in work group (0,0) says, Hello world!
Work-item (7,1) in work group (0,0) says, Hello world!
```

Work-items that belong to the same work group can synchronise their execution with the *work_group_barrier()* barrier function. Work-items that belong to different work groups are synchronised only when a kernel launch begins and ends.

4.3.4. Memory spaces and data transfers

An OpenCL program has five primary memory spaces:

- *Host memory* is accessible by the host thread and is located in the main memory of the computer. The host manages the host memory.
- *Global memory* is accessible by all work-items in all work groups and is usually located on a device. The host manages the global memory.
- *Local memory* is accessible by work-items that belong to the same work group and is usually located on a computing unit.
- *Constant memory* is readable by all work-items in all work groups and is usually located on a device behind a read-only cache. The host manages the constant memory.
- *Private memory* is accessible by a single work-item and is usually located on a device.

The OpenCL memory model is *relaxed*. Work-items that belong to the same work group can use various memory fence commands or the *work_group_barrier()* barrier function to order their memory transfers. Work-items that belong to different work groups cannot control the order of the memory transfers. OpenCL provides a set of atomic memory access functions but these functions only guarantee that the operation itself is atomic. All memory transfers are completed before a kernel launch ends.

The host allocates both the host memory and the global memory. The host also initialises the necessary data transfers between the host memory and the global memory. The data transfers can be blocking, in which case the host waits until the command-queue is empty before starting the transfer, or asynchronous, in which case the data transfer is placed into the command-queue. Alternatively, the host can allocate a buffer of *shared virtual memory* that is transparently accessible from both the host and a device.

4.3.5. Host-side API

Since OpenCL supports a wide variety of hardware, the OpenCL's host-side API is much more involved than CUDA's equivalent (see Section 4.2). This section briefly summarises the most relevant host-side OpenCL objects. A corresponding class diagram is presented in Figure 25. The host-side code can be written either in C or C++ (wrappers exists for several other languages). For the latter case, OpenCL provides a C++ wrapper header file that defines corresponding C++ classes.

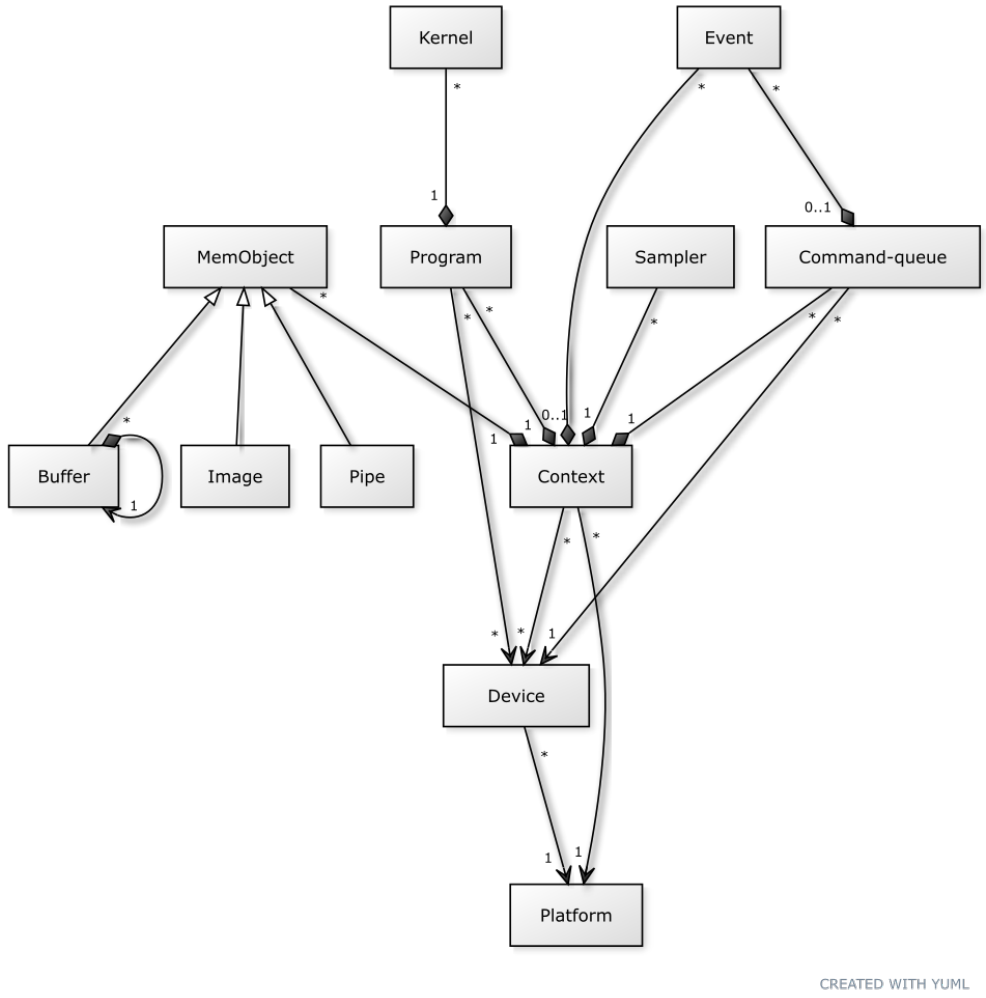

Figure 25. OpenCL 2.0 class diagram using the UML notation

Platform encapsulates the host and any devices connected to it. Typically, each device vendor provides their own platform.

Device encapsulates a CPU, a GPU, a DSP, a FPGA, vector processor, or other hardware accelerator device. A device can execute a kernel from a program that is compiled for the device.

Context encapsulates the entire environment in which all other objects, such kernels, memory objects, command-queues and events, are defined for a specific set of devices.

Program encapsulates a batch of device-side code that consists of one or more kernels. A program can be compiled for a given set of devices.

Kernel encapsulates a function that can be placed on a command-queue at the host side and later executed on a device.

MemObject is an abstract class for a memory object. OpenCL supports the following memory objects:

- *Buffer* encapsulates a linear collection of bytes. A buffer can be a sub-buffer inside another buffer. On the device side, a buffer appears as a pointer argument for a kernel.
- *Image* encapsulates a two- or three- dimensional structured array.
- *Pipe* describes an ordered sequence of data items. A pipe can be used to pass linear data between two kernels.

Command-queue stores a sequence of commands (kernels, data transfers, etc) that will be executed on a specific device. The commands may be executed in-order (default) or out-of-order.

Event encapsulates the status of an operation (in a command-queue). Events can be used to track and synchronise operations in a context.

Sampler describes how an image object is sampled when it is accessed in a kernel.

4.3.6. Terminology comparison

Table 7 offers a terminology translation between the OpenCL and CUDA programming models.

Table 7. OpenCL and CUDA terminology translation

OpenCL	CUDA	Comment
Host	Host	
Device	Device	
Computing unit	SM	
Processing element	CUDA core	
Kernel	Kernel	*__kernel* in OpenCL; *__global__* in CUDA
Command-queue	Stream	
Index space (NDRange)	Grid	*get_num_groups()* in OpenCL; *gridDim* in CUDA
Work-item	Thread	*get_local_id()* in OpenCL; *threadIdx* in CUDA
Work group	Thread block	*get_group_id()*, *get_local_size()* in OpenCL; *blockIdx*, *blockDim* in CUDA
Host memory	Host memory	*__host__* in CUDA
Global memory	Global memory	*__global* in OpenCL, *__device__* in CUDA
Local memory	Shared memory	*__local* in OpenCL, *__shared__* in CUDA
Constant memory	Constant memory	*__constant* in OpenCL, *__constant__* in CUDA
Private memory	Local memory	*__private* in OpenCL, *__local__* in CUDA

4.3.7. Examples

Let us again consider the example of computing a matrix-vector multiplication $y = Ax$. We are assuming that the target device is a GPU. Since each row of the product vector y can be computed independently, we can parallelise the operation in a trivial manner by mapping one work-item to each row of the vector. The kernel source code is stored into a C++ standard library string and compiled during runtime:

```
const std::string kernelSource {
R"CLC(
// a kernel that perform a matrix-vector multiplication y = A * x, where
// the matrix A has m rows and n columns
__kernel void gemv_kernel(
    int m, int n, int ldA, __global double const *A,
    __global double const *x, __global double *y)
{
```

```
    // we are assuming that each row of the vector y gets its own
    // work-item
    int global_id = get_global_id(0);

    if (global_id < m) {

        //
        // loop over the corresponding row of the matrix A and the
        // vector x
        //
        // |y_0|   |A_00 A_01 A_02 .... |   |x_0|
        // |y_1|   |A_10 A_11 A_12 .... |   |x_1|
        // |y_2| = |A_20 A_21 A_22 .... | * |x_2|
        // |...|   |.... .... .... .... |   |...|
        // |...|   |.... .... .... .... |   |...|
        //
        // y_k = A_k0 * x_0 + A_k1 * x_1 + A_k2 * x_2 ...
        //
        double y_k = 0.0;
        for (int i = 0; i < n; i++)
            y_k += A[i*ldA+global_id] * x[i];

        // store the result to the vector y
        y[global_id] = y_k;
    }
}
)CLC" };
```

Each work-item first acquires its global index number by calling the built-in *get_global_id()* kernel function. We are assuming that the number of work-items entering the kernel is larger than or equal to the number of rows in the product vector *y*. This means that the kernel does not need to loop over the rows of the matrix *A*. However, the excess work-items (each work group contains the same number of work-items) must skip all computations and continue directly to the end of the kernel. The work-items that are meant to be active loop over the corresponding row of the matrix, accumulate the dot product to an intermediate variable *y_k*, and store the result *y_k* to the product vector *y*. Note that work-items that have adjacent global indices access adjacent 64-bit words. This makes the memory access pattern coalescing (see Section 3.2.1, “GPUs”) as work-items that belong to the same warp or wavefront access neighbouring 64-bit words.

The matching host-side code looks like this:

```
// target OpenCL 2.0
#define CL_HPP_TARGET_OPENCL_VERSION 200

// enable OpenCL C++ exceptions
#define CL_HPP_ENABLE_EXCEPTIONS

#include <cstdlib>
#include <CL/cl2.hpp>

// the kernel source code is stored into a string
const std::string kernelSource;

int main()
{
    int m = 1000, n = 1000, ldA = 1008;
    double *A, *y, *x;              // host pointers
    cl::Buffer d_A, d_y, d_x;       // device buffers
```

```
srand(time(NULL));

// allocate host memory for the matrix A and the vectors y and x

A = (double *) malloc(n*ldA*sizeof(double));
y = (double *) malloc(m*sizeof(double));
x = (double *) malloc(n*sizeof(double));

// initialise host memory

for (int i = 0; i < n; i++) {
    x[i] = 2.0*rand()/RAND_MAX - 1.0;
    for (int j = 0; j < m; j++)
        A[i*ldA+j] = 2.0*rand()/RAND_MAX - 1.0;
}

// create an OpenCL context for GPU devices

cl::Context context = cl::Context(CL_DEVICE_TYPE_GPU);

// create an OpenCL program object and compile the kernel

cl::Program program = cl::Program(context, kernelSource, false);
program.build("-cl-std=CL2.0");

// create an OpenCL kernel object

cl::KernelFunctor<int, int, int, cl::Buffer, cl::Buffer, cl::Buffer>
    kernel(program, "gemv_kernel");

// allocate global memory

d_A = cl::Buffer(context, CL_MEM_READ_ONLY,  n*ldA*sizeof(double));
d_y = cl::Buffer(context, CL_MEM_READ_WRITE, m*sizeof(double));
d_x = cl::Buffer(context, CL_MEM_READ_ONLY,  n*sizeof(double));

// create an OpenCL command-queue for the first device in the context

cl::CommandQueue queue = cl::CommandQueue(context);

// copy the matrix A and the vector x from the host memory to the
// global memory

queue.enqueueWriteBuffer(d_A, false, 0, n*ldA*sizeof(double), A);
queue.enqueueWriteBuffer(d_x, false, 0, n*sizeof(double), x);

// enqueue the kernel

kernel(
    cl::EnqueueArgs(queue, cl::NDRange(m)), m, n, ldA, d_A, d_x, d_y);

// copy the vector y from the global memory to the host memory

queue.enqueueReadBuffer(d_y, true, 0, m*sizeof(double), y);

// free the allocated memory

free(A); free(y); free(x);
```

```
    return EXIT_SUCCESS;
}
```

First, we specify that we are targeting OpenCL version 2.0, enable OpenCL C++ exceptions, and include the OpenCL 2.0 C++ header file. In the *main()* function, the host thread first allocates (*malloc*) and initialises the required host memory. This example code skips the explicit creation of the *cl::Platform* and *cl::Device* objects and instead creates the *cl::Context* directly by calling a constructor function that allows us to directly specify the desired device type. In this case, we create a context that is associated with one or more GPUs (*CL_DEVICE_TYPE_GPU*). A *cl::Program* object is then constructed from the kernel source code and kernel source is compiled for all devices associated with the context. We are using C++ functors to simplify the code and create a *cl::KernelFunctor* object for the kernel *gemv_kernel*.

The global memory is allocated by creating three *cl::Buffer* objects: one buffer for the matrix and one buffer for each vector. Only the buffer *d_y* is allocated as read-write memory. Note that the OpenCL implementations generally return memory buffers that are optimally aligned for both scalar and vector data types. In order to preserve this alignment, the leading dimensions of the matrix (*ldA*) should be a multiple of the cache line width (in bytes). In case of the contemporary NVIDIA GPUs, the level-1 cache line is 128 bytes and the level-2 cache line is 32-bytes. The leading dimension 1008 therefore aligns the memory perfectly for the level-1 cache ($1008 \times 8 = 63 \times 128$). A *cl::CommandQueue* object is created for the first device that is associated with the context, and the data transfers and the kernel are enqueued. Note that the last data transfer (from the global memory to the host memory) is blocking (the second argument is *true*), i.e. the host threads return only after the data transfer has been completed. Also, we are letting the OpenCL runtime system decide the number of work-items created and the dimensions of the work groups (*cl::EnqueueArgs(queue, cl::NDRange(m))*). The number of work-items is guaranteed to be at least *m* (the number of rows in the product vector *y*).

In the above example, the number of active work-items is limited by the height of the matrix *A*. If the matrix is not large enough, it is possible that the kernel launch does not generate enough work-items to saturate the processing elements. Since the matrix-vector multiplication is memory bound on contemporary GPUs (see Section 3.1, “Roofline Model”), this low saturation will manifest itself as reduced memory throughput and thus lower performance. The solution to this problem is to introduce more work-items per row of the matrix as shown below:

```
const std::string kernelSource {
R"CLC(
// fix work group dimensions so that get_local_size(0) = 32 and
// get_local_size(1) = 32
#define WORK_GROUP_SIZE 32

// a kernel that perform a matrix-vector multiplication y = A * x, where
// the matrix A has m rows and n columns
__kernel void gemv_kernel(
    int m, int n, int ldA, __global double const *A,
    __global double const *x, __global double *y)
{
    // two-dimensional local memory array for storing the partial dot
    // products
    __local double tmp[WORK_GROUP_SIZE][WORK_GROUP_SIZE];

    // we are using index space's first dimensions for mapping the
    // work to the rows of the matrix
    int global_id = get_global_id(0);

    double y_k = 0.0;
    if (global_id < m) {

        // Loop over the corresponding row of the matrix A and the
        // vector x. Note that each work-item computes only a part of the
```

```
        // dot product.
        for (int i = get_local_id(1); i < n; i += WORK_GROUP_SIZE)
            y_k += A[i*ldA+global_id] * x[i];
    }

    // each thread stores its partial dot product to the local memory
    // array
    tmp[get_local_id(0)][get_local_id(1)] = y_k;

    // each-work item waits until all each-works have done the same
    work_group_barrier(CLK_LOCAL_MEM_FENCE);

    // work-items that share a common index in the second dimension sum
    // together the partial dot products (note the swapped dimensions)
    int active = WORK_GROUP_SIZE/2;
    while (0 < active) {
        if (get_local_id(0) < active)
            tmp[get_local_id(1)][get_local_id(0)] +=
                tmp[get_local_id(1)][get_local_id(0) + active];
        active /= 2;

        work_group_barrier(CLK_LOCAL_MEM_FENCE);
    }

    // one work-item in the second dimension stores the final dot product
    if (global_id < m && get_local_id(1) == 0)
        y[global_id] = tmp[get_local_id(0)][0];
}
)CLC" };
```

As before, the above example introduces two primary changes:

- Each row of the matrix *A* has *WORK_GROUP_SIZE* work-items mapped to it. Specifically, index space's first dimension is used for calculating the corresponding row of the product vector (*global_id*) and index space's second dimension is used for looping over the corresponding row of the matrix (*for* loop). In this example, this means that the number of active work-items is increased by a factor of 32 when compared to the first example.

- After each work-item has finished computing its own part of the dot product, it stores the partial result to a local memory array *tmp* and waits until all other work-items in the work group have done the same (*work_group_barrier(CLK_LOCAL_MEM_FENCE)*). After this, the work-item that shares a common index in the second dimension performs a pairwise tree-like summation to obtain the final dot product. At the beginning, half of the work-items are active as indicated by the *active* variable. Each work-item that is active during an iteration of the *while* loop sums two partial results together and waits until all other work-items in the work group are ready. At this point, the number of active threads is also halved. Finally, one work-item in the second dimension stores the final dot product to the product vector *y*.

Note that the above example is not meant to be the final completely-optimised version of the kernel. It is simply meant to illustrate how multidimensional work groups and local memory can be used to improve a code.

On the host side, we must update the work group dimensions:

```
    cl::NDRange global(
        (m/WORK_GROUP_SIZE+1)*WORK_GROUP_SIZE, WORK_GROUP_SIZE);
    cl::NDRange local(WORK_GROUP_SIZE, WORK_GROUP_SIZE);
    kernel(
        cl::EnqueueArgs(queue, global, local), m, n, ldA, d_A, d_x, d_y);
```

4.4. SYCL

Programming model environments for accelerators, such as CUDA, OpenCL and OpenACC, do not take advantage of modern object-oriented programming languages in order to encapsulate their verbose syntaxes. To overcome this issue, the Khronos Group has defined SYCL [21], a cross-platform abstraction C++ programming model for OpenCL, allowing to program OpenCL devices with modern C++ features, such as, inheritance, templates and lambda functions.

DPC++ is the Intel open source compiler for SYCL based applications [22], and results from ISO C++, Khronos SYCL and community extensions. Those extensions aim to increase the performance of SYCL programs. Therefore, DPC++ is not just a compiler for SYCL programs, as it contains features that are not yet integrated in the SYCL standard. This means that, any SYCL program can be compiled with DPC++ but a DPC++ program might not compile with other C++ compiler with SYCL support.

OneAPI [24] is Intel's framework and an open and unified programming model, to develop programs for heterogeneous systems, namely, CPUs, GPUs, FPGAs and other devices. The platform, execution and memory models are based upon the SYCL platform, execution and memory models. OneAPI framework aggregates several tools, namely, the DPC++ compiler, libraries, debuggers and other additional tools.

As these tools are based on the SYCL specification, in this section we illustrate how to develop a program using the SYCL 1.2.1 [21] standard. An introduction to DPC++ can be found in [23] and the framework OneAPI in [24].

4.4.1. Overview

SYCL builds on the underlying concepts of portability and efficiency of OpenCL, enhancing ease of use, and the flexibility of single-source C++. Developers are able to write standard C++ code and, at the same time, they have access to the full range of capabilities of OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly with the OpenCL APIs. With SYCL, programmers obtain simplicity, reuse and efficiency by implementing a single-source multiple compile design, which allows GPU code to be inline with host code. In this way, the resulting application combines the results of the compilation of several different compilers, including a standard C++ compiler used by the programmers to generate host binaries.

SYCL adds the data parallel programming model to C++ programs for heterogeneous systems, where a computing node may be composed of multicore CPUs, GPUs, FPGAs, and other problem-oriented devices like ASICs (Application-Specific Integrated Circuits).

There are several compilers available for SYCL programs: a) ComputeCpp [25], that requires the OpenCL runtime to be installed in the system; b) Intel DevCloud [26], where users can prototype heterogeneous solutions on a diversity of Intel hardware without local installation of software; c) DPC++, the open source compiler from Intel [27]; and d) hipSYCL [28] that relies on OpenMP for CPU, CUDA for NVIDIA GPUs and HIP/ROCm for AMD GPUs.

There are two possible ways for compiling a SYCL program. One is to use a single compiler that can interpret SYCL and generate the binaries. The other is to use different compilers to cope with the diversity of devices. For example, hipSYCL [28] uses different compilers for each of the following targets, i.e. CPU, NVIDIA GPUs and AMD GPUs.

4.4.2. Expressing parallelism

Parallelism is expressed in SYCL by data parallel kernels that are classified as basic kernels, OpenCL NDRange model work-group kernels, hierarchical kernels and single instance kernels. In the following examples, *h*, is an handler object created by the runtime when starting a command queue.

Basic kernel

The execution of a kernel is organised in work-groups that contain work-items, as illustrated in Figure 24. With the Basic kernel model, the runtime determines the configuration of work-groups and the work-group size that best fits the problem size. This model is appropriate for embarrassingly parallel operations, i.e. where a given operation

can be applied independently on each data element. It is expressed by the function *parallel_for*, a member of the handler class used to submit a command queue. The following code illustrates the definition of a kernel to add two vectors. In this case, an object of type SYCL *id* class, is passed as parameter, which gives access only to the global id of the work-items.

```
h.parallel_for(range<1>(N), [=](id<1> i)
{
   c[i] = a[i] + b[i];
});
```

The first parameter of the *parallel_for* is the number of work-items to use. In this case we use one work-item per each element in the vector. The second parameter is the kernel itself, provided as a lambda function. The lambda used for *parallel_for* expects an *id* or an *item* object as parameter. As the range defined has one dimension, we can refer to *i* to identify the work-items. The *range* class represents a one-, two- or three-dimensional range, receiving as parameter the size, or the number of work-items, in each dimension. In this case one-dimension, with N work-items.

A two-dimensional work group can be defined, as shown in the following code, to implement a matrix addition operation, where the number of work-items is equal to the number of elements to add (N×M).

```
h.parallel_for(range<2>(N, M), [=](id<2> idx)
{   int i = idx[0];
    int j = idx[1];
    c[i][j] = a[i][j] + b[i][j];
});
```

In the latter example the work-item is identified by *idx* which returns the global work-item position in both dimensions. If we need to query the group id as well as the local work-item id, then we can use the *item* class as shown in the next example.

OpenCL NDRange model

NDRange kernels can be expressed by the *nd_range* class, similarly to NDRange in OpenCL, where work-items belong to work-groups, and it is appropriate for implementing data locality aware solutions for performance improvement. The following example, starts a kernel with 96 work-items as the example given in Section 4.3.3 and illustrated in Figure 24.

```
range<2> global(8*3, 2*2);
range<2> local(8, 2);
h.parallel_for(nd_range<2>(global,local), [=] (nd_item<2> item){
   printf("Work-item (%d,%d) in work group (%d,%d) says, Hello world!\n",
   item.get_local(0), item.get_local(1),
    item.get_group(0), item.get_group(1));
});
```

Single task

SYCL provides a single kernel execution mode that is sequentially executed on an OpenCL device. This can be useful to create a chain of sequential tasks with each of them managing its own data transfers [21]. A single task kernel can be invoked as follows:

```
h.single_task([] () {
```

```
  // kernel code
});
```

Hierarchical parallel kernel

The hierarchical parallel kernel provides the same functionality as the *nd_range* model, but exposed differently. With the *parallel_for_work_group* the number of groups are defined explicitly, with two possibilities for defining the work-group size.

The following example, of hierarchical parallel kernel, launches a kernel with 96 work-items as in the example above:

```
h.parallel_for_work_group(range<2>(3, 2),
    range<2>(8, 2), [=] (group<2> myGroup){
    parallel_for_work_item(myGroup, [=] (item<2> myItem) {
        printf("Work-item (%d,%d) in work group (%d,%d) says,
        Hello world!\n", myItem(0), myItem(1), myGroup(0), myGroup(1));
    });
});
```

Here, the first range refers to the number of groups to create in each dimension, and the second range refers to the size of each work-group in terms of work-items configuration. If the second range parameter is omitted, the size of each group is defined by the runtime.

4.4.3. Data access and storage

The memory model is an abstraction that aims to be adaptable for different types of hosts and devices. In SYCL, data is stored by the classes *Buffer* and *Image*, which handle the ownership and the data. The class *accessor* specify how data is accessed and handles the access to the data. A *Buffer* object is a one-, two- or three-dimensional array of elements. An *Image* object allows to store image data in a straightforward way, where elements are pixels in a format such as RGB. The targets containing *host*, *global*, *constant* and *local*, refer to the same target memories as in OpenCL.

The transfer of data, between host and device, is implicit in the definition of a buffer or image object, and an accessor, and is controlled by the runtime. Therefore, the data is transferred as required so that it is available in the appropriate memory before it is used. However, SYCL also supplies a *memcpy* function to implement explicit data movements. The access mode informs the runtime how our program will access data to provide more information for implicit data transfer optimisations. With this information, the runtime ensures that any data dependencies are resolved by enqueuing any data transfers before or after executing a kernel. For example, the *access::mode::read* tells the runtime that the data must be available on the device before the kernel is executed. The *access::mode::write* informs the runtime that the kernel may change the contents of a buffer and, therefore, a copy may need to be made to the host when the computation ends. If an access mode contains *discard*, only data transfer from device to host is required, as previous contents are not preserved, and a complete copy of the buffer needs to be made to the host at the end of the kernel execution. The *accessor* template class takes five template parameters [21]: a) a typename specifying the data type that is provided; b) an integer specifying the dimensionality of the accessor; c) the access mode; d) the access target; and e) a value specifying if the accessor is a placeholder.

The access targets available are:

- *access::target::global_buffer* for accessing via global memory
- *access::target::constant_buffer* for accessing via constant memory
- *access::target::local* for work-group local memory
- *access::target::image* for access an image

- *access::target::host_buffer* for accessing a buffer in the host
- *access::target::host_image* for accessing an image in the host
- *access::target::image_array* for accessing an array of images on a device

The access modes available are:

- *access::mode::read* for read only
- *access::mode::write* for write only
- *access::mode::read_write* for read and write
- *access::mode::discard_write* for write only being previous content discarded
- *access::mode::discard_read_write* for read and write being previous content discarded
- *access::mode::atomic* for read and write atomic access

The default target is global memory and it can be omitted when defining the accessor. An example of usage is given in the next section.

4.4.4. Examples

This section presents the anatomy of a complete SYCL program, using the matrix-vector product *y=Ax*, as described in Section 4.3.7. The following code shows the implementation using SYCL:

```
#include <sycl.hpp>
using namespace cl::sycl;

const int SIZE = 256;

void gemv(int A[][SIZE], int *x, int *y) {

    // Device buffers
    buffer<int,2> A_buf(A, range<2>(SIZE,SIZE));
    buffer<int> x_buf(x, range<1>(SIZE));
    buffer<int> y_buf(y, range<1>(SIZE));

    // command queue
    queue q;
    q.submit([&](handler &h) {

        // Data accessors
        accessor<int, 2, access::mode::read, access::target::global_buffer>
            A_in(A_buf, h);
        accessor<int, 1, access::mode::read, access::target::global_buffer>
            x_in(x_buf, h);
        accessor<int, 1, access::mode::write, access::target::global_buffer>
            y_res(y_buf, h);

        // Kernel
        h.parallel_for(range<1>(SIZE), [=](id<1> idx) {
            for(int k=0; k < SIZE; k++)
                y_res[idx] += A_in[idx][k] * x_in[k];
        });
```

```
    });
}

int main() {

    int A[SIZE][SIZE], x[SIZE], y[SIZE];

    for (int i = 0; i < SIZE; ++i) {
        for (int j=0; j< SIZE; ++j)
            A[i][j] = i;
        x[i] = i;
        y[i] = 0;
    }

    gemv(A, x, y);

    for (int i = 0; i < SIZE; i++) std::cout << y[i] << std::endl;

    return 0;
}
```

The first line of the program includes the SYCL headers. All SYCL classes and objects are defined in the *cl::sycl* namespace, which is imported on the second line.

The *gemv* function receives as parameters the addresses of the host arrays created in the main function. To transfer/access the data to/on the device, a buffer object is created for each host array that needs to be accessed on the device. Note that the buffers are not associated with a command queue or context, so that they can handle data transparently across multiple devices. Also, read/write information is not configured at the buffer level, instead it is configured at the queue level.

The next line creates a queue to enqueue the kernel. In contrast to OpenCL, there is no need of any additional configuration. In this case the default constructor of the queue class automatically targets the first OpenCL-enabled device available. SYCL also supplies other constructors and related classes to choose a particular device that has a certain property, however, the code would still be simple as shown here.

With the *submit* function, the runtime retrieves a handler object that is then used to associate the data accessors and the kernel to be run on the device. In the next lines, data accessors are created to specify the data required to run the kernel, the access mode and the target memory. The runtime then transfers the data to/from the device as required to guarantee that the data is available where needed. As the *buffer* is the most common type used when creating an accessor, the *buffer* class provides a *get_access* method to construct an accessor from an existing buffer. Therefore, the three lines that create the accessors can be replaced by the following lines, where the global memory is used by default:

```
auto A_in = A_buf.get_access<access::mode::read>(h);
auto x_in = x_buf.get_access<access::mode::read>(h);
auto y_res = y_buf.get_access<access::mode::write>(h);
```

The kernel itself is specified in the *parallel_for* command. As explained above, the parameter is the configuration of the work-items to use, in this case, one work-item per each element of the result vector, which multiplies a row from *A_in* by the vector *x_in*. The second parameter is the kernel definition as a lambda function. Inside the kernel the code is not very different from the OpenCL version, with classes to retrieve the work-item position inside the setup chosen. When the command queue completes, the runtime copies buffer *y_res* to the host *y* array.

When running the code, if the runtime cannot find a GPU device, it tries to find a CPU that can run OpenCL code. If it fails, then the kernel is run in the host using the C++ runtime.

4.5. OpenACC

OpenACC is a user-driven directive-based performance-portable parallel programming model [29] initially developed by Cray, CAPS, NVIDIA and PGI. OpenACC programs can run on CPU as well as GPU architectures.

4.5.1. Overview

The directive-based approach of OpenACC is similar to OpenMP, which until the version 4.0 did not allow for GPU offloading. The portability makes it possible to use the OpenACC specification in multiple hardware architectures. Currently, supported programming languages are C, C++, and Fortran.

One of the main reasons for creating such a directive-based approach is to ease the development process of existing code where only computationally heavy parts should be ported first, but also to write partially parallel code from scratch. Conventional approaches to GPU-targeted computing such as CUDA or OpenCL allow for extensive and flexible low-level control. However, the downside of using those models is writing a rather complicated and specialised code which can be time consuming. Moreover, this manual approach is usually specified for particular devices. This led to searching for some methods which could allow the compiler to automatically accelerate code sections on the GPU. With that the semi-automatic way based on directives came up. Those directives point to specific code parts and control the data management between host and target devices. Such ways of hinting the compilers wraps the abstraction and facilitates the portability between different architectures of CPUs and GPUs.

The main advantage of OpenACC is that generally there is no need to rewrite existing code. Only in some cases, to increase performance, code needs major changes besides adding some pragmas. This feature allows to introduce parallelisation into complex projects, with small costs, in comparison to other programming models. As mentioned previously, portability provided by the compiler is also a significant advantage of OpenACC. The main disadvantage of using OpenACC is that performance mostly depends on the compiler. For this reason, using the newest compiler is very important. Although the performance of the code may result lower than a tunned CUDA/HIP implementation, for NVIDIA/AMD devices, the low implementation cost makes it worthwhile to use in many cases.

4.5.2. Configuration

For the Volta architecture, the most convenient access to OpenACC is provided by The NVIDIA HPC Software Development Kit (SDK). The information provided in this section is based on version 20.7. The comprehensive installation guide is available through the NVIDIA HPC SDK Installation Guide available at [30]. To check the configuration info of the installed SDK, check the *nvaccelinfo* command. Sample output is presented below. It shows the user parameters of all available NVIDIA GPUs.

```
CUDA Driver Version:           10020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module 440.33.01
                               Wed Nov 13 00:00:22 UTC 2019
Device Number:                 0
Device Name:                   Tesla V100-PCIE-16GB
Device Revision Number:        7.0
Global Memory Size:            16945512448
Number of Multiprocessors:     80
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1380 MHz
Execution Timeout:             No
```

```
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             877 MHz
Memory Bus Width:              4096 bits
L2 Cache Size:                 6291456 bytes
Max Threads Per SMP:           2048
Async Engines:                 7
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
  Multi-Device:                Yes
Default Target:                cc70
```

Above listing shows settings which are related to a given architecture (e.g. maximum block / grid dimensions or cache size). The last line, i.e. *Default Target* specifies the default compute capability. It can be overridden by the *-gpu* compiler flag.

The compilation can be made by the *gcc*, *nvcc*, *nvc++*, *nvc* or *nvfortran* for C, C++ and Fortran code, respectively. To enable OpenACC directives, one should add the *-acc* flag. To exclusively target a Volta GPU, one should specify *-gpu=cc70*. Typically, flags *-fast* and *-Minfo* are added. The former is responsible for code optimisation by inlining and the latter enables the compiler feedback about OpenACC code compilation (it can be limited to only acceleration level by *-Minfo=accel*). With that, the typical compilation would be similar to one of three examples listed below.

```
nvc -fast -acc -Minfo=accel program.c

nvc++ -fast -acc -Minfo=accel program.cpp

nvfortran -fast -acc -Minfo=accel program.f90
```

4.5.3. Programming Model

The general syntax for the *parallel* construct is presented below for C/C++ and Fortran, respectively [31].

```
#pragma acc [directive-name] [clause-list]
{
    structured block
}
```

```
!$acc [directive-name] [clause-list]
    structured block
!$acc end [directive]
```

Both *directives* shown above are made from 4 main parts. A prefix which is *#pragma acc* or *!$acc* depending on language. Next, a directive name that should be considered as *construct* or a function provided by OpenACC and *clause-list* which is a list of parameters for that function. The last part of a construct is a *structured block*, which is a code that will be parallelised, e.g. a *for* loop.

Parallel is a fundamental construct that starts parallel execution on the current device, i.e. the device which will execute parallelised regions. Mostly multicore CPU or GPU, which can be chosen directly from code or specified for all parallelised regions as a compilation flag. A block of code can be parallelised through *gang*, *workers* and *vectors*. *Vectors* are abstract to CUDA threads. A *worker* (CUDA warps) can be interpreted as a set of threads. A *gang* (CUDA thread blocks) is a representation of a group of workers. Please refer to Section 4.2, “CUDA” and Section 4.3, “OpenCL” for more details. For the whole parallelised region the number of gangs, workers in a gang and vectors in a worker remains constant. The code within that region and outside of the loop will be executed redundantly by all gangs.

Kernels is a construct that defines a region of the program that will be compiled into a sequence of kernels for execution on the current device. Typically, each loop nest will be a distinct kernel. It is possible to set different configurations for each one, including the number of gangs of workers and vector size.

Both *parallel* and *kernels* constructs have the same clause sets containing a *data* construct. Most important clauses for them are:

- *async [(int-expr)]* – if that clause does not appear, there is an implicit barrier at the end of the parallelised region, and the local execution will not proceed until the region completes the execution.
- *num_gangs (int-expr)* – a number of gangs that will execute the parallel region or execute kernels in the kernels region. If not specified, it will be set automatically. The default value is related to the implementation and it can be lower than the upper limits forced by the target architecture.
- *num_workers (int-expr)* – a number of workers within each gang. If the clause is not specified, a default value will be set (which can be 1) and might not be optimal. It may also be different for each parallel or kernels construct. Default values are based on the limitations of the target architecture.
- *vector_length (int-expr)* – a number of vectors in a worker. If not specified, default values of the target architecture will be set.
- *copy (var-list)* – for each variable *var* in the *var-list* this clause will copy *var* into the device memory at the entry of a given region. At the exit from that region, data will be copied back to the host.
- *copyin (var-list)* – for each variable *var* in the *var-list* this clause will copy *var* into the device memory at entry of a given region and delete it at the exit.
- *copyout (var-list)* – for each variable *var* in the *var-list* this clause will create *var* on the device at entry of a given region and copy it to the host at the exit.

Loop is a construct that applies to a loop that must immediately follow this directive. *Loop* can describe what type of parallelism to use for executing the loop and for declaring private variables and reduction operations. It should be placed inside a block of code specified as *parallel* or *kernels*, because the way it works partially depends on within which of them it is placed. For this construct the most important clauses are:

- *collapse (n)* – used to specify how many tightly nested loops are associated with the *loop* construct. The argument to the *collapse* clause must be a constant positive integer. If no *collapse* clause appears, only the immediately following loop is associated with the *loop* construct.
- *gang [(gang-arg-list)]* – when in a *parallel* construct, or in an orphaned *loop* construct (i.e. loop construct outside of the *parallel*, *kernel* or *serial* construct), the *gang* clause indicates that iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the gangs created by the *parallel* construct. A *loop* with the *gang* clause transitions a computation region from gang-redundant mode (when execution is made redundantly by all gangs) to gang-partitioned mode (when execution is parallelised by gangs). When it is in the *kernels* construct, the *gang* clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the gangs.
- *worker [([num:]int-expr)]* – when in a *parallel* construct, the *worker* clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the multiple workers within a single gang. A *loop* construct with a *worker* clause causes a gang to transition from worker-single mode (execution by only one worker) to worker-partitioned mode (execution parallelised by many workers). When it is in the *kernels* construct, the *worker* clause specifies that the iterations of the associated loop or loops are

to be executed in parallel across the workers within a single gang. The optional argument specifies how many workers per gang to use to execute the iterations of this loop.

- *vector [([length:]int-expr)]* – when in a *parallel* construct, the *vector* clause specifies that the iterations of the associated loop or loops are to be executed in a vector or SIMD mode. A *loop* construct with a *vector* clause causes a worker to transition from vector-single mode (i.e. the default vectors length equals to one) to vector-partitioned mode (i.e. execution is parallelised across the vector if length is greater than one). When it is the *kernels* construct, the *vector* clause specifies that the iterations of the associated loop or loops are to be executed with a vector or SIMD processing. If an argument is specified, the iterations will be processed in vector strips of that length, if no argument is specified, the implementation will choose an appropriate vector length.

- *reduction (operator:var-list)* – specifies a reduction operator and one or more scalar variables.

- *independent* – tells the implementation that iterations of a loop are data-independent with respect to each other. This allows for executing iterations in parallel with no synchronisation. When used inside of the *parallel* construct, the *independent* clause is implied in all *loop* constructs.

Data is a construct that defines variables to be allocated in the current device memory for the duration of the region, whether data should be copied from local memory to the current device memory upon region entry, and copied from device memory to local memory upon region exit. For this construct the most important clauses are:

- *copy (var-list)* – for each variable *var* in the *var-list* this clause will copy *var* into the device memory at the entry of a given region and copy it to the host at an exit.

- *copyin (var-list)* - for each variable *var* in the *var-list* this clause will copy *var* into the device memory at the entry of a given region and delete it at an exit.

- *copyout (var-list)* - for each variable *var* in the *var-list* this clause will create *var* on the device memory at the entry of a given region and copy it to the host at an exit.

4.5.4. Examples

As done in previous subsections, also this subsection will look at the matrix-vector multiplication *y=Ax* as an example. Firstly, we present a regular single core approach and then, step by step, we present a strategy for efficient acceleration with OpenACC. For these tests, we use a matrix of 16384x16384 and a vector of 16384 elements. The basic single core approach is presented below.

```
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        y[i] += A[i*m+j] * x[j];
```

This task can be parallelised for multiple cores as presented in the listing below. The code can be compiled with use of `nvc -fast -acc=multicore`.

```
#pragma acc parallel loop
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        y[i] += A[i*m+j] * x[j];
```

The first step is to assign the data that will be copied from the host device to the target device. This is done with use of the *copy* clause. Another essential part is to indicate code regions that could be parallelised with *loop* constructs. It is worth mentioning that certain nested loops can not be simply parallelised by adding *loop* construct since they contain data-dependent operations. The example with the *loop* and *copy* constructs is presented below.

```
#pragma acc data copy(A[:(n*m)], x[:(m)], y[:(n)])
```

```
{
    #pragma acc parallel loop
    for(i = 0; i < n; i++)
        for(j = 0; j < m; j++)
            y[i] += A[i*m+j] * x[j];
}
```

Above directives can be shortened because *parallel* (and *kernels* as mentioned earlier in Section 4.5.3, “Programming Model”) share the *data* construct syntax. With that, the usage of those directives can be modified to the following:

```
#pragma acc parallel loop copy(A[:(n*m)], x[:(m)], y[:(n)])
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        y[i] += A[i*m+j] * x[j];
```

It should be noted that due to the high cost of PCIe data transfers between host and device memories, it is often more beneficial to move sections of the application to the GPU, even when the code lacks sufficient parallelism to see a direct benefit. The performance loss of running serial code, or the code with a low degree of parallelism, on a parallel accelerator, is often less than the cost of transferring data back and forth between the two memories.

A step to optimisation of the execution time would be to maintain a data flow between host and target devices. In the presented examples it can be seen that the matrix *A* and vector *x* are not modified during the parallelised computations, so they can be only copied into the target device and deleted from memory at the end, eliminating the need for copying them back to the host device. Therefore, we can use the *copyin* clause instead of *copy*. Also, the vector *y* does not need to be copied in to the device but only results of computations stored in it. So the clause *copyout* can be used here as well. The code for that approach is presented below.

```
#pragma acc parallel loop copyin(A[:(n*m)], x[:(m)]) copyout(y[:(n)])
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        y[i] += A[i*m+j] * x[j];
```

Another optimisation step which can be taken here is to provide a *reduction* in the nested loop. That operation allows for parallelisation of that loop and significantly speeds up the computation. The example of that can be seen in the following listing.

```
#pragma acc parallel loop copyin(A[:(n*m)], x[:(m)]) copyout(y[:(n)])
for(i = 0; i < n; i++)
{
    float y_i = 0;
    #pragma acc loop reduction(+:y_i)
    for(j = 0; j < m; j++)
        y_i += A[i*m+j] * x[j];
    y[i] = y_i;
}
```

It is worth mentioning that some compilers will detect the reduction on *y_i* and implicitly insert the reduction clause, but for maximum portability, the programmer should always indicate reductions in the code.

Alternatively, the *parallel* construct can be replaced with the *kernels*. The important thing to remember in such replacements is that if the first loop is independent, it can be written as *loop independent* contrary to the example above.

```
#pragma acc kernels loop independent copyin(A[:(n*m)], x[:(m)])
        copyout(y[:(n)])
for(i = 0; i < n; i++)
{
    float y_i = 0;
    #pragma acc loop reduction(+:y_i)
    for(j = 0; j < m; j++)
        y_i += A[i*m+j] * x[j];
    y[i] = y_i;
}
```

As can be seen in the final listing the whole process of parallelisation can be done with use of just two *pragmas* added to the base CPU-targeted code. Such parallelised code is then compiled with:

```
nvc -fast -acc=gpu -gpu=cc70
```

It should be noted that the main portion of the time here is consumed by copying the data between the host and the device. Further details on execution time can be found in Section 3.2.1, “GPUs”.

It is worth mentioning that features offered by the V100 model allow for further optimisations. One of those is the specification of *-gpu=managed* compiler flag that enables the usage of unified memory. This setting allows the GPU to consider both RAM and VRAM as one and to maintain the direct communication with each of them. By using this compiler flag, any *acc data* construct will be omitted. The GPU itself will not wait for all data to be copied and will begin a computation earlier.

However, sometimes older techniques can turn out to be better, e.g. by using the *-gpu=pinned* flag which performed better than *-gpu=managed* in those cases. Managed memory simplifies the usage of memory and gives a significant speed-up of workflow. Pinned memory is oriented to speed-up data transfer by directly allocating host pined memory, but needs more attention in usage, because over-allocation of the pinned memory, can reduce overall system performance. In the considered example pinned memory performs better because each element of matrix A is accessed only once. In case of more frequent data flow between the host and the device, the better performance of the pinned memory may not be kept. It is hard to predict which one will perform better with OpenACC and that is why both *-gpu=managed* and *-gpu=pinned* flags are equally recommended to check in search of a better performance. Usually, when there is no time to research memory management, it is suggested to just use managed memory and take the advantage of not using the *data* construct.

For more details about OpenACC, we strongly recommend checking documentation [31] and other Best Practice Guides focused on OpenACC [32].

4.6. HIP

Heterogeneous-Computing Interface for Portability (HIP) is a C++ Runtime API and Kernel Language developed by AMD that allows developers to create portable applications for AMD and NVIDIA GPUs from a single source code [33]. It provides a C-style API and a C++ kernel language. The C++ interface can use templates and classes across the host/kernel boundary.

The HIPify tools automates the conversion source-to-source from CUDA to HIP. HIP code can run on AMD hardware (through the HCC compiler) or NVIDIA hardware (through the nvcc compiler).

4.6.1. HIP API

The HIP API includes functions such as hipMalloc, hipMemcpy, and hipFree. Programmers familiar with CUDA will also be able to quickly learn and start coding with the HIP API [34]. Compute kernels are launched with the ‘hipLaunchKernel’s macro call. For example:

```
 hipMalloc(&A_d, Nbytes));
 hipMalloc(&C_d, Nbytes));

 hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);

 const unsigned blocks = 512;
 const unsigned threadsPerBlock = 256;
 hipLaunchKernelCGL(vector_square,   /* compute kernel*/
                dim3(blocks), dim3(threadsPerBlock), 0/*dynamic shared*/,
                0/*stream*/,     /* launch config*/
                C_d, A_d, N);  /* arguments to the compute kernel */

 hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```

The HIP kernel language defines built-ins for determining grid and block coordinates, math functions, short vectors, atomics, and timer functions. It also defines additional keywords for function types, address spaces, and optimisation controls.

The kernels require the caller to specify a configuration that includes the grid and block dimensions. The execution configuration can also include other information for the launch, such as the amount of additional shared memory to allocate and the stream where the kernel should execute. HIP introduces a standard C++ calling convention to pass the execution configuration to the kernel in addition to the Cuda <<< >>> syntax.

In HIP, Kernels launch with the “hipLaunchKernelGGL” function. The first five parameters of the *hipLaunchKernelGGL* function are:

- *symbol kernelName*: the name of the kernel to launch.
- *dim3 gridDim*: 3D-grid dimensions specifying the number of blocks to launch.
- *dim3 blockDim*: 3D-block dimensions specifying the number of threads in each block.
- *size_t dynamicShared*: amount of additional shared memory to allocate when launching the kernel.
- *hipStream_t*: stream where the kernel should execute. A value of 0 corresponds to the NULL stream.

The hipLaunchKernelGGL macro always starts with the five parameters specified above, followed by the kernel arguments. The dim3 constructor accepts zero to three arguments and will by default initialise unspecified dimensions to 1. A kernel-launch example:

```
// Example showing use of host/device function
__host__ __device__
float PlusOne(float x)
{
   return x + 1.0;
}

__global__
void
MyKernel (const float *a, const float *b, float *c, unsigned N)
{
   unsigned gid = hipThreadIdx_x; // - coordinate index function
   if (gid < N) {
       c[gid] = a[gid] + PlusOne(b[gid]);
   }
}
```

```
void callMyKernel()
{
    float *a, *b, *c; // initialisation not shown...
    unsigned N = 1000000;
    const unsigned blockSize = 256;
    hipLaunchKernelGGL(MyKernel,
    (N/blockSize), dim3(blockSize), 0, 0,  a,b,c,N);
}
```

4.6.2. HIPify Tools

The AMD ROCm platform includes tools to automatically convert CUDA codes to HIP, like *hipify-perl* and *hipify-clang*.

The *hipify-perl* tool is an autogenerated perl-based script which heavily uses regular expressions. It is very easy to use and it does not depend on having CUDA installed, but it has limitations. The following constructs can not be transformed/addressed with hipify-perl:

- macros expansion
- namespaces
- distinction between device/host function calls
- correct injection of header files
- complicated argument lists parsing

An example of usage of hipify-perl:

```
$ perl hipify-perl square.cu > square.cu.hipc
```

The *hipify-clang* is a clang-based tool for translating CUDA sources into HIP sources. It translates CUDA source into an abstract syntax tree, which is traversed by transformation matchers. After applying all the matchers, the output HIP source is produced. This requires CUDA to be installed. In case of multiple CUDA installations the *--cuda-path* option should be used. Also, the input CUDA code should be correct or it would not be translated to HIP. An example of usage is provided below.

```
$ ./hipify-clang square.cu --cuda-path=/usr/local/cuda-11.2 -I /usr/lo-
cal/cuda-11.2/samples/common/inc
```

4.6.3. Examples

As an example of HIP usage, we consider the matrix-vector multiplication algorithm, $y = Ax$. In this case, we will translate the CUDA example code using the HIPify tools. The translated kernel looks like this:

```
#include "hip/hip_runtime.h"
// a kernel that performs a matrix-vector multiplication y = A * x, where
// the matrix A has m rows and n columns
__global__ void gemv_kernel(
    int m, int n, int ldA, double const *A, double const *x, double *y)
{
    // we are assuming that each row of the vector y gets its own thread
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    // threads that do not contribute to the final result skip to the
    // end of the kernel
```

```
    if (thread_id < m) {

        //
        // loop over the corresponding row of the matrix A and the
        // vector x
        //
        // |y_0|   |A_00 A_01 A_02 .... |   |x_0|
        // |y_1|   |A_10 A_11 A_12 .... |   |x_1|
        // |y_2| = |A_20 A_21 A_22 .... | * |x_2|
        // |...|   |.... .... .... .... |   |...|
        // |...|   |.... .... .... .... |   |...|
        //
        // y_k = A_k0 * x_0 + A_k1 * x_1 + A_k2 * x_2 + ...
        //
        double y_k = 0.0;
        for (int i = 0; i < n; i++)
            y_k += A[i*ldA+thread_id] * x[i];

        // store the result to the vector y
        y[thread_id] = y_k;
    }
}
```

And the translated host code looks like:

```
#include <stdlib.h>

__global__ void gemv_kernel(
    int m, int n, int ldA, double const *A, double const *x, double *y);

int main(int argc, char **argv)
{
    int m = 1000, n = 1000, ldA = 1000;
    double *A, *y, *x;           // host memory pointers
    double *d_A, *d_y, *d_x;     // global memory pointers
    srand(time(NULL));

    // allocate host memory for the matrix A and the vectors y and x

    A = (double *) malloc(n*ldA*sizeof(double));
    y = (double *) malloc(m*sizeof(double));
    x = (double *) malloc(n*sizeof(double));

    // initialise host memory

    for (int i = 0; i < n; i++) {
        x[i] = 2.0*rand()/RAND_MAX - 1.0;
        for (int j = 0; j < m; j++)
            A[i*ldA+j] = 2.0*rand()/RAND_MAX - 1.0;
    }

    // allocate global memory

    hipMalloc(&d_A, n*ldA*sizeof(double));
    hipMalloc(&d_y, m*sizeof(double));
```

```
    hipMalloc(&d_x, n*sizeof(double));

    // copy the matrix A and the vector x from the host memory to the
    // global memory

    hipMemcpy(d_A, A, n*ldA*sizeof(double), hipMemcpyHostToDevice);
    hipMemcpy(d_x, x, n*sizeof(double), hipMemcpyHostToDevice);

    // launch the kernel

    dim3 threads = 128;
    dim3 blocks = (m+threads.x-1)/threads.x;
    hipLaunchKernelGGL(gemv_kernel, dim3(blocks), dim3(threads),
                       0, 0, m, n, ldA, d_A, d_x, d_y);

    // copy the vector y from the global memory to the host memory

    hipMemcpy(y, d_y, m*sizeof(double), hipMemcpyDeviceToHost);

    // free the allocated memory

    free(A); free(y); free(x);
    hipFree(d_A); hipFree(d_y); hipFree(d_x);
    return EXIT_SUCCESS;
}
```

We can observe that the translator has added the following changes:

- Include directive with the HIP headers.
- String replacement of all "cuda" occurrences - are replaced by "hip".
- Replacement of the kernel launch using hipLaunchKernelGGL instead of the Cuda <<< >>> syntax.

4.7. HLS

FPGAs are commonly programmed at the circuit level, using Hardware Description Languages (HDLs) such as Verilog or VHDL. However, there has been a significant effort to raise the abstraction level and use languages such as C and OpenCL to program FPGAs. This is known as High-Level Synthesis (HLS). Current tools are not yet as mature as other technologies currently in use in HPC, but are already at a stage where it is possible to create efficient designs for certain types of workloads. Ideal targets for HLS are applications that can be mapped to a hardware implementation, such as e.g. a neural network; further optimisations are enabled at low effort to the programmer, if the computation can be expressed by the streaming paradigm, as shown below.

The two main FPGA vendors, Xilinx and Intel, each have their own toolchain for programming and developing for FPGAs. The flagship development platform for Xilinx is named Vitis[2], while Intel's is named Quartus Prime[3]. Xilinx favors an HLS approach based on C/C++ (although it also supports OpenCL), while Intel currently fully relies on OpenCL, with plans to use OneAPI for FPGAs [24].

Certain companies provide FPGA-based solutions for specific classes of problems, where the FPGA is not directly programmed. For instance, Maxeler sells FPGA-based dataflow engines that are programmed with a dialect of

[2] https://www.xilinx.com/products/design-tools/vitis.html
[3] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

Java[4], and MathWorks has a MATLAB and Simulink package that translates projects written in MATLAB code and Simulink models to HDL for FPGAs[5].

4.7.1. Xilinx Vitis HLS

Vitis HLS is Xilinx's high-level synthesis tool that maps kernels written in C, C++, and OpenCL to HDL. The preferred development flow is split into three consecutive stages:

- Software emulation
- Hardware emulation
- Hardware implementation

Software emulation is usually fast, and should be used to validate the algorithm being implemented. Hardware emulation is slower, but validates most of the actual behaviour being implemented, including host/kernel interaction, and enables the use of the profiler for debugging and optimisation. This is where developers should preferably spend most of their development effort. Hardware implementation can be a very slow process, where compilation times can start at half-an-hour, easily achieving several hours. However, after compilation, execution usually is fast. At this stage we verify the performance estimates and possibly catch any remaining bugs not present during hardware emulation.

Although Vitis allows programming Xilinx FPGAs with C/C++, this is still a very leaky abstraction [108] that does not hide hardware-related details and relies heavily on custom pragmas to guide the compiler. This means that programmers should always take into account that the code will ultimately be converted to hardware, and write code that takes advantage of custom circuits. This usually translates to writing loops that use pipelining, unrolling and dataflow pragmas. The learning curve can be steep, and building visual models, such as dataflow diagrams, may help in locating opportunities for optimisation. Dataflow in particular (i.e. task-level parallelism of functions and loops) is a great way of increasing performance with relatively low effort from the programmer, since the compiler automatically handles task scheduling.

Other issues that should be taken into account include the mapping of kernel operands to interfaces, bundling interfaces to avoid performance issues, and optimally mapping operands to memory banks. For instance, arrays should be implemented as hardware memories, for performance reasons, which means declaring them as static. Usually, large C/C++ kernels are preferable to small OpenCL kernels with task-level parallelism, following Xilinx recommendations [107]. As with other platforms and environments, it is recommended to take advantage, whenever possible, of available optimised libraries. The Vitis environment provides a math library[6] and, more recently, a set of libraries for high-level tasks[7]. As of Vitis version 2020.2, the library collection includes, among others:

- Vitis BLAS library (linear algebra)
- Vitis Data Analytics library
- Vitis Data Compression library
- Vitis DSP library (Digital Signal Processing)
- Vitis Quantitative Finance library
- Vitis Security library (hashes and cyphers)
- Vitis Vision library

Note that the state of maturity varies from library to library: some are at a quite early stage. Each library is divided into L(evel) 1 (primitive), L2 (kernel), and L3 (application), meant for FPGA-level, host-level, and pure software developers, respectively; for instance, there exist L3 Python bindings for several libraries.

[4] https://www.maxeler.com/technology/dataflow-computing/
[5] https://www.mathworks.com/discovery/fpga-programming.html
[6] https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_math_library.html
[7] https://xilinx.github.io/Vitis_Libraries/

Operators are code features rendered atomic and mapped onto the hardware (several forms of the same operator may exist). Such operators include integer addition and multiplication, signed and unsigned division and remainder, etc. Operators have a direct impact on the utilised area. By default, Vitis prioritises performance over area expenditure, but this may come at the cost of optimisations in another code region. In addition, operators are sized according to datatype, meaning knowledge of the potential size of a variable, to the bit level, is useful for optimisation. Data types can have arbitrary precisions (e.g. 17-bit), which can be a source of area or delay benefits. For instance, a 3-level operator using 12 Look-up Tables (LUTs) can be transformed into a 1-level operator using 2 LUTs by using arbitrary precision type variables. Data dependencies in loops limit the "mobility" of operators. This means that physical placement and scheduling is more constrained, leading to potential optimisation losses. Therefore, we want to reduce data-dependencies as much as possible by restructuring loops.

Vitis HLS defines a set of pragmas for code optimisation (as well as analysis)[8]. As mentioned above, the most important pragmas to keep in mind are the pipelining, unrolling, and dataflow pragmas. Below, we showcase the former two and present a more extensive example of the latter.

4.7.1.1. Unrolling

The `unroll` pragma, when applied to a loop, results in multiple copies of the loop being synthesised and implemented in hardware, instead of a single one. This will of course result in performance benefits, at the risk of a potentially significant FPGA area expenditure increase (which worsens in accordance to the number of unrolled iterations).

Full loop unrolling requires knowledge of the total number of iterations at compile time. However, Vitis also supports a *partial* unroll, where the user determines the number of iterations to synthesise. An exit check is also created, in case of remainder iterations.

Consider a classic graphics *bit blit*. The objective is to draw the result of a bitwise boolean operation over two bitmaps onto a screen. Assume a 1D array with a size corresponding to the screen resolution, which is known to be 160x144 px, and that a pixel is represented by an arbitrary `int`.

```
for(int i = 0; i < 160*144; ++i) {
    SCREEN[i] = A[i] | B[i];
}
```

In this instance, the iteration count is fixed. A programmer may therefore wish to generate hardware to blit all pixels at the same time. This may be achieved by fully unrolling the loop.

```
for(int i = 0; i < 160*144; ++i) {
#pragma HLS unroll
    SCREEN[i] = A[i] | B[i];
}
```

Note that there is a caveat to unrolling: there must be a way of making memory accesses in parallel, or Vitis will need to construct a state machine to handle access control, slowing down the kernel. This is often done using the `array_partition` pragma, which decomposes an array into smaller memory elements. A good rule of thumb is to make sure that the number of elements processed per synthesised loop unroll matches the partition size. Another rule of thumb is to only unroll loops with relatively small iteration counts, in order to keep area consumption low, and allow the compiler to better optimise the remaining code.

4.7.1.2. Pipelining

Unlike unrolling, pipelining affects throughput but not latency. Besides loops, functions are also supported. Pipelining works by decomposing a code block into its constituent operations, and minimising the time each operation spends unused by feeding further inputs to the pipeline while the ones before are still being processed.

[8] A comprehensive list is available here. [https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_pragmas.html]

Also unlike unrolling, pipelining does not require the number of iterations to be known at compile time (when dealing with loops), and results in minimal area increase. This makes pipelining somewhat more common than unrolling.

As an example, consider again the unrolling example, but restructuring the data into 2D arrays and assuming a large resolution such that unrolling the loop is unacceptable:

```
void screen_blit(int *in1, int *in2, int *screen) {
    for (int i = 0; i < HEIGHT; ++i)
        for (int j = 0; j < WIDTH; ++j) {
            screen[i][j] = in1[i][j] | in2[i][j];
        }
}
```

The pipeline pragma will automatically result in unrolling any nested loops. In some instances, pipelining a top-level loop or function may not be the right call: HLS pipelines tend to be more performant the closest they are to the processing of individual inputs (further down the loop tree), due to unrolling and the necessity to keep data transfers per clock high to better utilise the parallel resources.

```
void screen_blit(int *in1, int *in2, int *screen) {
    for (int i = 0; i < HEIGHT; ++i)
        for (int j = 0; j < WIDTH; ++j) {
#HLS pragma pipeline
            screen[i][j] = in1[i][j] | in2[i][j];
        }
}
```

Therefore, although we could place the pipeline pragma at function level (which would attempt to consume the entirety of the arrays in a single clock cycle), or at the first loop level (which would attempt to consume a screen line in a single clock cycle), we place it at the pixel level, achieving maximum optimisation.

4.7.1.3. Dataflow: a larger example using libraries

Whereas the pipeline pragma applies fine-grained, operator level pipelining, the dataflow pragma results in coarse-grained, task-level pipelining.

The dataflow pragma applies to an entire code block, function, or loop. The compiler then automatically analyses dependencies between constituents (which may be either functions or loops), makes sure they form a directed acyclic graph, transforms the region, including creation of the channels between tasks, and schedules those tasks concurrently.

The dataflow paradigm is the recommended coding pattern, whenever possible. It allows a significant amount of parallelism to be extracted at low cost to the programmer. However, the large degree of underlying automation enabling programmers to simply add a dataflow pragma comes at a cost: within a dataflow region, a specific, strict coding style must be followed (termed *dataflow canonical form*), under penalty of poor performance or failure to compile. In addition, certain features *should not* be present, such as conditional execution of tasks (for instance, a function), or early loop exit.

Certain types of computations are more suited to the dataflow programming than others. Most importantly, streaming-type computations can often be expressed as a dataflow. A recurring example is image processing, where several input images may be processed separately, blended, decomposed into channels, processed again, and so forth.

The following is an example of a full, runnable kernel, implementing the creation of a 3D anaglyph. The kernel is quite simple: it takes as input two stereo images (pictures of the same scene, with their perspectives offset at around the same magnitude as naturally in human binocular vision), and outputs a single image combining the red

and blue channels of the first image with the green channel of the second, thus creating a crude red-blue 3D image (meant for viewing with red-blue 3D glasses). We will make heavy use of the Vitis Vision library (`xf::cv`).

We will now explain some unintroduced concepts:

- `XF_8UC3`, `XF_8UC4`: Shorthands for 8 bits per channel, 3 and 4 channels respectively.
- `XF_NPPC`: Number of pixels per cycle (and per word), for Vision library optimisations. Can be safely ignored.
- `ap_uint`: The Vitis implementation of the arbitrary precision types mentioned above. Note the parametrisation; 32 bits per element for `img1`, `img2`, and `img_out`. While `img1` and `img2` have 24-bit pixels, we want to align to the default width of 32 bits used by the `m_axi` interface.
- `pragma HLS INTERFACE`: Each kernel parameter must correspond to a hardware interface. We have specified full AXI4 interfaces for our arrays and AXI4-Lite interfaces for scalar values. Note in particular the `bundle` parameter. This is an optimisation: we have chosen to implement our array inputs as separate ports, thus allowing concurrent transfers from the host.
- `XF_TNAME`: Converts a NPPC and XF type specification to an arbitrary precision type.

```
#define HEIGHT 1080
#define WIDTH 1920

#define IN_TYPE XF_8UC3
#define OUT_TYPE XF_8UC4
#define NPC XF_NPPC1

#define PTR_IN_WIDTH 32
#define PTR_OUT_WIDTH 32

extern "C" {

void make_anaglyph( ap_uint<PTR_IN_WIDTH>* img1_in,
                    ap_uint<PTR_IN_WIDTH>* img2_in,
                    ap_uint<PTR_OUT_WIDTH>* img_out,
                    int rows,
                    int cols) {

    #pragma HLS INTERFACE m_axi         port=img1_in    offset=slave \
  bundle=gmem0
    #pragma HLS INTERFACE m_axi         port=img2_in    offset=slave \
  bundle=gmem1

    #pragma HLS INTERFACE s_axilite     port=rows
    #pragma HLS INTERFACE s_axilite     port=cols

    #pragma HLS INTERFACE m_axi         port=img_out    offset=slave \
  bundle=gmem0

    #pragma HLS INTERFACE s_axilite     port=return


    xf::cv::Mat<IN_TYPE, HEIGHT, WIDTH, NPC> imgInput1(rows, cols);
    xf::cv::Mat<IN_TYPE, HEIGHT, WIDTH, NPC> imgInput1_r(rows, cols);
    xf::cv::Mat<IN_TYPE, HEIGHT, WIDTH, NPC> imgInput1_g(rows, cols);
    xf::cv::Mat<IN_TYPE, HEIGHT, WIDTH, NPC> imgInput2(rows, cols);
```

```
    xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, NPC> img_r(rows, cols);
    xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, NPC> img_g(rows, cols);
    xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, NPC> img_b(rows, cols);

    xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, NPC> img_a(rows, cols);

    xf::cv::Mat<OUT_TYPE, HEIGHT, WIDTH, NPC> imgOutput(rows, cols);

    int i, a_size;


#pragma HLS DATAFLOW

    // Fill our streams.
    xf::cv::Array2xfMat<PTR_IN_WIDTH, IN_TYPE, HEIGHT, WIDTH, NPC>
        (img1_in, imgInput1);
    xf::cv::Array2xfMat<PTR_IN_WIDTH, IN_TYPE, HEIGHT, WIDTH, NPC>
        (img2_in, imgInput2);


    // Duplicate our first image: we'll be taking two channels from it.
    xf::cv::duplicateMat<IN_TYPE, HEIGHT, WIDTH, NPC>
        (imgInput1, imgInput1_r, imgInput1_g);


    // Extract RGB channels.
    xf::cv::extractChannel<IN_TYPE, XF_8UC1, HEIGHT, WIDTH, NPC>
        (imgInput1_r, img_r, XF_EXTRACT_CH_R);
    xf::cv::extractChannel<IN_TYPE, XF_8UC1, HEIGHT, WIDTH, NPC>
        (imgInput1_g, img_g, XF_EXTRACT_CH_G);
    xf::cv::extractChannel<IN_TYPE, XF_8UC1, HEIGHT, WIDTH, NPC>
        (imgInput2, img_b, XF_EXTRACT_CH_B);


    // Fill our alpha channel.
    a_size = rows*cols;
    for(int i = 0; i < a_size; ++i) {
#pragma HLS PIPELINE
     img_a.write(i, (XF_TNAME(IN_TYPE, NPC)) 255);
    }


    // Merge all channels...
    xf::cv::merge<XF_8UC1, OUT_TYPE, HEIGHT, WIDTH, NPC>
        (img_b, img_g, img_r, img_a, imgOutput);

    // and convert to output.
    xf::cv::xfMat2Array<PTR_OUT_WIDTH, OUT_TYPE, HEIGHT, WIDTH, NPC>
        (imgOutput, img_out);

    return;

}}
```

Keep in mind that this is merely an example, and further optimisation is very much possible. However, while barely writing any raw code (mostly sticking to boilerplate, specifying a dataflow region, and a single loop to fill

the alpha channel), we have already managed to construct an FPGA implementation of the application, which will likely provide significant acceleration.

4.7.1.4. Additional hints on using HLS

The bullet list below provides some additional hints one might wish to consider before starting to use HLS:

- Take some extra time to read on any libraries you plan on using, and be aware that documentation is even more important compared to "regular" software development: HLS boilerplate makes heavily optimised code quite difficult to read, especially for beginners.
- Try not to start from scratch, but rather follow some example code you feel is similar to what you are trying to achieve: there is a lot to learn on your first run.
- As always, avoid premature optimisations. A trick to consider is using and abusing dataflow pragmas in conjunction with encapsulated behaviour in functions (you want this to be "operators"), and later restructuring the computation to involve less overhead. You will likely not lose anything by doing this, since dataflow is the recommended coding style whenever possible. Other optimisations can come later.
- Do not jump immediately into hardware emulation (and certainly not into hardware implementation). Software emulation lets you explore a bit more freely and already reports some synthesis information. Move ahead when you are satisfied with your code and with the diagnostic tools available during software emulation.
- Likewise, take your time with hardware emulation. Pay special attention to the application timeline report, and try waveform debugging (enabled in Run Configuration). These are visual tools, and are likely to give you hints as to where to optimise.
- Make sure to take note of the effect your optimisations/pragmas have on the profiling data and resulting performance. It is important to build up an intuition on how to optimise.
- Encountering very strange bugs might mean that the software abstraction has broken down; you might need to look more closely to the code, delve into the documentation, or try to isolate and rewrite the offending lines.

4.8. NEC Programming Environment

One advantage of the NEC SX-Aurora TSUBASA compared to graphics processing units is that no new programming model is required. As for ordinary SIMD architectures, vector operations are generated by the NEC compilers when loops are eligible for vectorisation. MPI or OpenMP can be applied to generate parallel code for the processor's 8 cores. Therefore this section will give an overview of NEC's compilers and MPI. Furthermore, we introduce the three different execution modes for SX-Aurora TSUBASA.

4.8.1. NEC Compilers

NEC provides several compilers that generate assembly code for the SX-Aurora TSUBASA. These are listed in Table 8. The C/C++ compiler supports C++11, C++14 and some C++17 features [85]. The Fortran compiler supports Fortran 2003 [84]. Both compilers support OpenMP 4.5 and MPI 3.1.

Table 8. NEC Compilers for SX-Aurora TSUBASA [87]

Language	Compiler Name
C/C++	`ncc`
C/C++ with MPI support	`mpincc`
Fortran	`nfort`
Fortran with MPI	`mpnfort`

Like other compilers, NEC compilers offer `-O0`, `-O1`, `-O2` and `-O3` optimisation levels. Instead of `-Ofast`, there is another optimisation level called `-O4`. While `-O2` enables basic optimisation, `-O3` adds mostly loop transfor-

mations and `-O4` adds unsafe (loop) optimisations. In this context, “unsafe” means that the optimised program does not necessarily produce the same output as the original program. To view the optimisations enabled by the compiler, one can use the flag `-report-cg`. A higher optimisation level does not guarantee lower computation times because optimisations can be unfavourable on rare occasions.

Automatic inlining of functions (routines) in C/C++ (Fortran) is enabled when using `-finline-functions`. This feature can be controlled with `-finline-max-depth=n`, `-finline-max-function-size=n` and `-finline-max-times=n`. Cross-file-inling works in C and Fortran and can be activated with `-finline-file=file1:file2:file3`. To confirm specific inlining decisions, one should consult the output generated by the flag `-report-inline`.

Setting `VE_TRACEBACK=ALL` and compiling with `-traceback -g` will cause program failures to result in printed stack traces in hex format. Those hex addresses can be translated to files and line-numbers with the `naddr2line` tool located at `/opt/nec/ve/bin/`. For a more detailed overview of all compiler flags, please consider the C/C++ and Fortran compiler manuals [84, 85].

4.8.2. NEC MPI

NEC MPI implements the MPI 3.1 standard and behaves accordingly [86]. Additionally, it extends pinning to Vector Hosts (VH) and Vector Engines (VE). Inspired by [86], we present the most important features by example:

`mpincc program.c` will generate a native VE program.
`mpincc -vh program.c` will employ GCC by default to generate a VH program.

`mpirun` in combination with the `-ve` flag is used to execute the program on one or more vector engines.
`mpirun -ve 5 -np 8 ./program` will launch 8 processes on vector engine #5.
`mpirun -ve 4-7 -np 32 ./program` will launch 8 processes on vector engine #4 to #7.
`mpirun -nve 8 -np 64 ./program` will launch 64 processes on vector engine #0 - #7.

Assuming that there is a total of 8 VEs, the previous command is equivalent to
`mpirun -ve 0-7 -np 64 ./program` and
`mpirun -np 64 ./program`

Multiple VHs can be specified with `-host`:
`mpirun -host vh1,vh2 -nve 8 -np 128 ./program`
will launch 8 processes on each of the 8 VEs on both VHs.
`mpirun -host vh1 -ve 0-7 -np 64 -host vh2 -ve 0-3 -np 32 ./program`
will launch 64 processors on 8 VEs on `vh1` and only 32 processors on 4 VEs of `vh2`.

To start a heterogeneous application that runs on VHs and on VEs, the binary has to exist in two versions, compiled for each architecture:
`mpirun -vh -host vh1 -np 24 ./vh_bin : -host vh1 -nve 8 -np 64 ./ve_bin`
will launch 24 ranks on the VH and 64 ranks on the VEs.

Besides that, NEC MPI offers many environment variables to control the MPI behaviour. Further information can be found in the corresponding documentation [86].

4.8.3. Execution Mode

The vector engine is controlled by the vector engine operating system (VEOS) which runs on the vector host, as shown in Figure 26. This combines the general purpose functionality of the x86 system with flexible management of the VE.

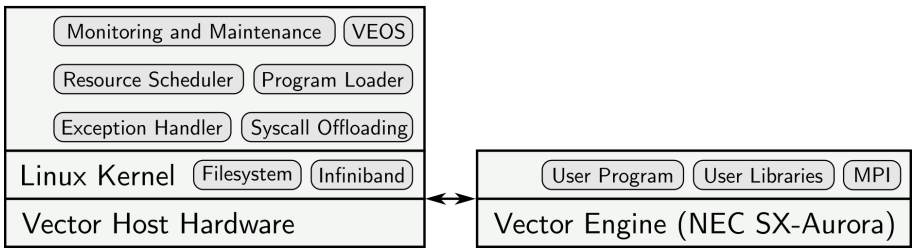

Figure 26. Software structure on SX-Aurora inspired by [81]

Depending on how VEOS is instructed, a user program can run on the VE in three different ways that are visualised in Figure 27:

1. Natively running on the vector engine, only using the vector host for system-calls. The advantage of this method is, that nearly the whole program is executed on the vector engine and no user-data has to be communicated through PCIe. The restrictions to the device can be challenging for applications that require large amount of main memory or applications that contain inherently scalar parts.

2. The classical accelerator model in which the vector host offloads those sections to the vector engine that benefit most from vectorisation and high memory bandwidth. However, there are two pitfalls. Firstly, the VE might idle when there is not enough work to offload, leading to a decrease in performance. Secondly, the PCIe 3.0 connection provides only 16 GB/s bandwidth and might become a bottleneck when a lot of data has to be exchanged between VH and VE. The procedure for this execution mode is, that the VE-code is compiled into a shared library by the NEC compiler which can then be called from normally compiled VH-code. As this guide focuses on the native execution mode, [82] can be considered for further instructions on manual VE-offloading.

3. The opposite way where the program is started on the vector engine and scalar user-workload is offloaded to the vector host [83]. Apart from the assumption that scalar workload may require less data-transfers, the same challenges as in the accelerator-mode exist.

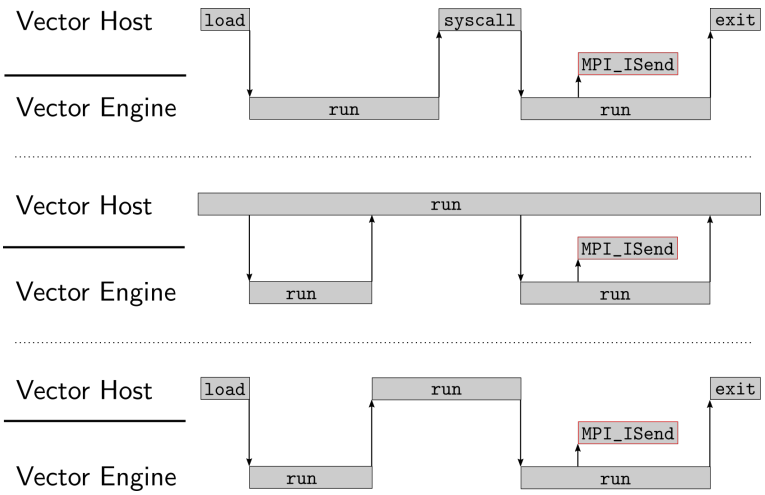

Figure 27. Three possible execution modes: native, hybrid and VH offloading inspired by [81]

4.9. Summary

The following list of items summarise the most important aspects discussed in this section.

- CUDA supports only NVIDIA GPUs but can be used to access the entire feature set of modern NVIDIA GPUs. CUDA SDK comes with a large set of vendor-optimised libraries.

- OpenCL is an open standard that resembles CUDA but supports most considered accelerator devices.

- SYCL runs over the OpenCL runtime and allows portability among CPU, NVIDIA GPUs and AMD GPUs.
- SYCL aims to ease the software development based on the OpenCL runtime, by exploring the object oriented C++ features such as inheritance, templates and lambda functions.
- OpenACC is a high level approach that allows the transformation of a C, C++ or Fortran sequential base code to run on parallel by adding pragma directives.
- Depending on the compiler, OpenACC allows parallelising code on multicore CPUs, as well as NVIDIA or AMD GPUs. Allowing also for optimisation by only adding compiler flags.
- HIP supports AMD GPUs and NVIDIA GPUs. Programmers familiar with CUDA will also be able to quickly learn and start coding with HIP.
- HIPify tools aim to convert CUDA codes to HIP. There are two tools included for that purpose, hipify-perl and hipify-clang.
- HLS is an emerging alternative to HDLs for synthesising circuitry. The goal is to raise the abstraction level further, allowing for more complex systems to be developed. The HLS market for FPGAs is currently dominated by Xilinx and Intel, the main FPGA vendors; alternatives do exist, however.
- Xilinx HLS uses C/C++ with custom pragmas. The development flow is split into three (consecutive) stages: (i) software emulation; (ii) hardware emulation; and (iii) hardware implementation.
- The abstraction provided by HLS is leaky. Third party tools such as libraries are appearing, which may alleviate this issue. Regardless, the preferred computational paradigm for HLS is dataflow: whenever possible, the HLS programmer should structure their kernels in this manner.
- NEC vector processors work with standard C/C++ and Fortran code.
- NEC provides a customized MPI implementation that extends mpirun for vector host and vector engine control.

5. Performance Analysis and Tuning

This section explains the tools and methodologies that are required for performance analysis and tuning the code. In scientific computing, mathematical models are converted into computer code using many programming languages. However, most of the time, these algorithms are not well optimised to utilise the given hardware, such as processor and accelerators. Computer memory is one of the main aspects to look for in performance analysis and tuning the code. Keeping the data close to the processor memory (register, cache, etc.) will give a better performance than keeping it, for example, in the main memory or hard disk. It is equally important to analyse applications performance bottlenecks (i.e. determining whether an application's execution time is dominated by communication, computation, etc.).

This section discusses how the contemporary tools would help to analyse the code and suggest some tuning options. More specifically, this section starts with explaining how to use the NVIDIA profiling tools for CUDA, OpenACC and OpenMP models on NVIDIA GPUs as well as a few tuning options for CUDA platform. It is then followed by the explanation of the PROGINF, FTRACE, and veperf tools that can be used for the NEC Vector processor to achieve better performance. The section concludes by explaining the usage of Extrae/Paraver toolkit used for hybrid programming environments where both processor and accelerators participate in computation and communication.

5.1. Profiling Tools

This subsection explains how to profile and optimise code that runs on the NVIDIA GPUs involving hybrid programming that uses both CPU and NVIDIA GPUs (see Section 4.2, “CUDA”). GPUs have many cores and different memory options. In order to write an optimised code on the GPU, it should be profiled and tuned. NVIDIA provides four kinds of profiling options for CUDA, OpenACC and OpenMP models. They are `nvprof` [45], `Visual Profiler` [46], `NVIDIA Nsight Systems` and `NVIDIA Nsight Compute` [47]. Among these, `nvprof` and `Visual Profiler` will be deprecated and no longer support future NVIDIA GPUs. The NVIDIA Volta platform is the last architecture that can support both `nvprof` and `Visual Profiler` [45].

5.1.1. nvprof

`nvprof` is a command-line profiling tool from NVIDIA, which supports both events (an event is a countable activity, action, or occurrence on a device) and metrics (a metric is a characteristic of an application calculated from one or more event values) from the application. In general, these events and metrics lists can be queried by `nvprof --query-events` and `nvprof --query-metrics` [44]. `nvprof` will collect the events, metrics, timeline of CUDA-related activities on both CPU and GPU. It mainly collects information on kernel execution, memory transfer, memory set, and other CUDA API calls. The profiled result can be viewed from the console or saved (`--log-file`) in an output file that can be opened in `Visual Profiler`. By default, `nvprof` will profile the entire application. However, sometimes, there is no need for profiling the entire application and only specific application regions can be profiled. For doing that, an application should have a `cuProfilerStart()` and `cuProfilerStop()` and should have the header files `cuda_profiler_api.h` or `cudaProfiler.h`. The below example shows how to profile just kernel calls in the CUDA application.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_profiler_api.h>
#include <cudaProfiler.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
```

```
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    // Initialise vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // cuda profile starts
    cuProfilerStart()

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // cuda profile ends
```

```
    cuProfilerStop()

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n,
    // this should equal 1 within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

Furthermore, during the execution time, `nvprof` can be set to `nvprof --profile-from-start off`. This specification disables the entire code profiling for a given CUDA application. Example 1 below illustrates a sample output of entire code profiling using `nvprof` for a vector addition CUDA application. The following command list shows the compilation and profiling the CUDA code:

```
# Compilation
nvcc -arch=sm_70 vector-add.cu -o vector-add

# Code execution
nvprof ./vector-add
```

Example 1. `nvprof` default summary mode

```
==80299== Profiling application: ./vector-add
==80299== Profiling result:
Type
Time(%)      Time Calls       Avg       Min       Max  Name
GPU activities:
 59.89%  7.1680us     2  3.5840us  3.5840us  3.5840us  [CUDA memcpy HtoD]
 24.33%  2.9120us     1  2.9120us  2.9120us  2.9120us  [CUDA memcpy DtoH]
 15.78%  1.8880us     1  1.8880us  1.8880us  1.8880us  vector_add(float*,
                                                       float*, float*, int)
API calls:
 99.47%  212.96ms     3  70.985ms  2.4710us  212.95ms  cudaMalloc
  0.20%  422.01us    97  4.3500us     111ns  164.06us  cuDeviceGetAttribute
  0.16%  341.11us     1  341.11us  341.11us  341.11us  cuDeviceTotalMem
  0.10%  205.20us     3  68.398us  4.2400us  187.24us  cudaFree
  0.04%  79.783us     3  26.594us  17.353us  32.998us  cudaMemcpy
  0.02%  51.755us     1  51.755us  51.755us  51.755us  cudaLaunchKernel
  0.02%  35.662us     1  35.662us  35.662us  35.662us  cuDeviceGetName
  0.00%  7.1410us     1  7.1410us  7.1410us  7.1410us  cuDeviceGetPCIBusId
  0.00%  1.3980us     3     466ns     118ns     957ns  cuDeviceGetCount
  0.00%     566ns     2     283ns     171ns     395ns  cuDeviceGet
  0.00%     198ns     1     198ns     198ns     198ns  cuDeviceGetUuid
```

Nowadays, it is quite common to use multiple GPUs on a single compute node. In this situation, a multiple GPU CUDA codes can also be profiled using `nvprof --print-summary-per-gpu ./a.out`, which will give profiling information for each GPU. The `nvprof --print-gpu-trace ./a.out` can be used for querying a detailed report of each GPU activity. Figure 28 shows the GPU trace for vector multiplication code. Here, we can see the information about the time taken for the CUDA API calls, kernel calls, a number of registers used and shared/dynamic memory allocated per thread block.

```
==83573== Profiling application: ./vector-mul
==83573== Profiling result:
   Start  Duration           Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType
          Device   Context    Stream  Name
279.21ms  3.5840us                    -               -         -         -         -  20.000KB  5.3218GB/s    Pageable      Device
  Tesla V100-SXM2         1         7  [CUDA memcpy HtoD]
279.23ms  3.6160us                    -               -         -         -         -  20.000KB  5.2747GB/s    Pageable      Device
  Tesla V100-SXM2         1         7  [CUDA memcpy HtoD]
279.27ms  1.7600us           (16 16 1)       (16 16 1)        16        0B        0B         -           -           -           -
  Tesla V100-SXM2         1         7  vector_add(float*, float*, float*, int) [112]
279.28ms  2.9120us                    -               -         -         -         -  20.000KB  6.5500GB/s      Device    Pageable
  Tesla V100-SXM2         1         7  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and
can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

Figure 28. GPU-Trace mode

Similarly, CPU activities can also be profiled using `nvprof --print-api-trace ./vector-add`. As can be seen in Example 2, this will show the trace of CUDA API calls from the CPU.

Example 2. CUDA runtime and driver API calls

```
==84665== Profiling application: ./vector-add
==84665== Profiling result:
   Start  Duration  Name
17.181ms  6.7400us  cuDeviceGetPCIBusId
65.366ms     757ns  cuDeviceGetCount
65.368ms     119ns  cuDeviceGetCount
65.655ms     338ns  cuDeviceGet
65.656ms     237ns  cuDeviceGetAttribute
65.697ms     299ns  cuDeviceGetAttribute
65.708ms     203ns  cuDeviceGetAttribute
65.888ms     336ns  cuDeviceGetCount
65.889ms     128ns  cuDeviceGet
65.891ms  36.109us  cuDeviceGetName
65.928ms  323.78us  cuDeviceTotalMem
66.252ms     218ns  cuDeviceGetAttribute
.
.
66.712ms     169ns  cuDeviceGetUuid
66.721ms  219.40ms  cudaMalloc
286.12ms  4.4720us  cudaMalloc
286.12ms  2.5790us  cudaMalloc
286.16ms  30.096us  cudaMemcpy
286.19ms  17.777us  cudaMemcpy
286.21ms  40.939us  cudaLaunchKernel (vector_add(float*,
                           float*, float*, int) [112])
286.25ms  31.490us  cudaMemcpy
286.49ms  14.906us  cudaFree
286.51ms  4.3960us  cudaFree
```

```
286.51ms  165.13us  cudaFree
```

As mentioned earlier, OpenACC and OpenMP based applications can also be profiled using the `nvprof`. Table 9 shows a few important command line options for profiling OpenACC and OpenMP based applications. CUDA aware MPI applications can also be profiled, for example, using `srun -n $SLURM_NTASKS nvprof ./a.out` [9].

Table 9. Command line options for OpenACC and OpenMP applications

Options	Description
`--openacc-profiling <on|off>`	Turn on/off OpenACC profiling. Default is on.
`--print-openacc-summary`	Print a summary of all recorded OpenACC activities.
`--print-openacc-trace`	Print a detailed trace of all recorded OpenACC activities, including each activity's timestamp and duration.
`--print-openmp-summary`	Print a summary of all recorded OpenMP activities.

5.1.2. Visual Profiler

NVIDIA Visual Profiler is a GUI profiler, which supports CUDA application events and traces. It provides programmers with good insight into understanding the application's compute time, memory, and memory transfer. The below example shows the CUDA code compilation and how to create an input file for the Visual Profiler.

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
.
.
# Compilation
nvcc -arch=sm_70 VecAddtion.cu

# To create a .nvvp file for the visual profiler
nvprof -o input.nvvp ./a.out
```

To open the Visual Profiler, use:

```
nvvp input.nvvp
```

shows the time consumption of all CUDA API calls. Figure 29 shows the timeline of the CUDA API in the application. It shows all the CUDA API calls time consumption for the given application. These calls are especially shown in the total timeline of the application. Figure 29 is not only showing the CUDA API calls but also shows the time consumption for the CUDA kernel calls. The timeline will give a clearer overview of where the application spends most of the time.

[9] Assuming the usage of SLURM as a resource management and scheduling system.

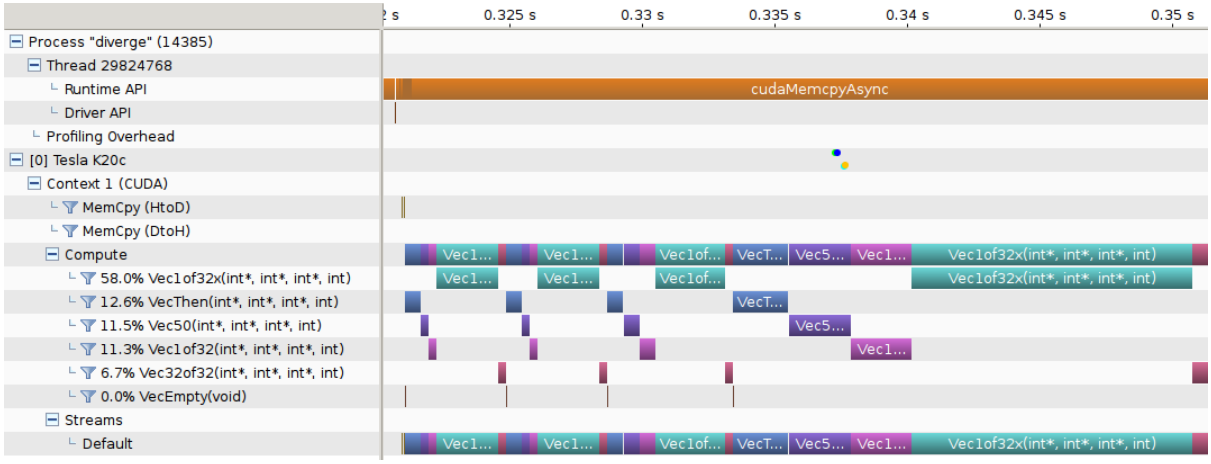

Figure 29. Visual Profiler Timeline

Table 10 lists the important events and metrics supported by the Visual Profiler. So, basically, when we use the visual profiler, we will be able to see the metrics and events under each topic mentioned in Table 10.

Table 10. Important Event and Metrics in the Visual Profiler

Options	Description
GPU Details View	It provides the information about the memory copy and kernel execution
CPU Details View	It gives the detail view of the application spent on the CPU
NVLink View	This will collect the NVLink topology and NVLink transmit/receive metrics
Memory Statistics	It will show the summary of the memory hierarchy of the CUDA programming mode

5.1.3. Nsight Compute and Nsight Systems

The next generation of NVIDIA GPUs will only support Nsight Compute and Nsight Systems [47]. Both of these profiling tools will help to profile the GPUs and CPU-GPU together. Figures 30 and 31 illustrate the difference between the regular NVIDIA profiling and Nsight Compute. Here, during the regular (traditional) profiling, the user will execute the CUDA application; it will communicate with the CUDA runtime library and CUDA driver. But in the Nsight Compute, the CUDA application will be executed by Nsight Compute.

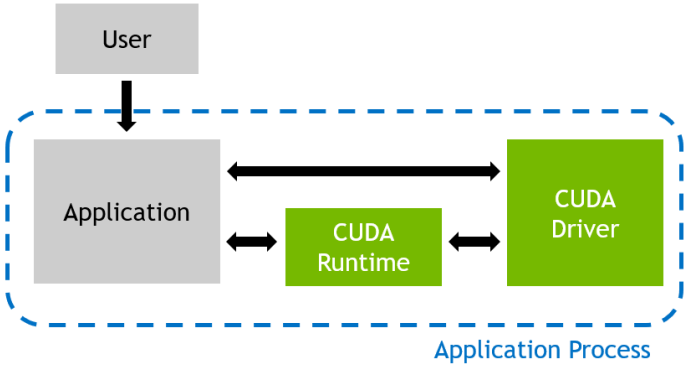

Figure 30. Regular CUDA Profiling [48]

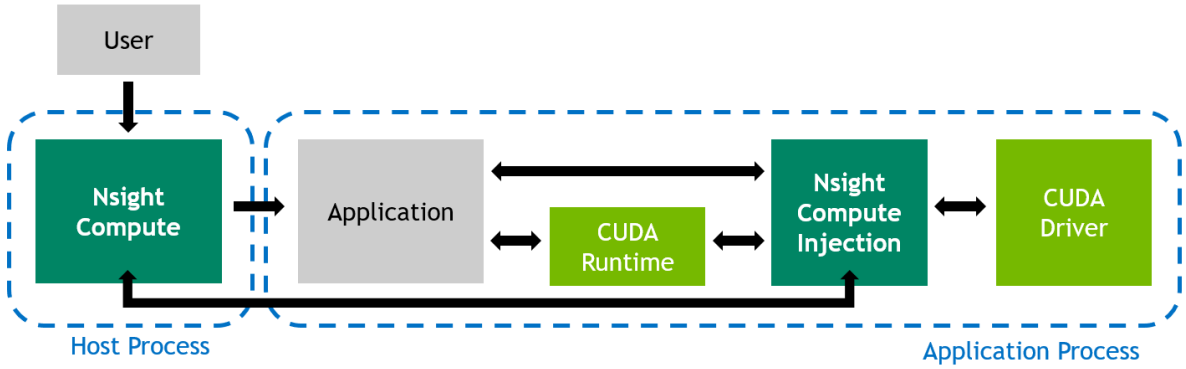

Figure 31. NVIDIA Nsight Compute [48]

NVIDIA Nsight Compute can be invoked by running `ncu` and is the profiling command to collect the default options (events and metrics) in the application, for example, using the command: `ncu -o profile ./a.out`. The profiled result can be seen in the command line temporarily but also using the flag `-o`, with `-o profile` will create `profile.ncu-rep`, which will have all the profiled result. The example below shows the profiling results of two CUDA kernel calls in the application.

```
[Vector addition of 1144477 elements]
==PROF== Connected to process 5268
Copy input data from the host memory to the CUDA device
CUDA kernel launch A with 4471 blocks of 256 threads
==PROF== Profiling "vectorAdd_A" - 1: 0%....50%....100% - 46 passes
CUDA kernel launch B with 4471 blocks of 256 threads
==PROF== Profiling "vectorAdd_B" - 2: 0%....50%....100% - 46 passes
Copy output data from the CUDA device to the host memory
Done
==PROF== Disconnected from process 5268
==PROF== Report: profile.ncu-rep
```

Applications using multiple GPUs on a single node can be also profiled using `ncu--target-processes all -o <single-report-name> <app> <args>.` Nsight Compute can support GUI as well; to invoke GUI use: `ncu-ui <profile.ncu-rep>`. The NVIDIA Nsight compute provides many `metrics`, to see the list of metrics, use:`--query-metrics`. However, sometimes, it would be just easier to collect and analyse the set of metrics. For doing that, Nsight provides `sets` and `sections`. To see the list of sets use: `ncu --list-sets` and to query the current list of sections use: `ncu --list-sections`. By default, the sections are associated with the default set.

Another tool from Nvidia CUDA ToolKit is Nsight Systems, which is used to profile the MPI based CUDA applications. For example, in MPI CUDA applications, we wanted to know the time spending on communication, data transfer and API calls. A command for the Nsight System is `nsys`. To profile the application, use `nsys profile ./a.out`, which will profile the entire application with default events and metrics; `nsys --help` will list out the available options in `nsys`. To know more information about MPI based CUDA profiling, please refer to [67].

5.1.4. PGI compiler (pgprof)

The PGI compiler is the well-known compiler for the OpenACC programming model and suits a wide variety of parallel architectures and programming languages. The main advantage of using the PGI compiler is that it tells you where your code is parallelised or needs to be parallelised at the compile time. By this, you will come to know where to optimise your code further. As of now, the PGI compiler is part of NVIDIA HPC SDK [66], but most of the original PGI functionality will continue to exist. We show a simple example of how to profile the code. Let consider the simple example of vector addition in the C programming language, see below.

```
// VecAdd.c
#include <stdio.h>
```

```
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n )
{
     #pragma acc kernels loop copyin(a[0:n],b[0:n]) copyout(r[0:n])
     for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
 }

 int main( int argc, char* argv[] )
 {
     int n; /* vector length */
     float * a; /* input vector 1 */
     float * b; /* input vector 2 */
     float * r; /* output vector */
     float * e; /* expected output values */
     int i, errs;
     if( argc > 1 )
     n = atoi( argv[1] );
     else n = 100000; /* default vector length */
     if( n <= 0 )
     n = 100000;
     a = (float*)malloc( n*sizeof(float) );
     b = (float*)malloc( n*sizeof(float) );
     r = (float*)malloc( n*sizeof(float) );
     e = (float*)malloc( n*sizeof(float) );
     for( i = 0; i < n; ++i )
     {
          a[i] = (float)(i+1);
          b[i] = (float)(1000*i);
     }
     /* compute on the GPU */
     vecaddgpu( r, a, b, n );
     /* compute on the host to compare */
     for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
     /* compare results */
     errs = 0;
     for( i = 0; i < n; ++i )
     {
       if( r[i] != e[i] )
        {
            ++errs;
        }
    }
    printf( "%d errors found\n", errs );
    return errs;
 }
```

For different programming languages, different compiler flags should be used, i.e.: `pgcc` for C, `pgc++` for C++ and `pgfortran` for Fortran. The below command line illustrates the compilation and the profiling of OpenACC programming model. As we can notice here, line number 7 is parallelisable with 128 threads by using the OpenACC directive. This is a clear indication that the for loop will be executed by the 128 threads in parallel. A user can also test without using the OpeACC directive and see the output, which would tell, for example, a loop is not parallelisable. Basically, these kinds of functionality are very useful from the programmers' perspective that would allow efficient parallelisation of the given code. The PGI compiler will tell us the loop parallelisation option, data movement, loop fusion, data dependencies and loop unrolling. More importantly, when we use the OpenACC directives, we would not know exactly how this will be effective on the given code. In this situation, the PGI compiler will tell more information about if the OpenACC directive is behaving as it is supposed to act. This is the main advantage of using the PGI compiler, as it guides or recommends for more code optimisation even at the compilation stage.

```
# Compilation
pgcc -fast -acc -ta=tesla -Minfo VecAdd.c

#Compilation output
  vecaddgpu:
      6, Generating copyin(a[:n]) [if not already present]
         Generating copyout(r[:n]) [if not already present]
         Generating copyin(b[:n]) [if not already present]
      7, Loop is parallelizable
         Generating Tesla code
         7, #pragma acc loop gang, vector(128)
         /* blockIdx.x threadIdx.x */
main:
     27, Loop not fused: function call before adjacent loop
         Loop not vectorized: data dependency
         Loop unrolled 8 times
     35, Loop not fused: dependence chain to sibling loop
         Loop not vectorized: data dependency
         Generated vector simd code for the loop
         Loop unrolled 8 times
         39, Loop not fused: function call before adjacent loop
```

Finally it offers time consumption of the code, for example, API calls and OpenACC directives. To see the CPU flat profile, use:

```
#Creating the profiling output
pgprof --cpu-profiling-mode flat ./a.out

==120848== PGPROF is profiling process 120848,
command: ./a.out
0 errors found
==120848== Profiling application: ./a.out
==120848== Profiling result:
No kernels were profiled.
API calls:
Time(%) Time     Calls Avg    Min       Max       Name
81.07%  299.09ms 1  299.09ms  299.09ms  299.09ms  cuDevice \
                                                 PrimaryCtxRetain
18.53%  68.365ms 1  68.365ms  68.365ms  68.365ms  cuDevice \
                                                 PrimaryCtxRelease
 0.28%  1.0285ms 1  1.0285ms  1.0285ms  1.0285ms  cuMemAllocHost
 0.07%  274.06us 1  274.06us  274.06us  274.06us  cuMemAlloc
 0.03%  105.24us 1  105.24us  105.24us  105.24us  cuModuleLoadDataEx
 0.01%  22.153us 1  22.153us  22.153us  22.153us  cuStreamCreate
 0.00%  8.0830us 1  8.0830us  8.0830us  8.0830us  cuDeviceGetPCIBusId
 0.00%  6.1490us 3  2.0490us     359ns  5.2780us  cuPointer \
                                                 GetAttributes
 0.00%  3.8520us 1  3.8520us  3.8520us  3.8520us  cuStreamSynchronize
 0.00%  2.0950us 1  2.0950us  2.0950us  2.0950us  cuModuleGetFunction
 0.00%  1.5520us 3     517ns     124ns  1.1720us  cuDeviceGetCount
 0.00%  1.4830us 4     370ns     266ns     577ns  cuDeviceGetAttribute
 0.00%  1.4630us 3     487ns     203ns     723ns  cuCtxSetCurrent
 0.00%     581ns 2     290ns     174ns     407ns  cuDeviceGet
 0.00%     220ns 1     220ns     220ns     220ns  cuDevice \
                                                 ComputeCapability
 0.00%     210ns 1     210ns     210ns     210ns  cuDriverGetVersion
 0.00%     205ns 1     205ns     205ns     205ns  cuCtxGetCurrent
 OpenACC (excl):
```

```
59.51%  139.39us 1  139.39us  139.39us  139.39us  acc_device_ \
                                                   init@VecAdd.c:6
28.78%  67.399us 1  67.399us  67.399us  67.399us  acc_enter_ \
                                                   data@VecAdd.c:6
 5.40%  12.655us 1  12.655us  12.655us  12.655us  acc_exit_ \
                                                   data@VecAdd.c:6
 3.93%  9.2150us 1  9.2150us  9.2150us  9.2150us  acc_wait@VecAdd.c:6
 2.37%  5.5620us 1  5.5620us  5.5620us  5.5620us  acc_compute_const \
                                                   ruct@VecAdd.c:6

======== CPU profiling result (flat):
Time(%)      Time  Name
 71.54%  996.35ms  ???
 16.92%   235.7ms  cuDevicePrimaryCtxRetain
  6.15%  85.708ms  cuInit
  5.38%  74.994ms  cuDevicePrimaryCtxRelease

======== Data collected at 100Hz frequency
```

For profiling default events and traces, the following should be considered:

```
# To profile default events and traces use:
pgprof -o a.prof ./a.out

# To open the a.prof, use:
pgprof -i a.prof
```

5.2. Tuning for NVIDIA GPUs

This section shows how to tune and optimise a CUDA and OpenACC code on the NVIDIA GPUs. CUDA and OpenACC are the most popular programming environments for NVIDIA GPUs. The following section shows the multiple options for how the CUDA and OpenACC code can be accelerated on the NVIDIA GPUs.

5.2.1. CUDA Occupancy Calculator

CUDA occupancy is defined as a ratio of the active warps to a maximum amount of warps that can be supported by the given NVIDIA architecture. Each NVIDIA architecture has a number of registers `N`, and these are shared among the thread blocks. The CUDA compiler tries to minimise the number of registers per thread blocks to maximise the CUDA occupancy. However, these thread blocks and register usage differ between different compute capabilities and NVIDIA architectures. To make these combinations (registers and thread blocks) easy, NVIDIA provides the pre-prepared simple calculation of the `CUDA Occupancy Calculator[XLS]` sheet to the users. With this, users can choose an optimised number of thread blocks depending on the compute capability and NVIDIA architecture in the CUDA kernel [49] without trying many combinations of thread blocks by themselves.

5.2.2. Shared Memory Programming

Shared memory is an on-chip memory, with higher bandwidth and lower latency than global memory. Each streaming multiprocessors (SMs) has its own shared memory that is shared by the threads within the thread block. Usage of shared memory might increase the overall performance of the CUDA application. All NVIDIA GPU architectures have the shared memory architecture concept. Figure 32 shows a shared memory architecture in `V100` NVIDIA GPU.

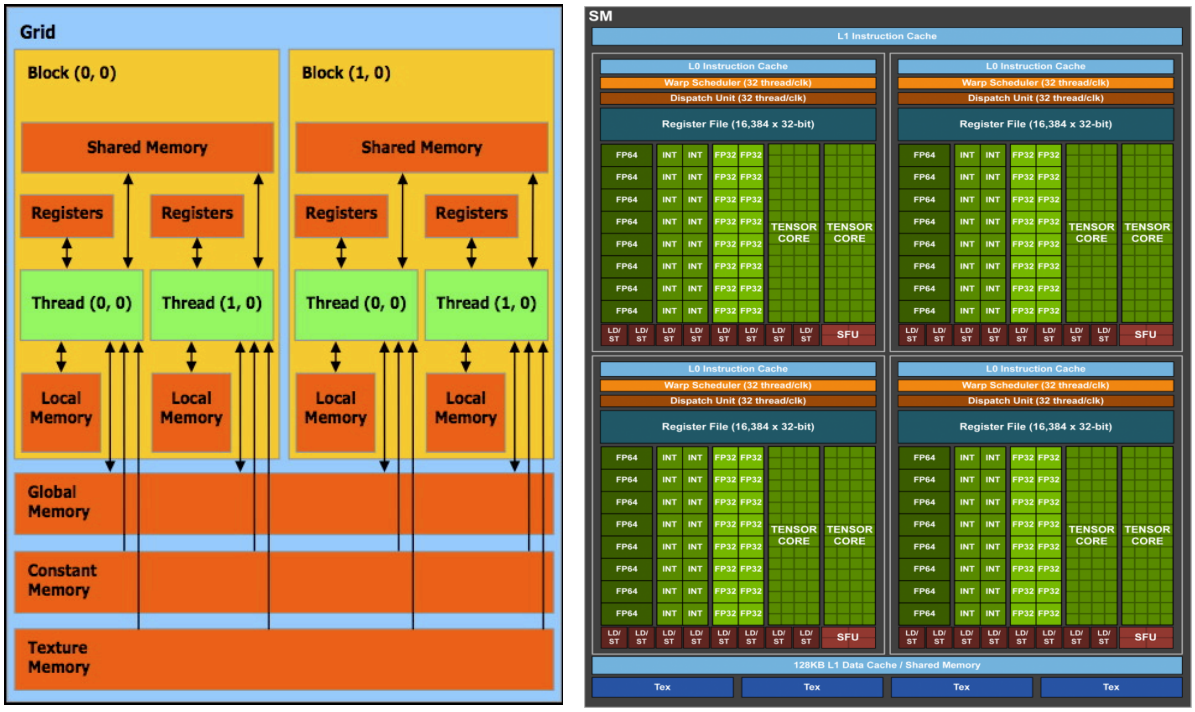

Figure 32. NVIDIA GPUs shared memory

Linear algebra routines are important for science and engineering problems. Example 3 shows the dot product utilising the shared memory. Here, the kernel can be executed using `<<<ceil((N+THREADS_PER_BLOCK)/THREADS_PER_BLOCK), THREADS_PER_BLOCK>>>`, where `N` is a number of elements in the vector. It is worth mentioning that the `THREADS_PER_BLOCK` should be chosen based on the NVIDIA GPU architecture. For example, it can be 256 and 512. Here, the multiplication of the two vectors will be stored in the shared memory, and threads per block will have a partial sum of the vector product. Finally, this partial sum will be added one by one. For the matrix multiplication, please refer to [15].

Example 3. Dot product kernel function in shared memory programming

```
__global__ void dot(float*a, float*b, float*res)
{
  // Allocate the shared memory for temp. dot product
  __shared__ float products[THREADS_PER_BLOCK];

  int id = blockDim.x* blockIdx.x+ threadIdx.x;

  // Shread memory shared by threads in the same block
  products[threadIdx.x] = a[id] * b[id];

  // Synchronize the threads
  __syncthreads();

  if(threadIdx.x == 0)
    {
      int sum= 0;
      for(int i=0; i<THREADS_PER_BLOCK; i++)
        {
          sum+=products[i];
        }
      // To avoid the reace condition
      atomicAdd(res, sum);
    }
```

```
}
```

5.2.3. Register Usage

Minimising the register usage helps to launch a maximum number of warps in the streaming multiprocessor. This is usually achieved in two ways: using the flag `-maxrregcount=<N>` at the compile time and setting the `__launch_bounds__` [68] along with the kernel function. Here, `N` is the number of registers that can be allocated per warps. Example 4 shows how to control the register usage with `__launch_bounds__`.

Example 4. Register usage tuning

```
// Option 1
nvcc -arch=sm_70 vector-add.cu -o vector-add -maxrregcount=16

// Option 2
#define THREADS_PER_BLOCK          256
#if __CUDA_ARCH__ >= 200
    #define MY_KERNEL_MAX_THREADS  (2 * THREADS_PER_BLOCK)
    #define MY_KERNEL_MIN_BLOCKS   3
#else
    #define MY_KERNEL_MAX_THREADS  THREADS_PER_BLOCK
    #define MY_KERNEL_MIN_BLOCKS   2
    #endif

// Device code
__global__ void
__launch_bounds__(MY_KERNEL_MAX_THREADS, MY_KERNEL_MIN_BLOCKS)
MyKernel(...)
{
    ...
}
```

This will not guarantee that the application will run really faster. It is obvious that using minimum registers will accommodate more warps to be executed on the streaming multiprocessors. At the same time, spilled registers will be cached in L1 or L2. This will slow down the warps to finish the work than before. So, in order to achieve a good performance on this register usage, the programmer should know the register count and try a few combinations (register count and thread blocks) before finding a good register count and threads combination.

5.2.4. Cache Configuration

Depending on the CUDA application, L1 cache and shared memory sizes can be modified by using CUDA runtime `cudaDeviceSetCacheConfig()` API call. Registers hold the frequently used values in the computation, but when the registers are more than the actual available count, it spills into different cache levels in the memory. This kind of behaviour will slow down the performance of an application. NVIDIA GPUs have an L1 cache and shared memory, so increasing the L1 cache size will accommodate the spilled registers and not let them go further into shared memory or global memory. Depending on the application, it might not be needed to use all the registers, but to use the shared memory more. In that case, shared memory size can be increased, and the L1 cache size can be decreased. The list below shows the example usage of `cudaDeviceSetCacheConfig()` in CUDA programming environment [68].

```
// example usage
cudaDeviceSetCacheConfig(Option);

// Default function cache configuration, no preference
// Option: cudaFuncCachePreferNone
// Prefer larger shared memory and smaller L1 cache
// Option: cudaFuncCachePreferShared
// Prefer larger L1 cache and smaller shared memory
// Option: cudaFuncCachePreferL1
```

```
// simple example usage increasing more shared memory
#include<stdio.h>
int main()
{
    // example of increasing the shared memory
    cudaDeviceSetCacheConfig(cudaFuncCachePreferShared);
    My_Kernel<<<>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

5.2.5. Memory Access Through NVLink Interconnect

NVLink from NVIDIA is a high-speed data interconnect. Using NVLink, a data transfer between the GPUs within the single compute node can be increased compared to using a data transfer via CPU. NVLink completely bypasses the CPU for the data transfer between GPUs. For example, the GV100 can support up to six NVLink interconnections. Each has up to 50 GB/s of bi-directional bandwidth. NVIDIA Volta can similarly support up to six NVLink with a total bandwidth of 300 GB/s. Figure 33 shows the simple schematic overview of NVLink interconnects between NVIDIA Ampere architecture GPUs. According to NVIDIA, Ampere can support up to 600 GB/s [64] when it is combined with the latest generation of NVIDIA NVSwitch [65].

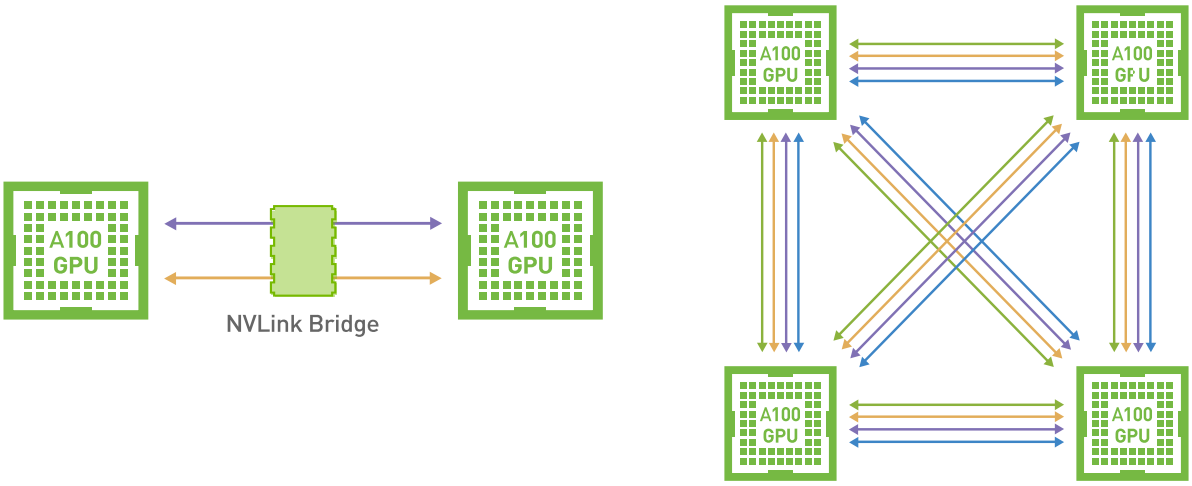

Figure 33. NVLink GPU-to-GPU connections [50]

The NVLink capability should be enabled via `cudaDeviceEnablePeerAccess()` and `cudaMemcpyPeer()` API calls. Example 5 shows how to enable `Peer-to-Peer Memory Access` using `DeviceEnablePeerAccess()`, which can directly access the `vec` allocated memory from `Device 0` in `Device 1`.

Example 5. NVLink peer-to-peer memory access

```
cudaSetDevice(0);                       // Set device 0 as current
float* vec;
size_t size = 1024 * sizeof(float);
cudaMalloc(&vec, size);                 // Allocate memory on device 0
MyKernel<<<128, 128>>>(vec);            // Launch kernel on device 0
cudaSetDevice(1);                       // Set device 1 as currentcuda
DeviceEnablePeerAccess(0, 0);           // Enable peer-to-peer access
                                        // with device 0
// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address vec
MyKernel<<<128, 128>>>(vec);
```

Example 6 illustrates the demo of the peer-to-peer memory copy using `cudaMemcpyPeer()`. Here, an allocated memory on the `Device 0` can be copied into `Device 1`.

Example 6. NVLink peer-to-peer memory copy

```
cudaSetDevice(0);                       // Set device 0 as current
float *vec0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&vec0, size);                // Allocate memory on device 0
cudaSetDevice(1);                       // Set device 1 as current
float *vec1;
cudaMalloc(&vec1, size);                // Allocate memory on device 1
cudaSetDevice(0);                       // Set device 0 as current
MyKernel;<<128, 128;>>(vec0);           // Launch kernel on device 0
cudaSetDevice(1);                       // Set device 1 as current
cudaMemcpyPeer(vec1, 1, vec0, 0, size); // Copy veco to vec1
```

5.2.6. NVIDIA CUDA Streams

The CUDA calls are either `synchronous` or `asynchronous`. That is, either they enqueue work and wait for the completion or enqueue work and return immediately. However, host and kernel calls are asynchronous, and they overlap between them. But usage of `CUDA streams` allows to make the kernel asynchronous calls safely; thus make the calls be executed concurrently. More precisely, the usage of CUDA streams allows one to have full control over the calls that can be executed concurrently. This kind of operation is quite helpful when computation and data transfer are happening at the same time in the application. The `CUDA streams` in the CUDA programming environment can be created and destroyed as follows:

```
// initialise the cuda stream
cudaStream_t stream;
// create the cuda stream
cudaStreamCreate(&stream);
// destroy the cuda stream
cudaStreamDestroy(stream);
```

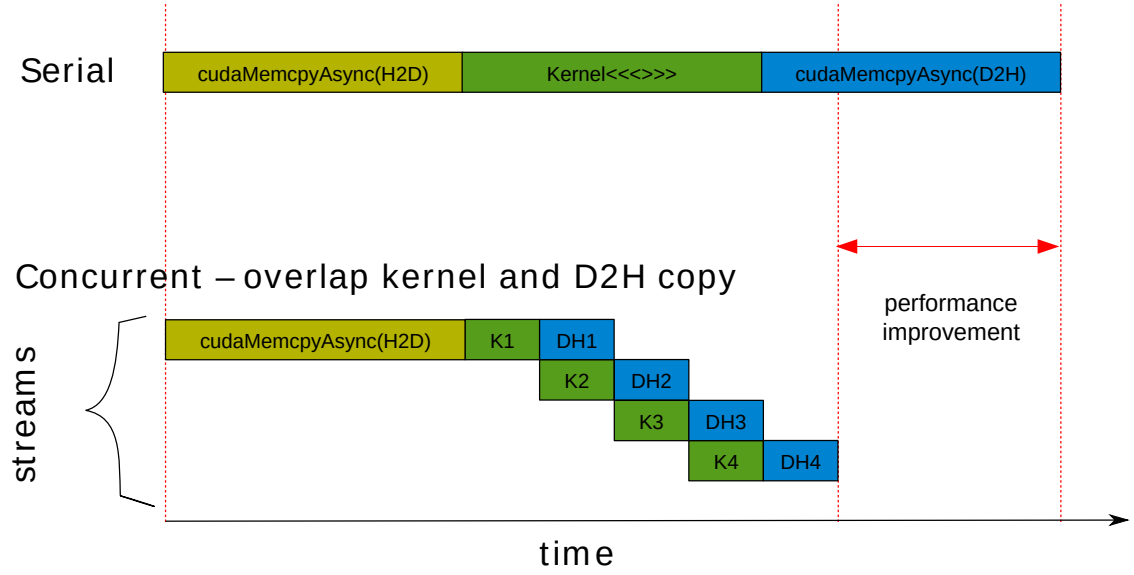

Figure 34. NVIDIA CUDA Streams

The CUDA streams should be passed as an actual argument in the kernel call from the host, for example:

```
MyKernel<<<126, 512, 0, stream>>>();
```

Figure 34 shows a simple schematic example of `CUDA streams` performing computation and data transfer in parallel. As can be seen, the usage of `CUDA streams` results in improved performance. The following code listing shows a simple example of using the `CUDA streams` for computation and communication in a CUDA application. In this example, `stream0` and `stream1` can be executed concurrently. At the same time, `stream0` calls are synchronised. That is, the same id of the CUDA streams will always be synchronised. To see this operation visually, please refer to Figure 35.

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
halo_computation<<<blocks,threads,0,stream0>>>();
computation<<<blocks,threads,0,stream1>>>();
cudaMemcpyAsync(img, d_img, size, cudaMemcpyDeviceToHost, stream0);
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

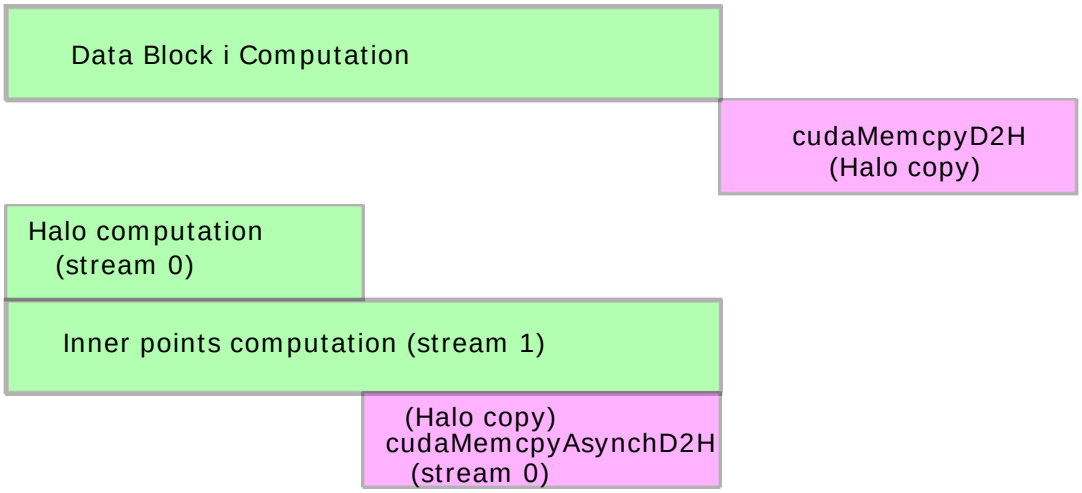

Figure 35. Example of CUDA streams in the application

5.2.7. NVIDIA GPUDirect

NVIDIA GPUDirect is a part of `Magnum_IO` [51] technology. It will enable GPUs to access the memory of another GPU across the multiple compute nodes and access the memory directly from the external storage without using the host CPU memory. It has two functionality: GPUDirect Storage [52] and GPUDirect Remote Direct Memory Access (RDMA) [53]. Usage of this functionality will eliminate unnecessary memory copy, decrease CPU overheads, and reduce latency. `GPUDirect Storage` can be used to transfer data from external storage to GPUs. `GPUDirect RDMA` enables faster data transfer between the GPUs across different compute nodes. Figures 36 and 37 show the simple illustration of these two methodologies.

This methodology is quite useful when we run the simulation on multiple compute nodes with GPUs on each node. All MPI libraries provide multiple node computation for the distributed memory architecture, for example, Intel MPI [60], MPICH [59], MVAPICH2 [58], OpenMPI [57] and MPICH [57]. At the same time, these MPI libraries also support the CUDA-Aware MPI; this means this will enable GPUDirect communication. There are two things you should be checking if your system can support the GPUDirect:(a) your system should have all the necessary drivers installed and uses the Mellanox InfiniBand network connectivity [63], and (b) your MPI library should support CUDA-Aware MPI.

For example, let us assume your system uses the OpenMPI MPI library, and you want to test if your system can support the CUDA-Aware MPI, then execute the following command:

```
// login to GPU compute node
// load or enable the MPI with CUDA compiler (CUDA Aware MPI)
```

```
// for example, module load mpi/OpenMPI/3.1.4-gcccuda-2019b
$ ompi_info --all | grep btl_openib_have_cuda_gdr
       MCA btl openib: informational "btl_openib_have_cuda_gdr"
       (current value: "true", data source: default,
       level: 5 tuner/detail, type: bool)
```

If you get a similar output as shown above, that means your MPI library and your GPU machine support the GPUDirect communication. Furthermore, if you want to test at the run time, please refer to CUDA-Aware MPI support [57]. In case, if your system has MPICH or Cray compilers, then refer to [61] to check if the system can support the GPUDirect. NVIDIA is also providing a few of the benchmark test cases (using MVAPICH) to support GPUDirect technology, to see the results, please refer to [62]. More information regarding NVIDIA GPUDirect can be found in [54].

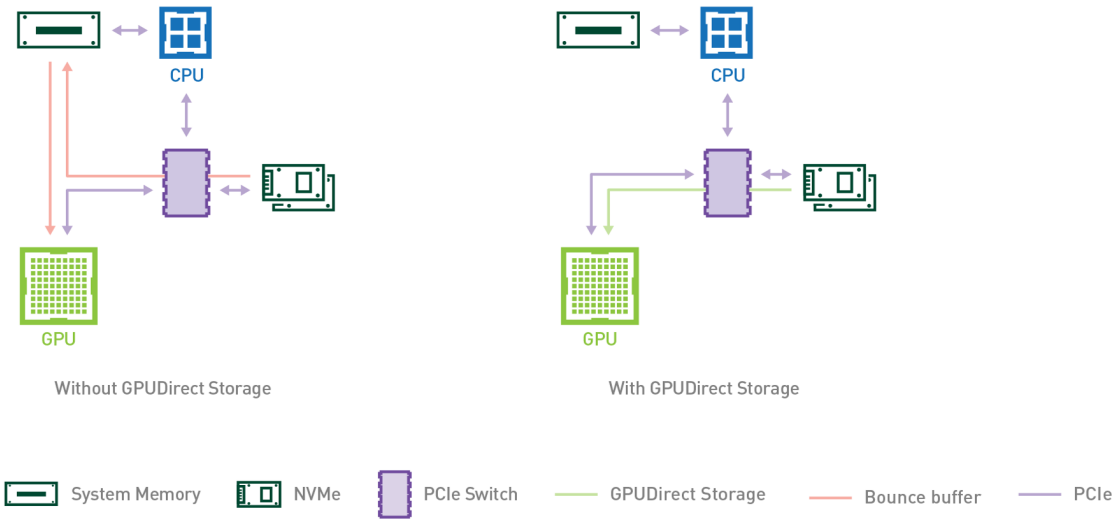

Figure 36. NVIDIA GPUDirect Storage

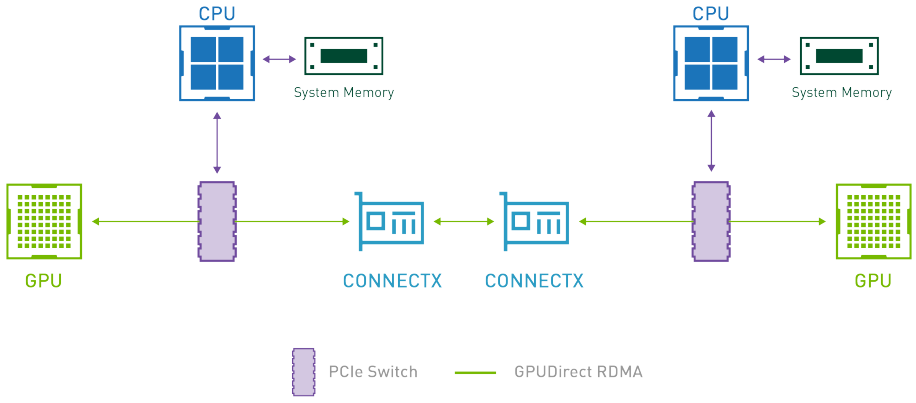

Figure 37. NVIDIA GPUDirect RDMA

5.2.8. Thread Blocks in OpenACC Programming Model

In native CUDA programming, users have to specify the thread blocks in the kernel, whereas in OpenACC the compiler sets the thread blocks to a default value. However, sometimes it is of a big use to control the thread blocks. During the OpenACC profiling, one can notice (see Section 5.1.4, "PGI compiler (pgprof)") `gang and vector` specifications and these parameters can be changed manually. For instance, if we set `vector(128)`, it means the number of threads per block is 128. By default, the compiler chooses the default threads per block (that is, the number of threads), but all of these gang, warps, and workers can be changed manually in the program. Depending on the given problem and architecture, it is advisable to change the values by the programmer. Because, the compute constructs from the OpenACC choose the different threads blocks for the same application. For example, `parallel` will give `vector(128)`, whereas `kernels` will give `vector(32)`. To change the number of threads in C programming language on the `parallel construct`, the following can be used: `#pragma acc <directive> num_gangs(24) vector_length(256)`. For changing the number of threads and blocks for the OpenACC compute constructs, see below example.

```
// C/C++ OpenACC programming model
#pragma acc kernels // Compiler chooses the thread blocks
for( int i = 0; i < n; i++ )
     y[i] +=a*x[i];

// For kernels, to set gang (thread blocks) and vector (threads per block)
#pragma acc kernels loop gang(24) vector(128)
for( int i = 0; i < n; i++ )
     y[i] +=a*x[i];

// For parallel, To set gang (thread blocks) and vector (threads per block)
#pragma acc parallel num_gangs(100) vector_length(128)
{
   #pragma acc loop
   for( int i = 0; i < n; i++ )
        y[i] +=a*x[i];
}
```

Table 11. CLAUSES for parallel and kernels

	OpenACC Kernels	**OpenACC Parallel**
Threads	vector(expression)	vector_length(expression)
Warps	worker(expression)	num_workers(expression)
Thread Blocks	gang(expression)	num_gangs(expression)
Device (NVIDIA or radeon)	device_type(device name)	device_type(device name)

Table 11 shows how the threads, warps, thread blocks, and target device can be defined in OpenACC for `parallel` and `kernels` constructs. To know more details on the available profiling options, please use: `pgprof -help`.

5.3. NEC SX-Aurora Performance Tuning

Most of the existing performance tools for x86-machines or GPU-based accelerators are not suitable for the SX-Aurora and therefore NEC offers custom solutions. In this section, we review optimisations for the High Performance Conjugate Gradients (HPCG) Benchmark [89] using three different performance analysis tools developed by NEC: PROGINF, FTRACE and veperf. More background information on the HPCG benchmark and it's relevance in HPC is given in Section 7.1. Third party tools, such as PAPI and TAU, started providing support for the SX-Aurora vector engines only recently [100]. PAPI is also used by Vftrace, another profiler provided by NEC [101].

When not further specified the compilation flags are `-O2 -finline-functions -finline` . For this demonstration we chose `-O2` instead of `-O4` because it makes distinguishing manual and compiler-driven loop optimisations easier. The problem size for conducted performance measurements is 104×104×104, which is default

for the HPCG reference implementation. All examples are computed on the prototype machine at HLRS. The technical details of the system are presented in Section 6.5

5.3.1. PROGINF

PROGINF [98] generates a report of several performance metrics after a program is finished execution. Enabling the feature requires using the flag `-proginf` for compilation and linking. Additionally, the environment variable `VE_PROGINF` has to be set to `YES` for a regular report or `DETAIL` for verbose output. For more detailed metrics with regards to memory access, `VE_PERF_MODE` can be set to `VECTOR-MEM`.

The output shown in Example 7 was generated using PROGINF in combination with the HPCG reference code on a single core. Initially, we focus on the two metrics: `V.Op.Ratio` and `A.V.Length`. `V.Op.Ratio` describes the percentage of operations that are vector operations. Using the scalar pipeline of the SX-Auorora regularly will inevitably result in poor performance. Thus, using SX-Aurora TSUBASA efficiently requires that `V.Op.Ratio` is close to 100%. `A.V.Length` is the average vector length and gives information on the actual throughput of the performed vector operations. When a vectorised loops spans 50 iterations, the vector instructions can only comprise as many elements at once. To exhaust SX-Aurora's full potential, the vector length should be as large as possible, which is 256. For the reference implementation of the HPCG benchmark, both goals are clearly missed, as only 86.1% of all instructions are vector-instructions and even those are on average only of length 38.8. As a result, the performance is very poor (676 MFLOPS).

Example 7. PROGINF output for HPCG reference code

```
          ********  Program  Information  ********
  Real Time (sec)                 :          232.056651
  User Time (sec)                 :          231.893382
  Vector Time (sec)               :          200.821410
  Inst. Count                     :       125780651761
  V. Inst. Count                  :        16086865419
  V. Element Count                :       624815233044
  V. Load Element Count           :       143775692015
  FLOP Count                      :       157072712448
  MOPS                            :         3404.928663
  MOPS (Real)                     :         3402.412784
  MFLOPS                          :          677.381983
  MFLOPS (Real)                   :          676.881470
  A. V. Length                    :           38.840086
  V. Op. Ratio (%)                :           86.106652
  L1 Cache Miss (sec)             :            4.139614
  VLD LLC Hit Element Ratio (%)   :           38.855712
  Memory Size Used (MB)           :         1296.000000

  Start Time (date)   :   Thu Jan  7 09:08:10 2021 Europe
  End   Time (date)   :   Thu Jan  7 09:12:02 2021 Europe
```

Other notable metrics generated by PROGINF are `VLD LLC Hit Element Ratio`, that indicates the quality of cache usage, and `Memory Size Used`, that reports how much of the 40 GB HBM2 memory is allocated. Using the environment variable `NMPI_PROGINF`, PROGINF can generate MPI-aware reports that show the accumulated, minimal, maximal and averaged metrics of all MPI processes. It can also output each individual rank's information depending on the value of `NMPI_PROGINF`, that can be `YES`, `ALL`, `DETAIL` or `ALL_DETAIL`. PROGINF itself does not generate significant execution overhead and therefore gives reliable information on the program as a whole. Since it sees the program as a whole, setup procedures,which are likely not as optimised as the compute kernels, are included in the accumulated results. The next section will introduce FTRACE, which gives more distinct information on individual functions.

5.3.2. FTRACE

FTRACE [98] produces a profiling based on code injection to associate the known performance metrics to individual functions. It works for all compilers supported by the NEC SX-Aurora TSUBASA. To utilise FTRACE,

the developer has to specify `-ftrace` as compile and link flag. Executing the binary will generate one `ftrace.out.0.#` file for each MPI rank. The data can be accessed using the ftrace binary (`/opt/nec/ve/bin/ftrace`).

```
mpirun -np 8 ./xhpcg
ftrace -num 3 -f ftrace.out.0*
```

generates a report for the three most time-consuming functions of the HPCG reference implementation. By default, FTRACE will focus on metrics concerning vector operations. When `VE_PERF_MODE` is set to `VECTOR-MEM`, the focus shifts towards metrics describing memory and cache behaviour. For the reference implementation of the HPCG benchmark, FTRACE prints the output shown in Example 8. Frequency and exclusive time are total measurements of all 8 MPI ranks. The other metrics are averages.

Example 8. FTRACE output for HPCG reference implementation

```
FREQUENCY  EXCLUSIVE        MFLOPS  V.OP   AVER.  VLD LLC PROC.NAME
           TIME[sec](  % )          RATIO  V.LEN  HIT E.%

     9296  2487.099( 59.0)   230.7  78.27  28.0     53.08 ComputeSymGS_ref
     5560  1199.237( 28.4)   232.4  80.43  26.7     68.64 ComputeSpMV_ref
       32   116.657(  2.8)     0.1   4.61   8.3    100.00 GenerateProblem_ref
------------------------------------------------------------------------------
738243981  4218.369(100.0)   206.8  68.57  27.7     57.14 total
```

Example 9. Sparse matrix-vector multiplication with CSR format

```
for(int i = 0; i < nrow; i++){
  double sum = 0.0;
  for (int j=0; j < A.nonzerosInRow[i]; j++){
    sum += A.matrixValues[i][j]*xv[A.mtxIndL[i][j]];
  }
  yv[i] = sum;
}
```

Example 10. Reference implementation of symmetric Gauss Seidel method (forward sweep)

```
for (local_int_t i=0; i < nrow; i++) {
  double sum = rv[i];
  for (int j=0; j < A.nonzerosInRow[i]; j++) {
    sum -= A.matrixValues[i][j] * xv[A.mtxIndL[i][j]];
  }
  sum += xv[i]*A.matrixDiagonal[i][0];
  xv[i] = sum/A.matrixDiagonal[i][0];
}
```

The FTRACE report in Example 8 shows, that symmetric Gauss Seidel (`SymGS`) requires 59.0% and sparse matrix-vector multiplication (`SpMV`) 28.4% of the total execution time. The source of the poor performance is the unfavorable sparse matrix format used by the reference implementation of `SymGS` and `SpMV` (Example 9 and 10). The compressed sparse rows (CSR) format stores all non zero indices in row-major order. It is not suitable in this case because each row has a variable number of non-zero entries but never more than 27, making the maximum vector length 27. This assumption is supported by the measured vector-lengths of the two functions, which are 28.0 and 26.7 on average. Like GPUs, the NEC SX-Aurora benefits from uniform memory access patterns and large leading dimensions. By using the ELLPACK sparse matrix format in column-major order, one achieves both desired properties. The 2D-array becomes uniform by padding rows that have fewer non-zeroes entries with explicit zeros. For some matrices, this optimisation introduces massive overhead but in case of the HPCG-matrix the required padding is negligible, as the majority of rows have 27 entries. The different matrix formats are visualised in Figure 38. Using the ELLPACK format, the possible vector length is limited by the number of equations, which is arbitrarily high for the HPCG benchmark. The vector length for `SpMV` can therefore be stretched closely to 256 with the code shown in Example 11.

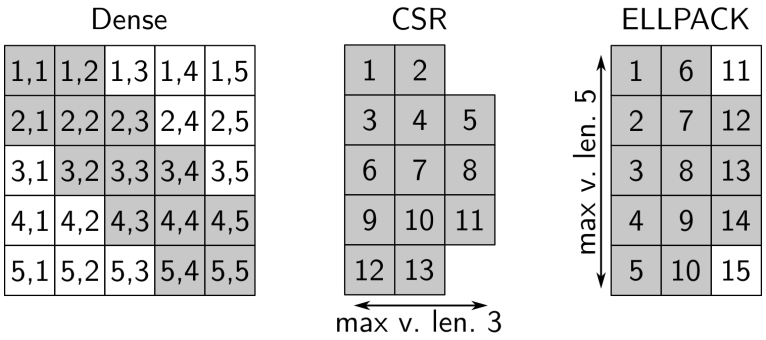

Figure 38. Sparse matrix formats and their vector lengths in sparse matrix-vector multiplication (greyed out entries are non-zeroes)

Example 11. Sparse matrix-vector multiplication with ELLPACK format in column-major

```
for(int i = 0; i < nrow; i++){
  yv[i] = 0.0;
}
for (int j=0; j < nmax; j++){
  for(int i = 0; i < nrow; i++){
    yv[i] += A.matrixValues[j][i]*xv[A.mtxIndL[j][i]];
  }
}
```

Transforming the matrix from row-major to column-major requires modification of the symmetric Gauss Seidel smoother. The reference implementation shown in Example 10 allows vectorisation of the innermost loop, because there are no loop dependencies and the data accesses are contiguous for `j`. In the column-major traversal, however, the loop contains a read-after-write dependency that theoretically only allows sequential execution. SX-Aurora executes loop bodies in chunks of 256 iterations at the same time and therefore `xv` might be read from while the updated value is not yet computed. Only if the developer is certain that this cannot happen, the loop can be vectorised using `#pragma _NEC ivdep`. Reordering matrix and vectors with respect to a multicolouring of the respective graph ensures this condition so that the vectorisable implementation can be programmed as shown in Example 12. The standard approach for GPUs is to colour the graph with as few colours as possible, which in case of the HPCG benchmark is 8 colours. `#pragma _NEC ivdep` can be used to signal to the compiler that a loop certainly does not contain any loop dependencies which makes it easier for the compiler to vectorise complex loops. In this case it is not strictly required, because splitting the `i`-loops into three separate loops solves the dependency. `#pragma _NEC novector` makes sure that the compiler does not vectorise the following loop. We can place it in front of the `j`-loop, although it is not necessary in this instance either, because the compiler knows that vectorising this loop is undesirable.

Example 12. ELLPACK and multicoloured version of symmetric Gauss Seidel method (forward part)

```
for(int k = 0; k < numColours; k++){
  const int lower = accumulatedColourSizes[k];
  const int upper = accumulatedColourSizes[k];

  for (local_int_t i=lower; i < upper; i++)
    sum[i] = rv[i];

  #pragma _NEC novector
  for(local_int_t j = 0; j < nmax; j++)
  #pragma _NEC ivdep
    for (local_int_t i=lower; i < upper; i++)
      sum[i] -= A.matrixValues[j][i] * xv[A.mtxIndL[j][i]];

  for (local_int_t i = lower; i < upper; i++){
    sum[i] += xv[i]*A.matrixDiagonal[i][0];
    xv[i] = sum[i]/A.matrixDiagonal[i][0];
  }
}
```

Example 13. Truncated FTRACE output for ELLPACK-optimised HPCG implementation

```
FREQUENCY  EXCLUSIVE        MFLOPS  V.OP  AVER.  VLD LLC PROC.NAME
           TIME[sec](  % )          RATIO V.LEN  HIT E.%

    29456   366.075( 15.8)  4369.3  99.31 255.3    10.70 ComputeSymGS
    17120   181.035(  7.8)  3119.0  99.26 256.0     3.63 ComputeSpMV
------------------------------------------------------------------------------
769304638  2324.212(100.0)  1089.2  92.01 140.1    12.31 total
```

The effectiveness of the optimisation becomes clear when considering the updated FTRACE output shown in Example 13. While the reference implementation has a sustained performance of less than 240 MFLOPS per rank for `SpMV` and `SymGS`, the ELLPACK-optimised version achieves 4.37 GFLOPS and 3.12 GFLOPS. The reason for this massive acceleration can be found in the same FTRACE output. Compared to the reference implementation of `SpMV`, whose vector length is on average 26.7, the vectors in the optimised version are as large as possible (256). Moreover, the ratio of vector operations to scalar operations increased from approximately 80% to more than 99%. Merely the LLC hit ratio is much lower: the new `SpMV` implementation achieves only 3.63% cache efficiency.

To optimise the cache efficiency for `SpMV`, we first consider the effects of the previous modifications with respect to caching behaviour. Every element of the matrix is accessed only once per invocation. Hence, the matrix itself is not cacheable and the difference must be the data access pattern to the vectors. With the default problem size of 104×104×104, each vector needs approximately 9 MB which is too large for the last level cache (2 MB per core). `yv[0]` will already be evicted from cache by the time `yv[nrow-1]` is reached. The new loop order results in a stride-one access to `yv` within the inner loop instead of the outer loop. Thus, `yv` is always accessed from HBM instead of LLC. Loop-blocking, as it is shown in Example 14, can be used to prevent this caching-behaviour. For 104×104×104 we choose a blocksize of 2197 which amounts to a cache-load of approximately 500 kb for matrix and vectors. Example 15 shows that this optimisation increases the LLC hit ratio for `SpMV` from 3.63% to 43.10% and consequently achieves more than twice as many FLOPS. The initial level of cache efficiency that the CSR-based implementation recorded cannot be reached because colouring and reordering the data structures inevitably leads to scattered accesses to `xv`.

Example 14. Sparse matrix-vector multiplication with ELLPACK format and loop blocking

```
for(int ii = 0; ii < nrow; ii+=BLOCKSIZE){
  for (int i=ii; i < ii+BLOCKSIZE; i++){
    yv[i] = 0.0;
  }
  for (int j=0; j < nmax; j++){
    for (int i=ii; i < ii+BLOCKSIZE; i++){
      yv[i] += A.matrixValues[j][i]*xv[A.mtxIndL[j][i]];
    }
  }
}
```

Example 15. Truncated FTRACE output for optimised HPCG implementation using loop-blocked `SpMV`

```
FREQUENCY  EXCLUSIVE        MFLOPS  V.OP  AVER.  VLD LLC PROC.NAME
           TIME[sec](  % )          RATIO V.LEN  HIT E.%
    36736   456.672( 19.6)  4368.1  99.31 255.3    10.83 ComputeSymGS
    21296    94.400(  4.0)  7424.2  99.15 244.1    43.10 ComputeSpMV
```

Although the same optimisation technique could be applied to the `SymGS` kernel, another method is more favorable in this case. When considering the implementation shown in Example 12, one may notice that the multi-colouring approach is inherently a loop-blocking scheme. For this reason, `SymGS` had a better caching behaviour and achieved three times more LLC hits. The remaining issue is that there are too many rows per colour so that the blocks are too large. An easy solution is to increase the number of colours artificially, such that each colour comprises the desired 2197 rows. A convenient side effect of this optimisation is improved numerical behaviour and therewith faster convergence of the multigrid solver. The updated measurements for `SpMV` and `SymGS` are shown in Example 16. On average, each core achieves 11.77 GFLOPS for `SymGS` and 9.41 GFLOPS for `SpMV`.

Example 16. Truncated FTRACE output for optimised HPCG implementation using artificially generated colours

```
FREQUENCY  EXCLUSIVE        MFLOPS  V.OP  AVER.  VLD LLC PROC.NAME
           TIME[sec](  % )          RATIO V.LEN  HIT E.%
    59696   275.426( 12.5) 11769.3  99.15 244.0    48.72 ComputeSymGS
    34496   120.375(  5.5)  9405.6  99.07 244.1    48.14 ComputeSpMV
```

Apart from the command line interface, FTRACE also comes with a GUI that visualises performance data. This is especially useful for finding load imbalance among MPI ranks. Figure 39 shows the minimum, maximum and average time of all ranks spent in distinct functions of the HPCG benchmark. The conclusion is that there is no significant load imbalance in the kernels but only in setup functions such as `GenerateProblem`. All features and use cases of the FTRACE viewer are discussed in the corresponding user's guide [99].

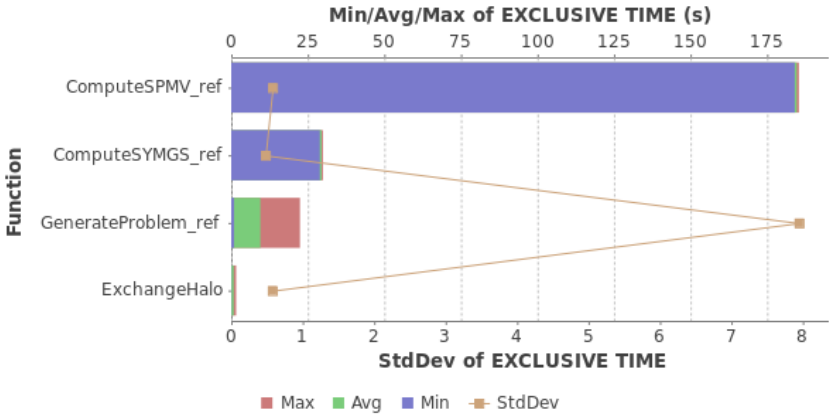

Figure 39. Detecting load imbalances using the FTRACE Viewer

5.3.3. veperf

The third performance analysis tool for the SX-Aurora TSUBASA is veperf. It acts as a performance monitor and provides almost the same information as PROGINF but at runtime. The original version [91] is extended by a more powerful version that shows memory bandwidth between cores and LLC in addition to power and energy consumption. Two program arguments are used to control the monitoring behaviour. The first one (`-n`) specifies which VE should be monitored and the second one (`-d`) specifies the output interval. Usually, veperf should be executed in a second terminal session. It is lightweight and does not significantly degrade the performance.

Example 17. veperf output for previously optimised HPCG implementation

```
pid           USRSEC     MFLOPS   AVGVL     VOPRAT     LOADBW    STOREBW
-> VE0
13114        223.35s       9877     243      98.9%    113.719     34.364
13115        223.37s       9874     244      98.9%    113.648     34.334
13116        223.36s       9873     243      98.9%    113.682     34.352
13117        223.37s       9873     244      98.9%    113.664     34.343
13118        223.36s       9875     244      98.9%    113.672     34.344
13119        223.37s       9871     244      98.9%    113.641     34.335
13121        223.37s       9870     243      98.9%    113.631     34.334
13122        223.36s       9861     241      98.8%    113.665     34.367
SUM VE0: MOPS    = 277961          MFLOPS  = 78974
         LOADBW = 909.322GB/s      STOREBW = 274.772GB/s
         POWER  = 138.8W           ENERGY  = 306.48kJ
```

The output shown in Example 17 corresponds to the HPCG benchmark optimised in this section. While power always represents the most recent measurement, energy is accumulated beginning from the moment when veperf was first called. The performance at the time of the monitoring interval is 78.94 GFLOPS in total. Apart from the information that we already discovered using PROGINF and FTRACE, the most interesting additional information concerns memory bandwidth and energy consumption. Data is moved from LLC to the cores at 909 GB/s and at 274 GB/s in the other direction. Normally we should expect a far higher ratio of loads to stores. `SpMV`, for example, with 27 entries per row should have at least 54 times more loads than stores. The frequent writing to LLC was introduced when we split up the loops in order to allow vectorisation. By manually unrolling the column-wise loop, as shown in Example 18, we avoid frequent store operations by updating a temporary variable that stays in the registers until it's final accumulation is done. In the code snipped of Example 19 the compiler is signaled to unroll the loop automatically. This is equivalent to manual loop unrolling since it requires the loop condition to be constant. However, both approaches are not allowed for official HPCG benchmark implementations as they are programmed for a fix sparsity pattern with a maximum of 27 entries per row.

Example 18. Sparse matrix-vector multiplication with manual loop unrolling

```
for(int i = 0; i < nrow; i++){
  double sum = 0.0;
  sum += A.matrixValues[0][i]*xv[A.mtxIndL[0][i]];
  sum += A.matrixValues[1][i]*xv[A.mtxIndL[1][i]];
...
  sum += A.matrixValues[26][i]*xv[A.mtxIndL[26][i]];
  sum += A.matrixValues[27][i]*xv[A.mtxIndL[27][i]];
  yv[i] = sum;
}
```

Example 19. Sparse matrix-vector multiplication with compiler-driven loop unrolling

```
for(int i = 0; i < nrow; i++){
  double sum = 0.0;
  #pragma _NEC unroll_completely
  for(int j = 0; j < 27; j++){
    sum += A.matrixValues[j][i]*xv[A.mtxIndL[j][i]];
  }
  yv[i] = sum;
}
```

Example 20. veperf output for optimised HPCG implementation using loop unrolling

```
pid          USRSEC     MFLOPS   AVGVL     VOPRAT     LOADBW    STOREBW
-> VE0
228867      223.22s      10329     244      99.3%     86.390      2.209
228868      223.23s      10327     244      99.3%     86.372      2.210
228869      223.23s      10326     243      99.3%     86.389      2.223
228870      223.23s      10326     243      99.3%     86.392      2.225
228871      223.23s      10325     241      99.2%     86.460      2.266
228872      223.23s      10328     243      99.3%     86.394      2.226
228873      223.24s      10328     243      99.3%     86.390      2.225
228875      223.23s      10328     243      99.3%     86.393      2.224
SUM VE0: MOPS   = 222639          MFLOPS  = 82617
         LOADBW = 691.180GB/s     STOREBW = 17.807GB/s
         POWER  = 129.3W          ENERGY  = 13.58kJ
```

The veperf report associated to the implementation featuring unrolling is given in Example 20. While the previous implementation had a ratio of 3.3 loads to one store, the unrolled one performs 38.8 loads for each store. The sustained performance increased by 4.4% and the power consumption decreased from 138.8W to 129.3W.

5.3.4. Final Recommendations and Guidelines

In this section, we demonstrated NEC's performance analysis tools using the HPCG benchmark. Even though our implementation achieved a speedup of over 50 compared to the reference implementation, there are further possible optimisations. The HPCG implementation optimised by NEC achieves a rating of 128.83 GFLOPS [90] for a single SX-Aurora TSUBASA device (Type 10B), which corresponds to 5.99% of it's peak performance. As most other systems score 1-3% of their peak performance for the HPCG benchmark, 5.99% is considered very efficient.

One of the vector processor's advantages is that ordinary C/C++ and Fortran applications developed for x86-architectures should compile without much porting effort. This fact is emphasised by the demonstration in this section. The HPCG benchmark compiled and executed without any modifications. The initial performance, however, was disappointing and optimisations are crucial for effective usage of the device. The four core lessons from this section are:

- Make use of the provided performance analysis tools to confirm assumptions and to discover unknown behaviours.
- Organise the data structures in such a way that the compiler can generate large vector instructions. This requires large leading dimensions and preferably stride-one access.
- Inspect the compiler reports to make sure that the correct loops are vectorised. If that is not the case, use NEC pragmas to guide the compiler. Be especially careful when using `-O3` and `-O4` since the loops might be transformed in an unintended or unfavourable way.
- Use loop-blocking or similar techniques to improve cache efficiency.

5.4. Performance Analysis with Extrae/Paraver: use case with GROMACS

Performance analysis plays an important role in standard parallelisation paradigms such as MPI and OpenMP. It has a crucial role when dealing with combined standard paradigms and CUDA because of the massive number of threads involved and the differences in the data transfer and storage. In such cases, profiling analysis tools capable of monitoring the thread behaviour at different levels (OpenMP, MPI, CUDA) are of major importance.

Extrae ("to extract" in Spanish) is a tool developed at Barcelona Supercomputing Center (BSC)[70] which allows to collect detailed traces by using different sampling mechanisms. The Extrae tool supports standard parallelisation programming models (also in combination) including MPI-CUDA, which is the model being described in this section. The traces collected with Extrae can be further analysed with its companion viewer tool Paraver ("to see" in Spanish) [71]. A beginners introduction to Extrae/Paraver can be found here [74].

In the present contribution, Extrae was used to analyse the GROningen MAchine for Chemical Simulations (GROMACS, v. 2020.4) [69] molecular dynamics simulation of a protein system consisting of 121,644 atoms (please refer to Section 7.2, “GROMACS” for further details). The analysis was done by using the MPI-CUDA version of GROMACS which was built with the following tool chain and Cmake options:

```
Dependencies: GCCcore/8.3.0, CMake/3.15.3, GCC/8.3.0, CUDA/10.1.243, \
              OpenMPI/3.1.4, and Extrae/3.7.1

cmake -DGMX_BUILD_OWN_FFTW=ON -DGMX_FFT_LIBRARY=fftw3 -DGMX_MPI=on \
-DCMAKE_INSTALL_PREFIX=PATH/gromacs-2020.4/install_gmx -DGMX_GPU=CUDA \
-DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=off \
-DGMX_INSTALL_NBLIB_API=OFF -DGMXAPI=OFF
```

In order to use Extrae, your application, in the present case GROMACS, must be compiled with debugging symbols enabled or with instrumented function calls. In the present case, we compiled GROMACS in debug mode which satisfies the former condition. In addition to this, both MPI and CUDA Cmake optios were enabled. CUDA related events can be traced by enabling the line: `cuda enabled="yes"` in the *extrae.xml* configuration file for Extrae. A typical SLURM batch script for tracing MPI-CUDA events with Extrae using 2 Nodes (with 2 GPUs each) looks like:

```
#!/bin/bash
#SBATCH -N 2
#SBATCH -n 4                 #Nr. MPI tasks, 1 per GPU
#SBATCH -c 14                #Nr. OpenMP threads per MPI task
#SBATCH --ntasks-per-node=2  #Forcing 2 tasks per node
#SBATCH --time=00:24:00      #Time for profiling analysis
#SBATCH --gres=gpu:v100:2    #GPUs per node

#Tool chain including Extrae
module load GCC/8.3.0  CUDA/10.1.243  OpenMPI/3.1.4
module load Extrae/3.8.3

source $PATH_EXTRAE/etc/extrae.sh                        #Source extrae.sh
export EXTRAE_CONFIG_FILE=extrae.xml                     #Extrae xml file
export LD_PRELOAD=${EXTRAE_HOME}/lib/libcudampitrace.so #MPI-CUDA lib.

#Running GROMACS
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
mpirun -np 4 $PATH_GROMACS/gmx_mpi mdrun -ntomp $SLURM_CPUS_PER_TASK \
-v -deffnm input_script
```

In Figure 40 we present a plot with the CUDA events collected by Extrae, among others: `cudaMemcpyAsync` (yellow), `cudaLaunch` (blue), `cudaStreamSynchronize` (red), `cudaStreamCreate` (light green), and `cudaThreadSynchronize` (pink) (a) together with a zoomed-in plot (b). Because of the large size of the resulting traces it is recommended to extract relevant chunks by using the Paraver complementary tool *paramedir* ("to measure" in Spanish).

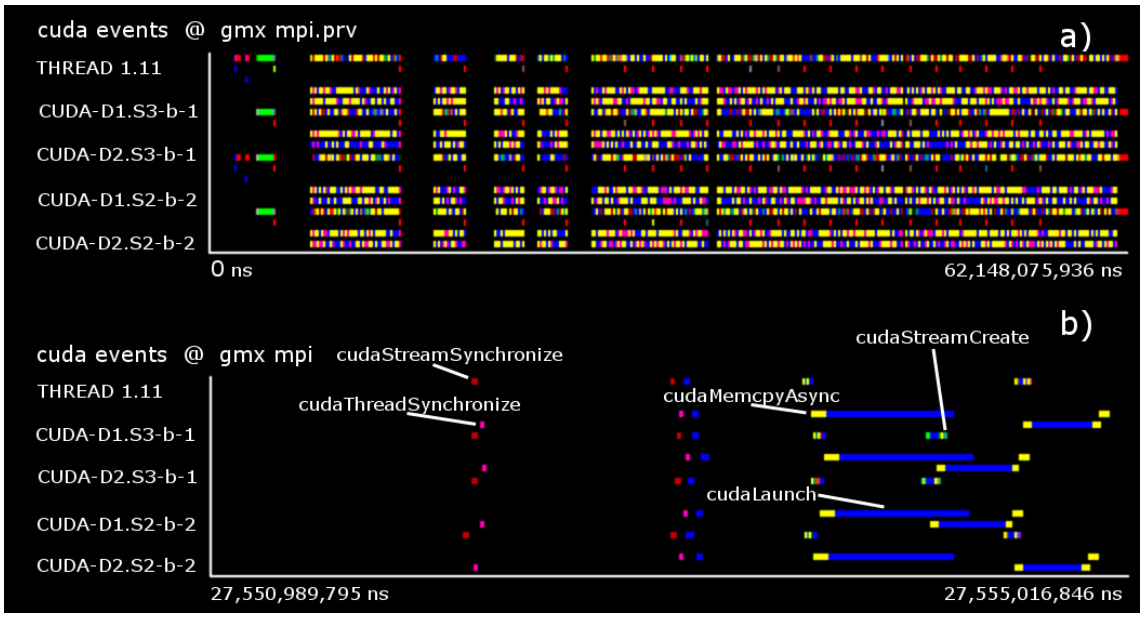

Figure 40. CUDA events of a GROMACS simulation obtained with Extrae tool.

In order to show some of the capabilities of Extrae/Paraver, we performed an analysis for a GROMACS test case where we fixed the number of GPUs per node (`--gres=gpu:v100:2`) and MPI processes (4) and changed the number of OpenMP threads from 14 to 6 (other parameters were kept fixed). Under these conditions, we observed a decrease in performance (measured in the standard performance unit for GROMACS nanoseconds per day) from 15.8 ns/day to 11.4 ns/day for 14 and 6 threads, respectively. The traces collected with Extrae and analysed with Paraver showed that the set of interactions offloaded to the GPUs are different in both cases, see Figure 41 a) and b). For instance, the interactions derived from the twin-range scheme are present in the 14 OpenMP case but not in the 6 OpenMP case after the initialisation period. This contributes to the lower number of kernels offloaded to GPUs in the 6 OpenMP case which can in turn contribute to the observed lower performance of GROMACS. The difference in the offloading pattern in GROMACS is most likely due to the internal mechanism of task assignment (for CPU and GPU) that depends on the available resources. This effect is considerably being minimised in the latest versions where most of the tasks can be offloaded to the GPU by the user explicitly.

In contrast to the plain MPI and OpenMP thread visualisation, the visualisation of CUDA events in Paraver is currently (version 3.8.0 and 3.9.0) under development. For instance, configuration (*.cfg*) files, which are required to extract the relevant metrics from Extrae traces, are not provided in the released version. Some configuration files, for instance the ones related to CUDA kernels and events, are provided as a section of a Paraver hands-on [71]. However, no further explanation can be found on the tools website at BSC. In addition to this, not all the hands-on configuration files work out of the box. For the present analysis, the ones that worked were CUDA events, 3DP CUDA kernel user functions and CUDA kernels. Manuals for Paraver are not updated currently regarding CUDA events support.

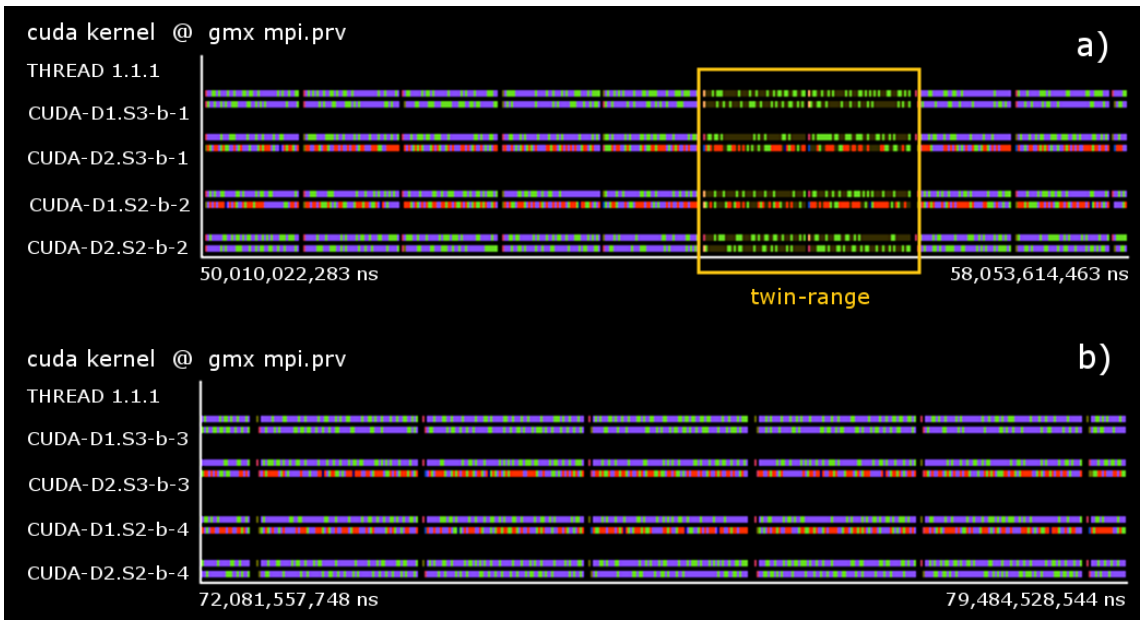

Figure 41. CUDA kernels of a GROMACS simulation obtained with Extrae tool. a) 14 OpenMP threads, b) 6 OpenMP threads.

In addition to the existing configuration files for Paraver, one can also use them in combination with additive and multiplicative metrics from the Performance Optimisation and Productivity center (POP) to monitor the efficiency of MPI-CUDA applications which are highly threaded [73].

Despite the above mentioned shortcomings in analysing traces with Paraver, we consider that learning the combined tools Extrae/Paraver is worth the effort because one can have an initial insight into the software performance without the need of instrumenting the code (for a more detailed analysis, code instrumentation could be required though). In addition to this, there is a well oriented and organised user's support for both Extrae/Paraver software. We also emphasise that the analysis from these software could be combined with the one from other tools such as Nvidia to obtain a more accurate idea of the performance of GPU applications.

5.5. Debugging

5.5.1. NVIDIA Debugging Tools

NVIDIA offers multiple debugging tools, such as NVIDIA Nsight, CUDA-GDB and CUDA-MEMCHECK. They are briefly described in Table 12.

Table 12. NVIDIA Tools Overview

Tool	Overview
NVIDIA Nsight Visual Studio Edition	Powerful IDE that integrates multiple NVIDIA Debugging tools. It is only available on Windows as complement for Microsoft Visual Studio.
CUDA-GDB	Command-line and graphical tool to control the execution of a CUDA application. It integrates CUDA-MEMCHECK too.
CUDA-MEMCHECK	Correctness checking. Integrated on other debugger tools. The standalone mode has more features like Racecheck, Initcheck and Synccheck.

5.5.2. AMD ROCm Debugger

The AMD ROCm Debugger (ROCgdb) is a source-level debugger for Linux based on the GNU Debugger (GDB). It enables heterogeneous debugging on the AMD ROCm platform and it is supported by the AMD Debugger API. It can be used with the standard GDB commands for both CPU and GPU code debugging [39].

5.5.3. TotalView

TotalView is a visual portable powerful debugger designed to handle complex CPU and GPU based multi-threaded, multi-process and multi-node cluster applications. The operating systems and compilers supported by TotalView vary depending on the platform used. They are defined in [41]. The languages and architectures supported by TotalView are shown in Table 13.

Table 13. TotalView Support

Languages supported	C, C++, Fortran and mixed-language Python (2.7 and 3.5 and above)
Architectures supported	Linux x86-64, Arm64, and OpenPower platforms
Accelerators supported	NVIDIA GPU (Tesla, Fermi, Kepler, Pascal, Volta, and Turing), NVIDIA Jetson Xavier
CUDA versions supported	8, 9, and 10

5.5.4. Vitis (HLS)

Paralleling the existence of separate HLS development flows (see Subsection 4.7.1 for details), Vitis supports the *software emulation*, *hardware emulation*, and *hardware* debugging flows. In addition, debug capabilities differ between host-side and kernel code.

The recommended approach is to spend as much debug time as possible in the software emulation mode: write-compile-debug cycles are faster and debugging capabilities are more extensive. Debug mode may be enabled in the Vitis GUI build configuration options.

For further details, please refer to the *Debugging Applications and Kernels* Vitis documentation subsection [42].

5.6. Summary

The following list of items provide a summary of the most important aspects discussed in this section.

- `nvprof` (command line) and Visual Profiler (GUI) are recommended for profiling CUDA applications. It is recommended to use the PGI compiler (now it is part of NVIDIA HPC SDK) for profiling the OpenACC applications.
- The briefly explained NVIDIA Compute (kernels level profiling) and NVIDIA Nsight Systems (for MPI CUDA applications) are profiling tools recommended for use with future generation NVIDIA GPUs.
- For CUDA applications (NVIDIA GPU based application) tuning, the following techniques shall be considered: CUDA occupancy calculator, register usage, memory usage and CUDA streams.
- Regarding OpenACC applications using PGI compiler (for any GPU architecture) tuning, it is recommended to use the different gang and vector combination depending on the applications and GPU architecture. And also the efficiency of the combination can be already seen at the compilation time.
- NEC vector processors work with standard C/C++ and Fortran code but applications developed for CPUs often have to be restructured for good performance.
- NEC provides many performance analysis tools that should be used to confirm assumptions and to discover unknown behaviours.

- For efficient usage of the NEC SX-Aurora the data structures have to be organized in such a way that the compiler can generate large vector instructions. This requires large leading dimensions and preferably stride-one access.
- Regarding the profiling tools for the analysis of CUDA aware applications, the Extrae tool for trace collection together with its graphical interface Paraver offer a good choice for obtaining a first insight into the functioning of these types of applications. A test case for GROMACS was studied in which different workloads were inspected at the CUDA thread level.
- Multiple debuggers are available for debugging codes running on accelerators.

6. European Systems

This section provides a brief overview on the available hardware infrastructures at some PRACE HPC sites and universities, that enable access to the earlier discussed accelerator technologies. More specifically: subsections 6.1 and 6.2 provide overview concerning systems utilizing NVIDIA GPUs (NVIDIA A100 and NVIDIA V100); subsection 6.3 discusses AMD MI50 GPU based system deployed at BSC; subsection 6.4 describes Intel's Hardware Accelerator Research Program (HARP) and outlines the possibilities for accessing Intel's FPGA resources; and subsection 6.5 concludes the section with the description of NEC SX-Aurora TSUBASA prototype system deployed at HLRS.

6.1. JUWELS Booster

JUWELS Booster consists of 936 compute nodes, each equipped with 4 NVIDIA A100 GPUs. The GPUs are hosted by AMD EPYC Rome CPUs. The compute nodes are connected with HDR-200 InfiniBand in a DragonFly+ topology. The Booster module designed for extreme computing power and artificial intelligence tasks extends the JUWELS Cluster module [118]. Both modules were designed by Atos in cooperation with ParTec, involving NVIDIA and Mellanox using a co-design process focusing on a modular supercomputing approach [117]. With the launch of the Booster in 2020 it was the fastest system in Europe and positioned seven amongst all systems listed in the November 2020 Top500 list [1], having a theoretical peak performance of 70 PFLOPS. Both modules, cluster and booster, can be used individually but are also tightly coupled to form a modular supercomputer system.

Figure 42. The JUWELS supercomputer, Copyright: Forschungszentrum Jülich / Ralf-Uwe Limbach

Each node has the following configuration:

- AMD EPYC 7402 processor; 2 sockets, 24 cores per socket, SMT-2 (total: 2×24×2 = 96 threads) in NPS-4 configuration
- Memory: 512 GB DDR4-3200 RAM (of which at least 20 GB is taken by the system software stack, including the file system); 256 GB per socket; 8 memory channels per socket (2 channels per NUMA domain)
- GPU: 4 × NVIDIA A100 Tensor Core GPU with 40 GB; connected to each other via NVLink3
- Network: 4 × Mellanox HDR200 InfiniBand ConnectX 6 (200 Gbit/s each), HCA
- Periphery: CPU, GPU, and network adapter are connected via 2 PCIe Gen 4 switches with 16 PCIe lanes going to each device (CPU socket: 2×16 lanes).

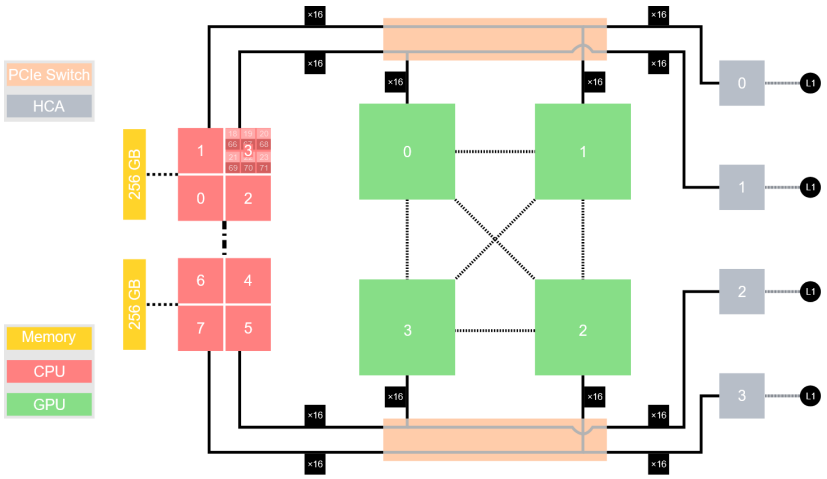

Figure 43. Layout of a single JUWELS Booster node

Due to the four individual NUMA domains, the default process pinning on JUWELS Booster is adapted to spawn processes first in NUMA domains 3,1,5,7 (in this order) to align them to the individual GPUs. This allows a common 'one MPI task per GPU' setup to always use the closest GPU per CPU task, which can have a significant impact on the Memory-CPU-GPU communication process.

6.1.1. Programming Environment

The programming environment is built on top of an EasyBuild [119] based software stack, which provides a larger number of individual scientific software packages. The main compilers provided on JUWELS Booster are NVHPC and GCC. For MPI, ParaStationMPI and OpenMPI are supported as both offer GPU-aware capabilities. The overview on the full software stack can be found in the JUWELS Booster module overview: https://apps.fz-juelich.de/jsc/llview/juwels_modules_booster/

6.2. Iris Cluster, Luxembourg University

6.2.1. Iris Cluster Overview

Iris [141] is a Dell [120]/Intel-based supercomputer configured with 196 compute nodes, in a total of 5824 compute cores and 52224 GB RAM, with a peak performance of about 1,072 PFLOPS.

All nodes at Iris are interconnected through a fast InfiniBand EDR network, configured over a fat-tree topology [121] (blocking factor 1:1.5). Iris nodes are built with Intel Broadwell [122] or Skylake [123] processors. Moreover, several nodes are equipped with 4 NVIDIA Tesla V100 [124] SXM2 GPU accelerators. In total, Iris features 96 NVIDIA V100 accelerators that facilitate the GPU enabled applications to be run on the GPUs, in particular ML/AI software. Finally, a few large-memory (fat) computing nodes offer multiple high-core density CPUs and a large memory capacity of 3 TB RAM/node allowing the large memory computations to be handled within the node. The Luxembourg HPC (ULHPC) system storage is based on three high-performance file systems: GPFS/SpectrumScale, Lustre and oneFS.

Iris cluster runs a Red Hat Linux Family [125] operating system, which offers various HPC utilities, scientific applications, and programming libraries to its user community on all clusters. The user software environment is generated using Easybuild (EB) [126] and is made available through environment modules [127] from the compute nodes. The SLURM [128] is the Resource and Job Management Systems (RJMS), which provides computing resources allocations and job execution. Figure 44 provides an overview of the Iris cluster.

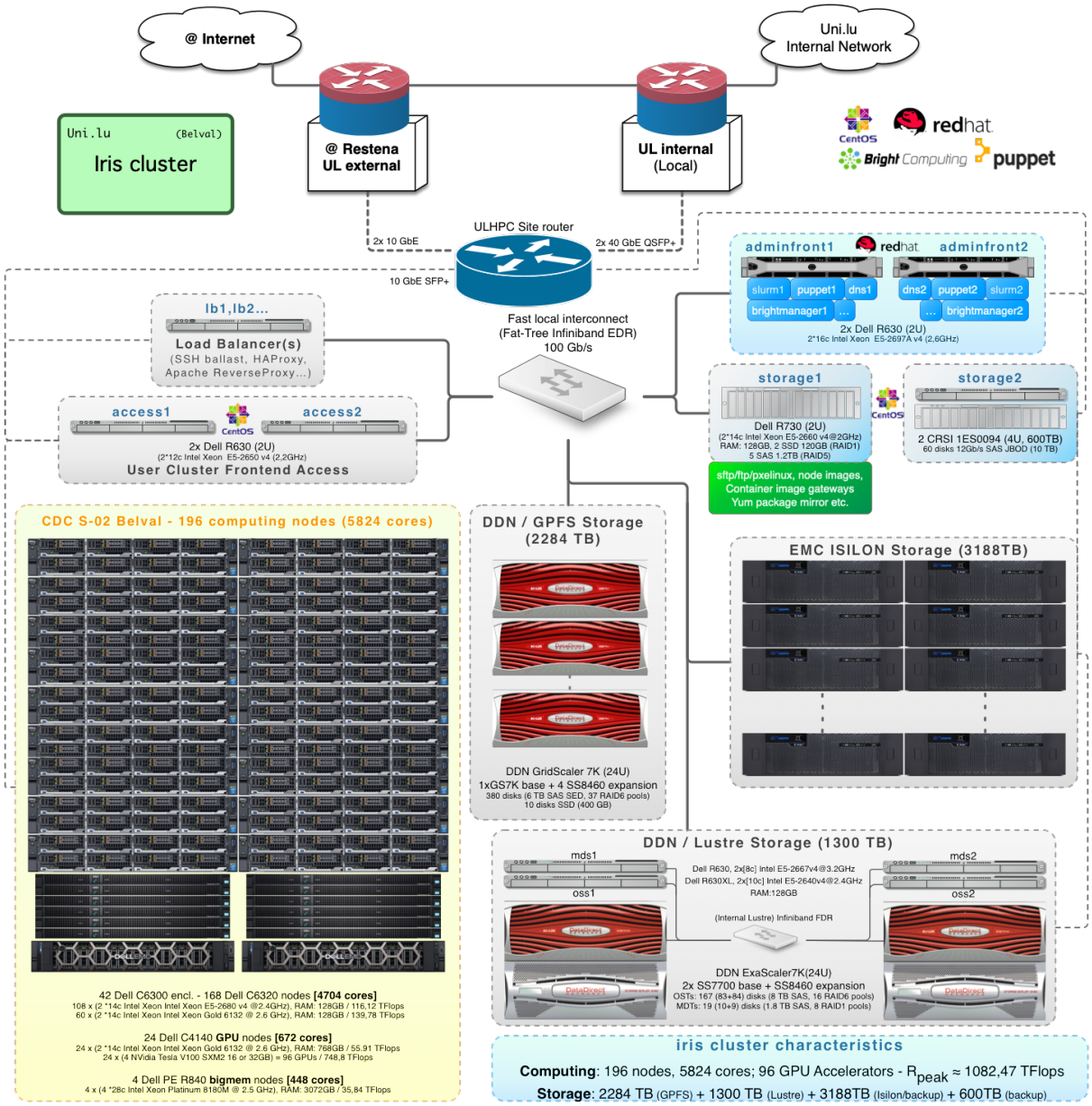

Figure 44. Iris Cluster Organisation

The ULHPC has a login node where users can log in. On the login node, users are not allowed to do any computation. Moreover, users can not load/add any modules (software) to it. For conducting a computation, users should consider the usage of compute node(s) for both active (interactive) and passive (batch jobs) computation.

6.2.1.1. Compute Nodes

The Iris cluster has x86-64 Intel-based compute nodes. Furthermore, it has 196 compute nodes, named iris-[001-196], and the architecture of these compute nodes are listed in Table 14.

Table 14. ULHPC compute node configuration

Hostname (#Nodes)	Node type	Processor	RAM
iris-[001-108] (108)	regular Broadwell	2 Xeon E5-2680v4 @ 2.4 GHz [14 c/120 W]	128 GB
iris-[109-168] (60)	regular Skylake	2 Xeon Gold 6132 @ 2.6 GHz [14 c/140 W]	128 GB

Hostname (#Nodes)	Node type	Processor	RAM
iris-[169-186] (18)	multi-GPU Skylake	2 Xeon Gold 6132 @ 2.6GHz [14c/140W] 4x Tesla V100 SXM2 16GB	768GB
iris-[187-190] (4)	large Memory Skylake	4 Xeon Platinum 8180M @ 2.5GHz [28c/205W]	3072GB
iris-[191-196] (6)	multi-GPU Skylake	2 Xeon Gold 6132 @ 2.6GHz [14c/140W] 4x Tesla V100 SXM2 32GB	768GB

6.2.1.2. Performance of Processors

The Iris nodes are based on the Intel x86_64 processor architecture of Broadwell or Skylake. This architecture's performance is listed in Table 15.

Table 15. Intel's Broadwell and Skylake performance

Processor Model	#cores	TDP(*)	CPU Freq. (AVX-512 T.Freq.)	R_{peak} [TFLOPS]	R_{max} [TFLOPS]
Xeon E5-2680v4 (Broadwell)	14	120W	2.4GHz (n/a)	0.538TF	0.46TF
Xeon Gold 6132 (Skylake)	14	140W	2.6GHz (2.3GHz)	1.03TF	0.88TF
Xeon Platinum 8180M (Skylake)	28	205W	2.5GHz (2.3GHz)	2.06TF	1.75TF

(*)Thermal Design Power (TDP), in watts, indicates the maximum power consumption a processor can consume without overheating.

6.2.1.3. Accelerators Performance

The Iris cluster is equipped with 96 NVIDIA Tesla V100-SXM2 GPU accelerators with 16 or 32GB of GPU memory. Each GPU node interconnected by the NVLink 2.0 architecture [65] described in Chapter 5, which provides faster bandwidth and improved scalability for multi-GPU system configurations. Table 16 reports the GPU configuration and performance of the ULHPC system.

Table 16. ULHPC GPU configuration and performance

NVIDIA GPU Model	#CUDA cores	#Tensor cores	Maximum Power Consumption	Interconnect Bandwidth	GPU Memory	R_{peak} [TFLOPS]
V100-SXM2	5120	640	300W	300GB/s	32GB	7.8TF
V100-SXM2	5120	640	300W	300GB/s	16GB	7.8TF

6.2.2. Network

ULHPC uses two kinds of interconnects for supporting the operation of its HPC systems. Iris system utilizes EDR IB fabric using a fat-tree topology, whereas the other major HPC system Aion (based on AMD Epyc ROME 7H12 processors) utilizes HDR100 IB fabric (also using a fat-tree topology).

Despite the recent acquisition of Mellanox by NVidia, the fast interconnect technology based is placed at the ULHPC does not yet permit multi-GPU runs to bind efficiently the internal GPU interconnect (based on NVLink 2.0 architecture) and the inter-node IB network. Nevertheless, several scalability evaluations of the reference Deep Learning frameworks such as Tensorflow, Keras, or PyTorch over the distributed training framework Horovod

[144] demonstrated already excellent performances allowing to reach a throughput speedup of 10.11 when using 12 NVidia Tesla V100 GPUs when training Resnet44 on the CIFAR10 dataset [145].

6.2.3. Software/Modules Environment

The efficient exploitation of heterogeneous computing resources featuring different processor architectures and generations, coupled with the presence of GPU accelerators, remains a challenge. In all HPC centres, support teams invest a significant amount of time (i.e., several months of effort per year) in providing a software environment optimised for hundreds of users, but the complexity of HPC software was quickly outpacing the capabilities of classical software management tools. Since 2014, the ULHPC scientific software stack (including for GPU compute nodes) is generated and deployed in an automated and consistent way through the RESIF framework, a wrapper on top of Easybuild and Lmod[119] meant to efficiently handle user software generation. The latest release of this framework, RESIF 3.0 [142] implements a major code refactoring operated in 2020 with the advent of the new supercomputer Aion featuring a different CPU architecture.

The HPC software environment is exposed to the facility users through the modules [130] command. ULHPC modules and their associated software are kept organised through the categorized naming scheme - they are thus hosted within dedicated directories reflecting the cluster and the architecture the applications were compiled against for optimised builds. In particular, the production-ready modules and software are organized as depicted in Figure 45 where the MODULEPATH environment variable is set to hold only one of the directory associated to a given CPU architecture (and a default environment is defined as a symbolic link targeting the commonly supported CPU architecture for a given cluster such as broadwell for the Intel-based Iris cluster). A notable exception of importance for the present report concerns the GPU computing nodes for which the accelerated CUDA-optimised scientific applications are kept as first searched modules, but completed by CPU-optimised builds (skylake in the case of Iris) to offer the most flexible environment.

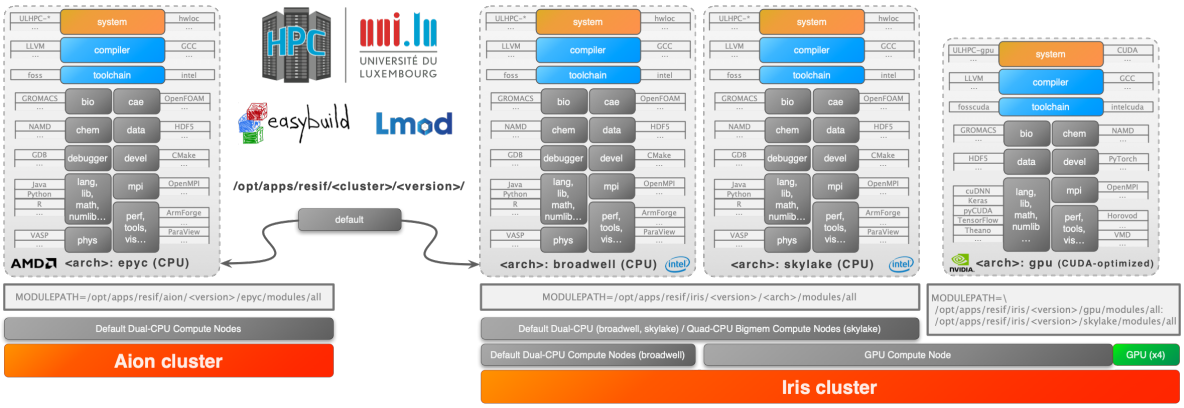

Figure 45. ULHPC Software modules organization

At the heart of RESIF 3.0 stands the concept of ULHPC bundles defining the software sets to be built for a given software release, aligned with the Easybuild toolchain release cycle. This permits fixing the major software dependency components in the bundles which facilitates the porting of a given bundle definition from one version to another. Table 17 summarizes the components version set for the current software set releases. The GPU bundles, relying on the fosscuda and intelcuda toolchains, can be found in the public repository hosted on Github [143].

Table 17. ULHPC software set releases characteristics (*: projections)

Toolchain component	2019a (deprecated)	2019b (old)	2020a (prod)	2021a* (devel)
GCCCore	8.2.0	8.3.0	9.3.0	10.3.0
foss[cuda]	2019a	2019b	2020a	2021a
intel[cuda]	2019a	2019b	2020a	2021a
binutils	2.31.1	2.32	2.34	2.36
Python	3.7.2 (and 2.7.15)	3.7.4 (and 2.7.16)	3.8.2 (and 2.7.18)	3.9.2

6.3. BSC CTE-AMD

CTE-AMD is a cluster based on AMD EPYC processors, with a Linux Operating System and an Infiniband interconnection network. Its main characteristic is the availability of 2 AMD MI50 GPUs per node, making it an ideal system for GPU accelerated applications.

It has the following configuration:

- 1 login node and 33 compute nodes, each of them:1 x AMD EPYC 7742 @ 2.250 GHz (64 cores and 2 threads/core, total 128 threads per node)
  - 1024 GB of main memory distributed in 16 DIMMs x 64 GB @ 3200 MHz
  - 1 x SSD 480 GB as local storage
  - 2 x GPU AMD Radeon Instinct MI50 with 32 GB
  - Single Port Mellanox Infiniband HDR100
  - GPFS via two copper links 10 Gbit/s

The operating system is CentOS Linux release 8.1.1911 (Core).

Figure 46. BSC CTE-AMD Rack

CPU EPYC 7742

- 64 cores (128 threads) with a base Clock of 2.25 GHz (Max Boost Clock Up to 3.4 GHz)
- Total L3 Cache 256 MB
- PCI Express Version PCIe 4.0 x128
- Default TDP / TDP 225 W

GPU Instinct™ MI50 (32 GB)

- GPU Architecture Vega20 (Lithography TSMC 7nm FinFET)
- 3840 Stream Processors and 60 Compute Units (Peak Engine Clock 1725 MHz)
  - Peak Performance Half Precision (FP16): 26.5 TFLOPS
  - Peak Performance Single Precision (FP32): 13.3 TFLOPS
  - Peak Performance Double Precision (FP64): 6.6 TFLOPS

Figure 47. BSC CTE-AMD Node

6.3.1. Programming Environment

6.3.1.1. AOCC

AMD Optimizing C/C++ Compiler (AOCC) is a highly optimised C, C++ and Fortran compiler for x86 targets especially for Zen based AMD processors. It supports Flang as the default Fortran front-end compiler.

6.3.1.2. AOCL

AMD Optimizing CPU Libraries (AOCL) is a set of numerical libraries optimised for AMD EPYCTM processor family. AOCL comprises of eight packages:

- BLIS (BLAS Library) – BLIS is a portable open-source software framework for instantiating high-performance Basic Linear Algebra Subprograms (BLAS) functionality.
- libFLAME (LAPACK) - libFLAME is a portable library for dense matrix computations, providing much of the functionality present in Linear Algebra Package (LAPACK).
- FFTW – FFTW (Fast Fourier Transform in the West) is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases thereof.
- LibM (AMD Core Math Library) - AMD LibM is a software library containing a collection of basic math functions optimised for x86-64 processor-based machines.
- ScaLAPACK - ScaLAPACK is a library of high-performance linear algebra routines for parallel distributed memory machines. It depends on external libraries including BLAS and LAPACK for Linear Algebra computations.
- AMD Random Number Generator Library - AMD Random Number Generator Library is a pseudorandom number generator library.

- AMD Secure RNG - The AMD Secure Random Number Generator (RNG) is a library that provides APIs to access the cryptographically secure random numbers generated by AMD's hardware random number generator implementation.
- AOCL-Sparse - Library that contains basic linear algebra subroutines for sparse matrices and vectors optimised for AMD EPYC family of processors

6.3.1.3. ROCm

AMD ROCm is the first open-source software development platform for HPC/Hyperscale-class GPU computing. AMD ROCm brings the UNIX philosophy of choice, minimalism and modular software development to GPU computing.

6.4. Intel Hardware Accelerator Research Program (HARP)

Intel established the HARP as a way to provide Universities an access to computer systems containing Intel microprocessors and an Altera Arria 10 FPGA in the same chip. This kind of integration is novel, and opens several research and application possibilities. The second iteration of this program (HARP2) made available Intel Xeon +FPGA systems (Broadwell + Arria 10, see Figure 48) via two centralized cluster installations in the US and in Germany (e.g. Paderborn Center for Parallel Computing [111]). Intel Labs' Academic Compute Environment (ACE) [112] is the most recent iteration of this program, and provides remote access to Intel FPGA resources. This program is open to all researchers, who can apply using a form[10].

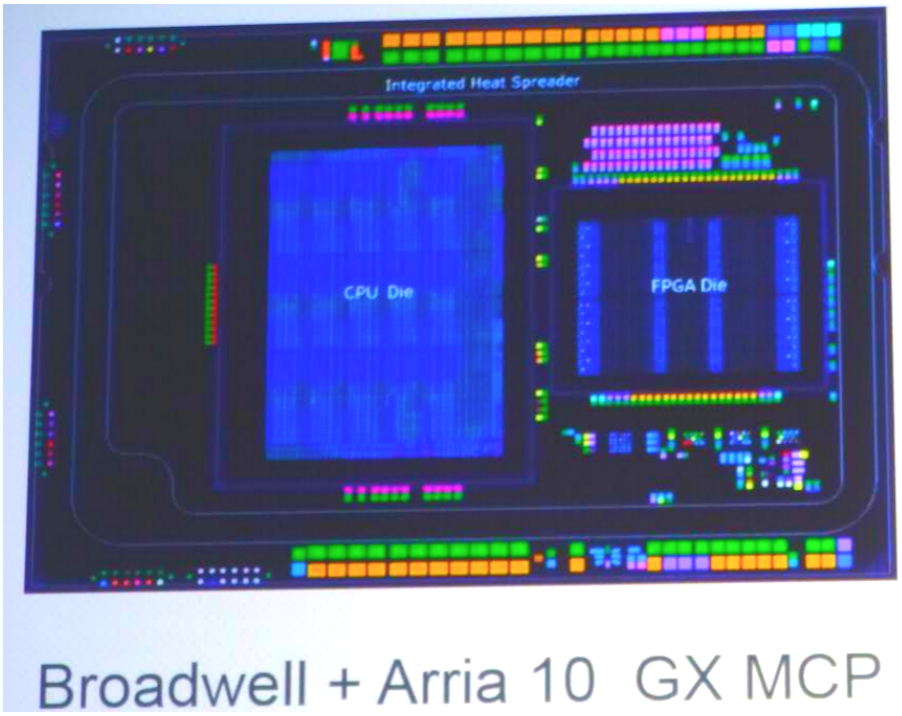

Figure 48. The Intel Xeon+FPGA chip that powers the HARP2 system [115]

The program currently offers the following configurations, which support FPGA programming both with OpenCL and RTL:

- PCIe D5005 Programmable Acceleration Cards (PACs) with a Stratix 10 SX FPGA. They have PCIe Gen 3 x16 and 4 channels of 8GB DDR4-2400 with ECC.
- Two PCIe D5005 PACs with a Stratix 10 SX FPGA. They can be used to test using multiple cards that communicate through system memory. In the future they plan to connect the cards through QSFP+ ports.

[10] https://registration.intel-research.net/register

- PCIe PACs with an Arria 10 GX FPGA. Two cards are installed in each system with the 40GbE QSFP+ ports connected to each other, in order to facilitate research on networked FPGAs.

- Broadwell Xeon CPUs (E5-2600v4) with an integrated in-package Arria 10 GX1150 FPGA.

Each compute node has access to both NFS and Lustre file systems. The nodes are connected to each other by Intel Omni-Path (Intel OPA) 100 Series interconnect, and all nodes are connected via 10Gb Ethernet. To access the compute nodes users should use `ssh`, and jobs can be submitted with `qsub`. Local users are allotted 20GB, while the user's institution has 1TB of storage.

To compile OpenCL-based FPGA designs, they must be submitted to Lustre. Users only need to submit the `.cl` file, and after compilation, there will be a folder with the same name as the project which will contain a `.aoco` file, which is an intermediate object file that contains information for later stages of the compilation, and a `.aocx` file, which is the bitstream file that programs the FPGA at runtime. If user needs to recompile (e.g. in case there is an error), it is recommended that the complete folder is deleted in order to perform a clean compilation.

The bash configuration script that the user launches will set the FPGA target. For instance, `fpga-bdx-opencl` will allow the user to compile to the Arria 10 + Xeon, while `fpga-pac-s10` will target the Stratix 10. By default, the `qsub` command will execute on a Xeon and will not provide an access to the FPGA. The user will need to use the flag `-fpga` to load your bitstream in the FPGA. Please note that the resources being accessed with `qsub -fpga` are extremely limited, and the computation time of those jobs should be minimised (e.g. just run the FPGA). All compilation jobs must be submitted with the default `qsub` command.

6.5. NEC SX-Aurora TSUBASA Prototype System at HLRS

HLRS's Vulcan cluster hosts a vector partition [95] that contains 8 A300-8 systems [96]. The nodes form a fat tree topology through a common switch via two InfiniBand EDR HCAs. Each A300-8 system features:

- 8 vector engines of type 10B

- 17.2TFLOPS vector engine performance

- 384GB vector engine memory

- 1 vector host (2x Intel Xeon Gold 6126 CPU)

- 384GB main memory

The vector partition was officially benchmarked for HPCG and ranked #123 in Top500 list[97]. It achieved 7TFLOPS at 5.1% of it's peak performance (137.6TFLOPS). The cluster is mainly used for CFD simulations as they favor high memory bandwidth over pure computational power.

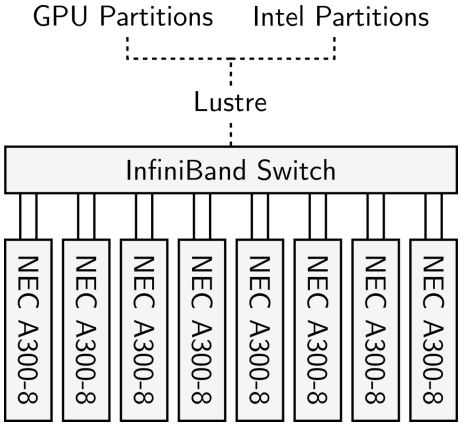

Figure 49. SX-Aurora prototype system integrated into HLRS's Vulcan cluster

Since the Prototype system is part of the Vulcan cluster, the login nodes and scheduling system (PBS) of the latter are used. The operating system is CentOS 7. Interactive and batch-jobs targeting the SX-Aurora system are submitted to the *vector*-queue, which does not have any additional restrictions. Besides the specialised VE-Software from NEC, users can utilise the entire software stack of Vulcan on the x86-VH. One restriction is that applications have to be compiled on the compute nodes because the NEC programming environment is not available on the login nodes.

7. Brief Description of Used Applications/Benchmarks

7.1. HPCG Benchmark

The high performance conjugate gradient (HPCG) benchmark, developed by Jack Dongarra, Michael A. Heroux and Piotr Luszczek, complements the high performance linpack (HPL) benchmark. Both HPCG and HPL are essentially equation solvers but from a computational perspective they behave completely different. The task of solving dense linear systems of equations (HPL) mostly comes down to DGEMM (matrix multiplication), which has high arithmetic intensity. HPCG solves sparse linear systems of equations using Krylov and multigrid methods that essentially consist of sparse matrix vector multiplications and stencil operations. Both have low arithmetic intensity. Therefore, modern supercomputers usually achieve more than 70% efficiency for HPL but less than 3% for HPCG. Due to the nature of numerical simulation, most PDE-solvers rely on sparse matrices which makes the HPCG benchmark a far more realistic depiction of actual modern supercomputing.

7.2. GROMACS

The GROningen MAchine for Chemical Simulations (GROMACS) is a highly parallel and highly scalable software for solving the classical Newton's equations to simulate the dynamics of Biomolecules and inorganic compounds in a method known as Molecular Dynamics. GROMACS developers make a continuous effort in optimising this engine and because of that, it can run efficiently in novel hardware such as GPUs. GROMACS has support for hybrid systems through the use of combined parallelisation layers such as MPI, OpenMP, and CUDA.

A. Acronyms and Abbreviations

1. Units

n	S.I. Unit nano = 10^{-9}
ns	S.I. Unit nanosecond = 10^{-9} second
k	S.I. Unit kilo = 10^{3}
M	S.I. Unit Mega = 10^{6}
G	S.I. Unit Giga = 10^{9}
T	S.I. Unit Tera = 10^{12}
Byte	8 bits, (an octet)
GHz	Giga Hertz, Hertz is frequency, events per second.
MHz	Mega Hertz, Hertz is frequency, events per second.
nm	Nano meter.
W	Watt, units of energy per second, Joules/second

2. Acronyms

AOCC	AMD Optimizing C/C++ Compiler
AOCL	AMD Optimizing CPU Libraries
ASIC	Application-Specific Integrated Circuit
AVX	Advanced Vector Instructions, a set of extra vector (SIMD) instructions added to the SSE vector instruction set. There is also a an extra addition, AVX2
AXPY	A common way to abbreviate vector-vector addition operation (scAlar times X Plus Y)
BPG	Best Practice Guide
BSC	Barcelona Supercomputing Center
CAPS	CAPS, the Many-core Programming Company, is a provider of software and solutions for HPC Community.
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CSR	Compressed Sparse Row, a storage format for sparse matrices
CUDA	Compute Unified Device Architecture
DDR4	Double Data Rate 4 Synchronous Dynamic Random-Access Memory
DIMM	Dual In line Memory Module, variants RDIMM (registered ECC), LRDIMM (Load Reduced DIMM)
DNN	Deep Neural Networks

DP	Double Precision
DSP	Digital Signal Processors (DSP)
EDA	Electronic Design Automation
EuroHPC JU	European High-Performance Computing Joint Undertaking
FF	Flip-Flop
FFT	Fastest Fourier Transformation
FLOPS	Floating Point Operation per Second
FMA	Fused Multiply-Add
FPGA	Field-Programmable Gate Array
GEMM	GEneral Matrix to Matrix Multiplication
GPC	GPU Processing Clusters
GPU	Graphics Processing Unit
GPFS	General Parallel File System
GROMACS	GROningen MAchine for Chemical Simulations
HARP	Hardware Accelerator Research Program
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HDR	Mellanox InfiniBand High Data Rate, 200 Gbits/s. HDR-100 is a split of the 4x IB into two 2x each with 100 Gbits/s capacity
HIP	Heterogeneous-Computing Interface for Portability
HLRS	High Performance Computing Center Stuttgart
HLS	High-Level Synthesis
HPC	High-Performance Computing
HPCG	High Performance Conjugate Gradients (Benchmark)
HPL	High-Performance Linpack (Benchmark)
IB	InfiniBand
LAPACK	Linear algebra package, higher level linear algebra subroutines
LAN	Local Area Network
LLC	Last Level Cache
LRZ	Leibniz Supercomputing Centre
LUT	Look-Up Table
MPI	Message Passing Interface
NUMA	None-Uniform Memory Access

NVCC	NVIDIA CUDA Compiler
OpenMP	Open Multi-Processing
PAC	Programmable Acceleration Card
PCIe	Peripheral Component Interconnect Express
POP	Performance Optimisation and Productivity
PTX	Parallel Thread Execution
PGI	Formerly Portland Group Inc. Presently part of NVIDIA, products are now known as PGI compilers
PDE	Partial Differential Equation
RAM	Random Access Memory
RJMS	Resource and Job Management Systems
RTL	Register-Transfer Level
SDK	Software Development Kit
SFU	Special Function Unit
SIMD	Single Instructions Multiple Data
SLURM	Simple Linux Utility for Resource Management
SM	Streaming MultiProcessor
SMT	Simultaneous Multi Threading
SpMV	Sparse Matrix-Vector Multiplication
SPMD	Single Program, Multiple Data
SVE	Scalable Vector Extension [7]
SymGS	Symmetric Gauss Seidel
TDP	Thermal Design Power
TPC	Texture Processing Clusters
ULHPC	University of Luxembourg HPC
VEOS	Vector Engine Operating System
VE	Vector Engine
VH	Vector Host
WAN	Wide Area Network

Further documentation

[1] TOP500 list November 2020 [https://www.top500.org/lists/top500/2020/11/].

[2] The European High Performance Computing Joint Undertaking (EuroHPC JU) [https://eurohpc-ju.europa.eu/].

[3] PRACE Best Practice guides [https://prace-ri.eu/training-support/best-practice-guides/].

[4] PRACE Best Practice Guide Modern Processors https://prace-ri.eu/training-support/best-practice-guides/modern-processors/ .

[5] Software Intel, Intel® Math Kernel Library (Intel® MKL) [https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html].

[6] Fastest Fourier Transfer in the West A popular open source implementation of Fourier Transforms [http://www.fftw.org/].

[7] Explore the Scalable Vector Extention [https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator/resources/tutorials/sve].

[8] Intel product specifications [https://ark.intel.com/].

[9] NVIDIA Tesla V100 product specifications [https://www.nvidia.com/en-gb/data-center/tesla-v100/].

[10] Xilinx Alveo product specifications [https://www.xilinx.com/products/boards-and-kits/alveo.html].

[11] Intel® Stratix® 10 GX/SX Product Table [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf].

[12] NEC SX-Aurora TSUBASA product specifications [https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html].

[13] DigiKey Electronics Homepage [https://www.digikey.com].

[14] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM. 52. 4. 65–76. 2009.

[15] Programming Guide :: CUDA Toolkit Documentation [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html].

[16] NVIDIA TESLA V100 GPU ARCHITECTURE [https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf].

[17] INTRODUCING AMD CDNA ARCHITECTURE [https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf].

[18] NVIDIA CUDA-X: GPU-Accelerated Libraries [https://developer.nvidia.com/GPU-ACCELERATED-LIBRARIES].

[19] NVCC :: CUDA Toolkit Documentation [https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html].

[20] OpenCL homepage [https://www.khronos.org/opencl/].

[21] SYCL specification [https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf].

[22] Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, Xinmin Tian, and James Reinders. Data Parallel C++, Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. Apress. 2020.

[23] DPC++ model [https://docs.oneapi.com/versions/latest/dpcpp/model/index.html].

[24] OneAPI framework [https://software.intel.com/content/www/us/en/develop/tools/oneapi.html].

[25] ComputeCpp Compiler [https://developer.codeplay.com/].

[26] Intel DevCloud [https://devcloud.intel.com/edge/get_started/devcloud/].

[27] Intel DPC++ open-source compiler [https://www.github.com/intel/llvm/].

[28] hipSYCL compiler [https://github.com/illuhad/hipSYCL].

[29] OpenACC Homepage [https://www.openacc.org].

[30] NVIDIA HPC SDK Installation Guide [https://docs.nvidia.com/hpc-sdk/archive/20.11/hpc-sdk-install-guide/index.html].

[31] OpenACC 2.7 Standard [https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf].

[32] OpenACC Programming and Best Practices Guide [https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf].

[33] HIP Programming Guide [https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html].

[34] HIP API Guide [https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP_API_Guide.html].

[35] Nsight Visual Studio Edition Documentation and Support [https://developer.nvidia.com/nsight-visual-studio-edition-documentation-and-support].

[36] CUDA-GDB User Manual [https://docs.nvidia.com/cuda/cuda-gdb/index.html].

[37] CUDA-MEMCHECK User Manual [https://docs.nvidia.com/cuda/cuda-memcheck/index.html].

[38] CUDA-MEMCHECK Hardware Exception Reporting [https://docs.nvidia.com/cuda/cuda-memcheck/index.html#hardware-exception-reporting].

[39] AMD ROCm Debugger Documentation [https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCgdb.html#].

[40] TotalView Debugger [https://developer.nvidia.com/totalview-debugger].

[41] TotalView Supported Platforms [https://help.totalview.io/current/PDFs/TotalView_Platforms_Guide.pdf].

[42] Vitis Documentation: Debugging Applications and Kernels [https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/debuggingapplicationskernels.html].

[43] Vivado Design Suite User Guide: Programming and Debugging [https://www.xilinx.com/support/documentation-navigation/see-all-versions.html?xlnxproducttypes=Design%20Tools&xlnxdocumentid=UG908].

[44] NVIDIA Events and Metrics [https://docs.nvidia.com/cuda/profiler-users-guide/index.html#terminology].

[45] NVIDIA nvprof [https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview].

[46] NVIDIA Visual Profiler [https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual].

[47] NVIDIA Nsight Compute [https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual].

[48] NVIDIA Nsight Compute Comparison [https://docs.nvidia.com/nsight-compute/2020.3/ProfilingGuide/index.html].

[49] CUDA Occupancy Calculator [https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html#abstract].

[50] NVIDIA NVLink [https://www.nvidia.com/en-us/data-center/nvlink/].

[51] NVIDIA Magnum IO [https://www.nvidia.com/en-us/data-center/magnum-io/].

[52] https://docs.nvidia.com/gpudirect-storage/overview-guide/index.htmlGPUDirect Storage.

[53] https://docs.nvidia.com/cuda/gpudirect-rdma/index.htmlGPUDirect RDMA.

[54] https://developer.nvidia.com/gpudirectGPUDirect.

[55] https://www.openacc.org/get-startedOpenACC Programming Model.

[56] https://www.pgroup.com/index.htmPGI Compiler.

[57] https://www.open-mpi.org/faq/?category=runcudaRunning CUDA-aware Open MPI.

[58] http://mvapich.cse.ohio-state.edu/userguide/gdr/MVAPICH2-GDR 2.3.5.

[59] https://www.mpich.org/Message Passing Interface (MPI) standard.

[60] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.htmlIntel® MPI Library.

[61] https://www.olcf.ornl.gov/tutorials/gpudirect-mpich-enabled-cuda/GPUDirect MPICH Enabled CUDA.

[62] https://docs.mellanox.com/display/GPUDirectRDMAv17/Benchmark+TestsNVIDIA Mellanox GPUDirect RDMA User Manual Benchmark Tests.

[63] https://www.mellanox.com/products/GPUDirect-RDMAMellanox OFED GPUDirect RDMA.

[64] https://www.nvidia.com/en-us/data-center/ampere-architecture/NVIDIA Ampere Architecture.

[65] https://www.nvidia.com/en-us/data-center/nvlink/NVLink AND NVSwitch.

[66] https://www.pgroup.com/index.htmPGI Compilers and Tools have evolved into the NVIDIA HPC SDK.

[67] https://docs.nvidia.com/nsight-systems/UserGuide/index.html#cli-analyze-mpi-codesNsight Systems: Using the CLI to Analyze MPI Codes.

[68] Ezhilmathi Krishnasamy. Hybrid CPU-GPU Parallel Simulations of 3D Front Propagation. 2014.

[69] M.J. Abraham, T. Murtola, R. Schulz, S. Pall, J.C. Smith, B. Hess, and E. Lindahl, GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers, Software X, 1-2, 19-25 (2015).

[70] Extrae profiling tool [https://tools.bsc.es/extrae].

[71] Paraver trace viewer [https://tools.bsc.es/paraver].

[72] Paraver hands-on [https://tools.bsc.es/tools_hands-on].

[73] POP metrics [https://pop-coe.eu/blog/19th-pop-webinar-identifying-performance-bottlenecks-in-hybrid-mpi-openmp-software].

[74] Introduction to different profiling tools including Extrae/Paraver [https://prace-ri.eu/wp-content/uploads/WP237.pdf].

[75] NEC SX-Aurora TSUBASA - Vector Engine [https://www.nec.com/en/global/solutions/hpc/sx/index.html].

[76] NEC SX-ACE [https://de.nec.com/de_DE/en/documents/SX-ACE-brochure.pdf].

[77] NEC SX-Aurora TSUBASA - Brochure [https://www.nec.com/en/global/solutions/hpc/sx/docs/SX_Aurora_TSUBASA_brochure_2020_oct.pdf].

[78] NEC SX-Aurora TSUBASA - Architecture [https://www.nec.com/en/global/solutions/hpc/sx/architecture.html].

[79] NEC Vector Engine Models [https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html].

[80] "Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA". Yamada, Yohei, and Shintaro Momose. Proceedings of A Symposium on High Performance Chips, Hot Chips. Vol. 30. 2018.

[81] "Vector Engine Programs: Native, Offloaded, Hybrid". Erich Focht, NEC HPC Europe [https://www.hpc.nec/api/v1/forum/file/download?id=LbGhNY].

[82] "Vector Engine Offloading". Erich Focht, NEC HPC Europe [https://sx-aurora.github.io/posts/AVEO/].

[83] "Vector Host Offloading". Erich Focht, NEC HPC Europe [https://sx-aurora.github.io/posts/VH-call-offloading/].

[84] SX-Aurora TSUBASA - Fortran Compiler User's Guide [https://www.hpc.nec/documents/sdk/pdfs/g2af02e-FortranUsersGuide-021.pdf].

[85] SX-Aurora TSUBASA - C/C++ Compiler User's Guide [https://www.hpc.nec/documents/sdk/pdfs/g2af01e-C++UsersGuide-021.pdf].

[86] NEC MPI [https://www.hpc.nec/documents/mpi/g2am01e-NEC_MPI_User_Guide_en/frame.html].

[87] SX-Aurora TSUBASA - Performance Tuning Guide [https://www.hpc.nec/documents/guide/pdfs/AuroraVE_TuningGuide.pdf].

[88] NEC Numeric Library Collection 2.2.0 User's Guide [https://www.hpc.nec/documents/sdk/SDK_NLC/UsersGuide/main/en/index.html].

[89] "HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems". Jack Dongarra, Michael A. Heroux, Piotr Luszczek. Technical Report, Electrical Engineering and Computer Science Department, Knoxville, Tennessee, UT-EECS-15-736, November 2015.

[90] HPCG Tuning for NEC SX-Aurora TSUBASA [https://sx-aurora.github.io/posts/hpcg-tuning/].

[91] VE monitoring with veperf [https://sx-aurora.github.io/posts/VE-monitoring-veperf/].

[92] veperf memory bandwidth, power and energy monitoring [https://sx-aurora.github.io/posts/veperf-bw-power/].

[93] TensorFlow for SX-Aurora TSUBASA [https://github.com/sx-aurora-dev/tensorflow].

[94] PLASMA SIMULATOR - SX-AURORA TSUBASA A412-8 [https://www.top500.org/system/179871/].

[95] HLRS - SX-Aurora Platform [https://kb.hlrs.de/platforms/index.php/Aurora_Tsubasa].

[96] NEC SX-Aurora TSUBASA A300-8 Rackserver [https://www.nec.com/en/global/solutions/hpc/sx/A300-8.html].

[97] June 2020 HPCG Results [https://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=303].

[98] PROGINF/FTRACE User's Guide [https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf].

[99] NEC Ftrace Viewer User's Guide [https://www.hpc.nec/documents/sdk/pdfs/g2at01e-NEC_Ftrace_Viewer_User_Guide_en.pdf].

[100] Performance Evaluation using TAU Performance System on NEC VE [https://www.nec.com/en/global/solutions/hpc/event/sc20/images/SC20_NEC_Aurora_Forum_Day1_02_Univ_of_Oregon.pdf].

[101] Vftrace [https://github.com/SX-Aurora/Vftrace].

[102] arXiv preprint arXiv:1903.06697. Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on fpgas: Taxonomy, survey, challenges. "Graph processing on fpgas: Taxonomy, survey, challenges". 2019.

[103] 10. 2. IEEE Solid-State Circuits Magazine. 16-29. Stephen M Steve Trimberger. IEEE. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation. “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation”. 2018.

[104] Xilinx Extends Data Center Leadership with New Alveo U280 HBM2 Accelerator Card; Dell EMC First to Qualify Alveo U200 [https://www.xilinx.com/news/press/2018/xilinx-extends-data-center-leadership-with-new-alveo-u280-hbm2-accelerator-card-dell-emc-first-to-qualify-alveo-u200.html].

[105] AMD to Acquire Xilinx, Creating the Industry’s High Performance Computing Leader [https://www.amd.com/en/press-releases/2020-10-27-amd-to-acquire-xilinx-creating-the-industry-s-high-performance-computing].

[106] Intel Should Worry About AMD's Rumored Interest in Xilinx [https://www.nasdaq.com/articles/intel-should-worry-about-amds-rumored-interest-in-xilinx-2020-10-14].

[107] Vitis Unified Software Development Platform 2020.2 Documentation: Kernel Execution [https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/devhostapp.html#xeu1553548473838].

[108] 197-202. Joel Spolsky. Springer. The law of leaky abstractions. The law of leaky abstractions. Joel on Software. 2004.

[109] 99-107. Paolo Gorlani, Tobias Kenter, and Christian Plessl. OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs. “OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs”. 2019 International Conference on Field-Programmable Technology (ICFPT). 2019.

[110] Terasic Homepage [https://www.terasic.com.tw].

[111] HARP2 at Paderborn Center for Parallel Computing [https://wikis.uni-paderborn.de/pc2doc/HARP2].

[112] Intel Labs Academic Compute Environment [https://wiki.intel-research.net/].

[113] Semanticscholar: Seattle, WA, USA. Stephanie Labasan. Energy-efficient and power-constrained techniques for exascale computing. “Energy-efficient and power-constrained techniques for exascale computing”. 2016.

[114] Andrew Moore and Ron Wilson. John Wiley & Sons, Inc. FPGAs for Dummies - 2nd Intel Special Edition. 2017.

[115] Intel Marrying FPGA, Beefy Broadwell for Open Compute Future [https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/].

[116] JUWELS Booster Overview [https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html].

[117] Supercomputer Made in Jülich Setting New Benchmarks [https://www.fz-juelich.de/SharedDocs/Pressemitteilungen/UK/EN/2020/2020-11-16-juwels-booster.html].

[118] JUWELS (Cluster + Booster) - Jülich Wizard for European Leadership Science [https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html].

[119] Easybuild [https://github.com/easybuilders/easybuild].

[120] Dell Intel [https://www.delltechnologies.com/lb-lu/solutions/high-performance-computing/index.htm].

[121] Fast InfiniBand (IB) EDR network [https://clusterdesign.org/fat-trees/].

[122] Products formerly Broadwell [https://ark.intel.com/content/www/us/en/ark/products/codename/38530/broadwell.html].

[123] Products formerly Skylake [https://ark.intel.com/content/www/us/en/ark/products/code-name/37572/skylake.html].

[124] NVIDIA V100 TENSOR CORE GPU [https://www.nvidia.com/en-us/data-center/v100/].

[125] Linux platforms RED HAT ENTERPRISE LINUX [https://www.redhat.com/en/store/linux-platforms].

[126] EasyBuild documentation [https://docs.easybuild.io/en/latest/].

[127] Lmod: A New Environment Module System [https://lmod.readthedocs.io/en/latest/].

[128] SLURM workload manager [https://slurm.schedmd.com/documentation.html].

[129] Intel® Xeon® Processor E5-2697A v4 [https://ark.intel.com/content/www/us/en/ark/products/91768/intel-xeon-processor-e5-2697a-v4-40m-cache-2-60-ghz.html].

[130] Environment Modules open source project [http://modules.sourceforge.net/].

[131] List of supported software [https://docs.easybuild.io/en/latest/version-specific/Supported_software.html].

[132] What is EasyBuild? [https://docs.easybuild.io/en/latest/Introduction.html].

[133] Component versions in foss toolchain [https://docs.easybuild.io/en/master/Common-toolchains.html#component-versions-in-foss-toolchain].

[134] Component versions in intel toolchain [https://docs.easybuild.io/en/master/Common-toolchains.html#component-versions-in-intel-toolchain].

[135] Intel Xeon Scalable Spec Update [https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf].

[136] A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers [http://www.netlib.org/benchmark/hpl/index.html].

[137] What is EasyBuild? [https://www.hpcadvisorycouncil.com/pdf/Intro_to_InfiniBand.pdf].

[138] HPE Slingshot [https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html].

[139] BullSequana eXascale Interconnect V2 [https://atos.net/en/solutions/high-performance-computing-hpc/bxi-bull-exascale-interconnect].

[140] Bisection Bandwidth [https://www-sciencedirect-com.proxy.bnl.lu/topics/computer-science/bisection-bandwidth].

[141] ULHPC Technical Documentation [https://hpc-docs.uni.lu/systems/iris/].

[142] Sebastien Varrette, Emmanuel Kieffer, Frederic Pinel, Ezhilmathi Krishnasamy, Sarah Peter, Hyacinthe Cartiaux, and Xavier Besseron. RESIF 3.0: Toward a Flexible and Automated Management of User Software Environment on HPC facility. “RESIF 3.0: Toward a Flexible and Automated Management of User Software Environment on HPC facility”. ACM Practice and Experience in Advanced Research Computing (PEARC'21). 2021.

[143] ULHPC/sw repository: User Software Management for Uni.lu HPC Facility [https://github.com/ULHPC/sw].

[144] 99-107. A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. “Horovod: fast and easy distributed deep learning in TensorFlow”. arXiv preprint arXiv:1802.05799. 2018.

[145] Sean Mahon, Sebastien Varrette, Valentin Plugaru, Frederic Pinel, and Pascal Bouvry. Performance Analysis of Distributed and Scalable Deep Learning. “Performance Analysis of Distributed and Scalable Deep Learning”. 20th IEEE/ACM Intl. Symp. on Cluster, Cloud and Internet Computing (CCGrid'20). 2020.