POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING

COMPUTER SCIENCE AND ENGINEERING
MASTER'S THESIS

PROGRAMMING DEEPLY HETEROGENEOUS ARCHITECTURES WITH RESOURCE MANAGEMENT SUPPORT

Author:
Manuel Pedrozo Favier
Student ID:
935908

Author:
Tomás Perez Molina
Student ID:
935913

Supervisor:
Prof. Giovanni Agosta

Co-supervisor:
Giuseppe Massari PhD.

A.Y. 2020/2021

Contents

List of Figures

List of Tables

Acknowledgements

To our families, who gave us their unconditional support and made this possible.

To our friends: Marcelo, Pipo, Gaston, Martín, Ricardo, Romi and Augusto, who kept us sane during dark times.

And to our tutors, Giovanni Agosta and Giuseppe Massari, who guided us and were never too busy to answer our questions.

Abstract

THE increasing number of processing-power hungry applications, from traditional physics and financial computing solutions, to the now raising wave of machine and deep learning, has led to a surge in demand for High Performance Computing (HPC) systems. The MANGO project was introduced with the objective of enabling the development of user applications on heterogeneous HPC systems. Up to this point, MANGO offered support for a custom hardware architecture, used to explore manycore architectures for HPC systems. This thesis focuses on a number of additions to the MANGO platform, starting with a Python language API, followed by the Just In Time compilation of computing kernels, and culminating in the introduction of GPU accelerators: highly parallel devices that are commonly found in heterogeneous systems. The incorporation of GPU support resulted in a restructure of the MANGO software stack, with the addition of a Hardware Abstraction Layer and a complete rework of the user-facing module.

Sommario

Il crescente numero di applicazioni ad alte prestazioni, dalle simulazioni fisiche alla finanza computazionale, fino alle applicazioni di intelligenza artificiale ha portato ad un incremento della richiesta di sistemi per il calcolo ad alte prestazioni (HPC). Il progetto MANGO è stato creato per permettere lo sviluppo di applicazioni per sistemi HPC eterogenei. Fino ad ora, MANGO ha offerto supporto per una architettura hardware specializzata, impiegata per l'esplorazione di soluzioni many-core. Questo lavoro di tesi fornisce varie estensioni alla piattaforma MANGO: una API Python, il supporto per la compilazione Just-In-Time (JIT) dei kernel computazionali, e infine il supporto per l'accelerazione su GPU, ovvero il sistema di calcolo parallelo eterogeneo più comune. L'introduzione del supporto per le GPU ha portato ad una profonda ristrutturazione dell'architettura software, con l'introduzione di uno strato di astrazione dell'hardware (HAL) e una riformulazione del modulo in interfaccia.

CHAPTER 1

Introduction

Humankind's push for innovation and problem-solving of the ever extending set of challenges it faces, resulted in a rapid evolution of the technological forefront. The development of digital systems capable of processing levels beyond humanly possible, expanded the realm of possibilities in both the scientific and engineering fields. Applications of this technology, from physics, graphical and meteorological computation, to the now emerging machine learning and deep learning solutions, demand an incredibly high amount of computational resources.

As time passed, resources that were once hardly accessible by end users, due to the elevated cost of acquisition and maintenance, are now made widely available through Cloud Computing. These circumstances called for a rapid evolution of High Performance Computing (HPC), in the search for performance and power-efficiency optimization. The power requirements of HPC centers are increasingly higher, to the point where power supply availability is a main concern in their scaling capabilities. Combined with the growing demand for this technology, it is clear that HPC hardware and software architectures need not only high performance technology solutions, but also power efficient ones. Deeply heterogeneous systems are capable of achieving the nec-

essary performance per watt levels, due to the exploitation of different hardware components specifically designed with certain applications in mind, i.e. parallel computing. Aside from power and performance, another relevant quality is predictability, a requirement of video transcoding and medical imaging applications. Predictability acts as a trade-off with respect to power consumption, due to the reservation of higher amounts of computing resources to ensure the satisfaction of requirements.

The MANGO project was introduced with the objective of enabling the development of user applications on heterogeneous HPC systems. Initially, the MANGO software portion was developed along with a custom hardware architecture capable of fast reconfiguration, allowing the quick and efficient exploration of manycore architectures for HPC systems. MANGO focused on what the original developers called the PPP space: power, performance and predictability. Citing the initial development goals as stated in [1]:

- Simplify the development of parallel applications accessing heterogeneous hardware platforms.
- Enable efficient utilization of the computing resources, with an eye to the application QoS/performance requirements.
- Minimize the power consumption of the HPC infrastructure.

These objectives can only be achieved through the efficient integration and cooperation of an user-friendly but highly functional programming model, an optimized runtime resource management and a hardware abstraction layer capable of expandability.

1.1 Initial software stack

The MANGO software stack, at the beginning of our contribution to the project, looked as shown in figure 1.1.

Starting from the top of the stack, we find the **Libmango** module, in charge of stablishing the programming model for users of the MANGO system by exposing an API that provides access to the functionalities of the underlying layers. It interacts with **BarbequeRTRM**, the resource manager, responsible for scheduling user applications and allocating system resources according to performance requirements, as well as power and thermal information. BarbequeRTRM also monitors running applications to maintain a complete view of the system and collect profiling information.

Next in the stack is the Hardware Abstraction Layer, consisting of the **HN** library

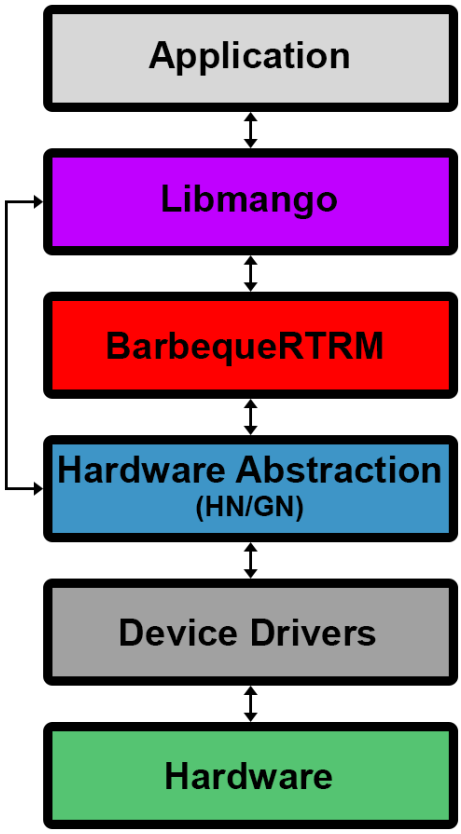


Figure 1.1: *MANGO initial software stack*

and daemon. HN was designed as an abstraction layer for the MANGO custom hardware architecture, supporting PEAK, DCT and nu+ accelerators. To facilitate development and enable running MANGO applications on personal computers, the **GN** emulator was introduced. The GN library follows the same API as HN, which allows them to be used interchangeably, and emulates the presence of hardware accelerators by using instead the host CPU. The bottom-most layers are the **Device Libraries/Drivers**, through which communication with the hardware is stablished and finally the **Hardware** layer, containing the accelerators themselves.

1.2 Thesis Contributions

1.2.1 Python Wrapper

Our contribution to the MANGO project started with the development of a Python Wrapper of the Libmango module, resulting in a fully functional Python API. Up to that point, only the C and C++ programming languages could be used to interact with

the MANGO system, so Python was a perfect fit to expand its compatibility into scripting languages. The project duration was of six months, which also fulfilled the goal of helping us familiarize with the inner workings of the Libmango module. Further details on the Python wrapper implementation are given in the Libmango section of the Architecture chapter, in subsection 3.3.5.

1.2.2 JIT Compilation

As explained in the following chapters, user applications running on the MANGO system consist of a set of tasks (or kernels), that are executed on designated accelerators.

Previous to our contribution, kernels could only be loaded as binaries (compiled kernels ready for execution). Allowing developers to work directly with the kernel's source code, instead of requiring pre-compilation, greatly reduces friction in the iterative process of software design.

To achieve this goal, Just In Time (JIT) compilation of loaded kernels was introduced, enabling developers to load kernels from source code, which are compiled as required by a configurable tool we called the **Dynamic Compiler**.

The first implementation of the Dynamic Compiler was as an extension of the Libmango module, work that took four months to complete. However, due to a restructuring of the MANGO system, it was later moved to the HHAL module.

1.2.3 GPU Support

The next challenge to tackle was the addition of GPU support to the MANGO system, specifically CUDA compatible Nvidia GPUs. With the objective of further fulfilling MANGO's aim at enabling end users to work with highly heterogeneous HPC systems, GPU support is of utmost importance. Apart from their efficient graphics processing capabilities, GPUs are the main accelerator type used in machine learning and deep learning applications due to their highly parallel architecture.

At the time, the HN library played the role of a hardware abstraction layer between hardware-specific libraries and the rest of the MANGO system. Despite being developed with abstraction in mind, the subset of HN-supported hardware accelerators did not merit great levels of generalization, resulting in an abstraction granularity that is too specific for the smooth integration of accelerators such as GPUs.

As a consequence, instead of expanding the HN library, development of the Nvidia Architecture Node (NAN), the platform library in charge of launching kernels on system-

available Nvidia devices, begun as an independent MANGO module. The NAN is further discussed in section 3.6.

An initial integration effort of the NAN with the Libmango module was successful, enabling kernel execution on target GPUs. But a problem with MANGO's structure became increasingly clear. The limited abstraction provided by HN caused the Libmango module to be closely attached to its API and working requirements. This inter-dependency was made evident by the presence of elements like a memory manager tackling virtual-physical address translations in the Libmango module, which was supposed to be architecture agnostic.

A decision was made to restructure the MANGO system with the addition of a new module, the HHAL, which was developed side by side with the NAN.

1.2.4 HHAL

HHAL, which stands for Heterogeneous Hardware Abstraction Layer, was built for the ground up as a solution that enables the easy integration of accelerator architectures to the MANGO system, while freeing Libmango (and other modules) from the inherent complexity of handling multiple architectures. This would also hide the HN library behind the HHAL API.

A detailed explanation of HHAL's implementation is given in section 3.5.

While integration of HHAL with the BarbequeRTRM was a smooth process (covered in section 3.4.2), Libmango required considerable work.

1.2.5 Libmango rework

For Libmango to fit into the new MANGO structure, a rework of most of its components was necessary. The rework was done with the following conditions and objectives in mind:

- Keep the public Libmango API the same, for backwards compatibility.
- Remove architecture specific calculations, now carried out by the HHAL module.
- Simplify bloated data structures, containing information no longer required.

As previously mentioned, the Dynamic Compiler was also moved from Libmango to the HHAL module.

1.2.6 Final Software stack

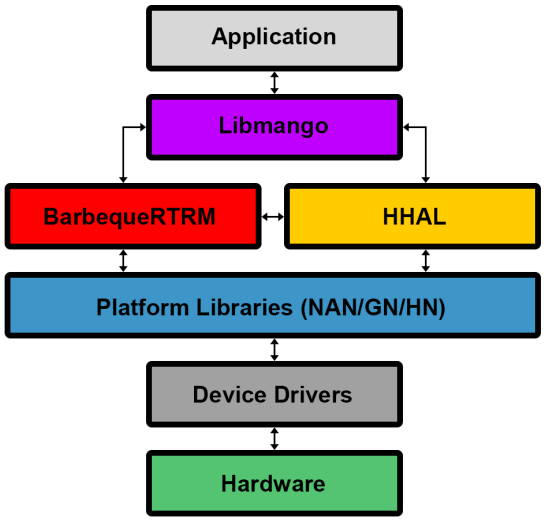


Figure 1.2: *MANGO final software stack*

HHAL and NAN development, Libmango's rework and BarbequeRTRM integration, all took ten months to complete. The implementation work was performed across all modules in an iterative manner, so time dedicated to each module was not measured. The resulting MANGO software stack can be seen in figure 1.2.

1.3 Document Structure

The document is structured as follows:

First, in chapter 2, we present a detailed study of the State of the Art, regarding relevant technologies and their evolution through time. We also expand upon the gaps left by covered technologies, and MANGO's role in closing them.

A thorough Architecture Description is given in chapter 3, going over each module of the MANGO software stack.

In chapter 4, Experimental Results and explanations of each of the conducted experiments are described.

Finally, in chapter 5, we draw the final conclusions, and talk about future work that may be carried out.

CHAPTER 2

State of the Art

In this Chapter we cover the state of the art in heterogenous computing programming models, with a focus on Graphic Processing Units (GPUs).

Today, over six hundred applications utilize GPU acceleration across a broad range of industries including: finance, design for manufacturing/construction, artificial intelligence, medical imaging and more. As such, it was a very important platform to add to the MANGO software stack, and one of the main contributions of our thesis.

Section 2.1 covers the history of advancements in GPGPU technology and is largely based on [3], a talk by Nvidia Software Engineer Mark Harris. This is a very interesting topic and shows the evolution in development using GPUs, culminating in the release of Nvidia's CUDA.

The following sections cover other programming models starting from OpenCL in section 2.2. It being the most well established programming model in GPUs after CUDA, which also provides support for programming in CPUs and FPGAs.

ROCm, AMD's response to the significant adoption of CUDA, is covered in section 2.3. By adopting a very similar API to CUDA, AMD is able to easily port existing

CUDA applications in order to run them on their own GPUs while still being able to compile for Nvidia's.

After that, section 2.4 covers both OpenMP and OpenACC, which adopt a very similar compiler-directive-based approach to programming GPUs.

Finally, sections 2.5 and 2.6 cover SYCL and Kokkos respectively. SYCL being the latest open standard for heterogeneous computing, which aims at providing a higher level approach than OpenCL. On the other hand, Kokkos works towards abstracting away the memory access patterns needed by different architectures in such a way that the same code can exhibit the best performing one depending on the target device.

2.1 The History of GPGPU

2.1.1 Inception

The first documented case of computation on a graphics processor dates to June 1985, when Tim Van Hook implemented the world's first GPU ray-tracing on the Ikonas RDS-3000 [4]. Van Hook followed this up the next year with a paper on solid modelling with the Ikonas [5].

In August 1999, Kedem et al. [6] published a paper where they used experimental graphics engine PixelFlow to perform a brute force attack on Unix passwords. PixelFlow was a heterogeneous parallel machine used for high-speed and high-quality image generation. For their research, PixelFlow was setup with 18 SIMD arrays, each one with 8K processing elements (PE) for a total of 144K (147,456) PEs running at 100Mhz. The machine had some performance problems for this application due to the limited instruction set, which was focused on image computations. Because of this, the results were poorer than expected. It was calculated that the machine would be able to check all lowercase passwords (28.9% of passwords at the time) in 3.19 hours.

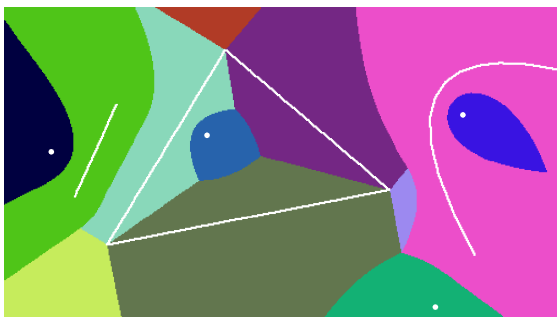


Figure 2.1: *Generalized Voronoi diagram computed interactively on PC (Credit: Hoff et al.)*

Also in 1999, Hoff et al. [7] managed to perform computations of generalized Voronoi diagrams using graphics accelerators, such as the Nvidia TNT2, connected to a PC [3], as opposed to the specialized hardware in the PixelFlow. This was achieved by using the OpenGL API [8]. However, at that time GPUs were not programmable. The hardware exposed what is known as a Fixed Function Pipeline which the user could configure according to their needs. With that configuration, the GPU would execute a series of built in math functions which were focused on rendering, not on computation [9].

2.1.2 Programmable GPUs

Programmable GPUs did not come until 2001, as Nvidia introduced GeForce series 3. This replaced the fixed functions in the previous model with programmable shaders which could be controlled by the developers [10, 11]. These features were, of course, aimed at game developers and 3D designers, but they also allowed for new applications of GPU technology.

Using a GeForce 3, Larsen and McAllister achieved the first matrix multiplication done on a GPU [12]. Their work was done by mapping the matrices into textures that could be manipulated with the OpenGL API. This textures would be transferred to the GPU, rendered and then copied back to CPU memory to be mapped again to a matrix format so results could be read. Incidentally, the resulting "matrix texture" would be shown on screen. There is no explicit mention of the use of programmable shaders in this work, however this would not have changed the study drastically as the fixed functions of the previous model could handle the operations required. The main problem Larsen and McAllister found was the 8 bit fixed point precision and saturation arithmetic used by the hardware. Saturation arithmetic, although very useful in graphics, makes it harder to design a higher precision fixed-point implementation.

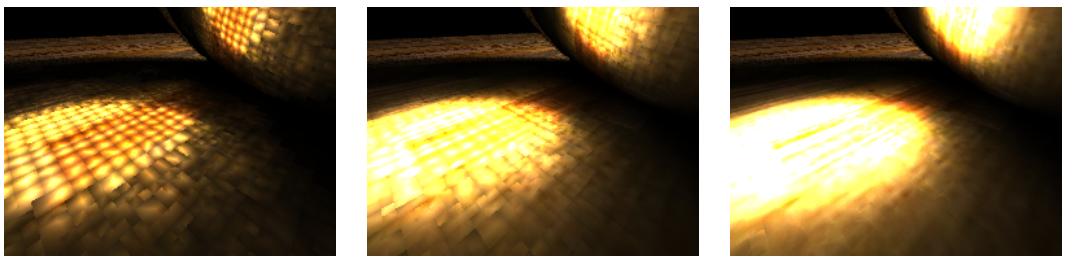
Approximate simulations of natural phenomena were achieved on the GPU by Harris et al. [13]. This included interactive visualizations of convection, reaction-diffusion and boiling. As the end effect of the simulation was to display visuals, this application had the advantage that data did not need to be transferred back to the CPU once results were computed. Again, during these experiments the most problematic aspect was the precision of the fixed-point operations. This contributed to more difficult programmability and arithmetic errors. During this work, the researchers exploited the programming capabilities of the GeForce 3 and, at the time, newly released GeForce 4. ATI had also released a programmable GPU in the form of the Radeon 8500, which promised to add more power to the simulations, however the system was not ported to

it at the time of publication.

In late 2002, after seeing the growing trend in general purpose computation on GPUs, Harris coined the term GPGPU, an acronym for "General Purpose computing on Graphics Processing Units" [3]. GPGPU.org, a website dedicated to news and resources on GPGPU research, would go live August 2003.

DirectX 9 (DX9) introduced the Shader Model 2.0 and with it support for floating-point operations. In July 2002, ATI released the first DirectX 9 capable graphics card in the form of the Radeon 9700 PRO [14]. Nvidia followed up with their own DX9 GPU, the GeForceFX, in January 2003 [15].

DX9 Hardware allowed researchers to implement algorithms which previously could not be ran on the GPU due to lack of floating-point support. One example of this is global illumination. In [16], Purcell et al. achieve interactive frame rates on a GeForce FX 5900 computing global illumination via photon mapping. This is done by utilizing an incremental approach, where increasingly better approximations are rendered until the image converges, the initial approximations can be shown to the user for interactive feedback. On a 160 x 160 window, it is possible to interactively manipulate the camera, scene geometry and light source. Once interaction stops, the illumination converges in one or two seconds, an example can be seen in figure 2.2.



(a) 0.5s (b) 1.0s (c) 2.0s

Figure 2.2: *Lighting approximations over time (Credit: Purcell et al.)*

The main bottleneck that Purcell et al. faced in their paper stemmed from the lack of random access writes. While the original photon mapping algorithm uses a balanced k-d tree, it is not possible to construct one on the GPU due to this limitation. Instead, the researchers had to modify the algorithm in order to account for this, replacing the k-d tree with a uniform grid. To build this grid, they implemented two algorithms, bitonic merge sort and stencil routing. The bitonic sort is computationally less efficient, needing $O(\log^2 n)$ rendering passes. On the other hand, stencil routing can be computed in a single pass but suffers from memory readback performance bottlenecks.

All the kernels were written in Cg, a general purpose language for GPUs released

by Nvidia at the start of 2003 [17]. The design of Cg was inspired by the C language to provide a high level language that is still close to the underlying hardware. The C syntax served as a starting point which was then extended and modified as necessary to support GPU architectures effectively. The general purpose nature of the language allows the programmer to use very similar code to program the vertex and fragment stages of the rendering pipeline. This also simplifies programming for GPGPU applications, an aspect that was taken into account when designing the language due to the rising popularity of the field.

2.1.3 Brook

Languages like Cg, Microsoft's HLSL and OpenGL's GLslang allowed for shaders to be written in C-like syntax. However, they still required the programmer to express GPU applications in terms of graphics primitives and to use the existing graphics APIs to control the rest of the graphics pipeline, such as memory allocation and loading programs. In August 2004 researchers at Stanford University presented Brook [18], a programming environment that provides developers with a view of the GPU as a streaming coprocessor.

Instead of working with textures and shaders, the Brook language allows the programmer to think in terms of streams and kernels. A stream is a collection of records (elements) and is denoted by angle-brackets, i.e. `float x<100>`. Access to streams is limited to kernels and the `streamRead` and `streamWrite` operators, which transfer memory between memory and streams. A kernel is a function that performs parallel operations over one or more streams. Calling a kernel on a stream performs an implicit loop over the elements of the stream, invoking the body of the kernel for each element. An example Brook snippet can be seen in listing 2.1.

A kernel accepts different types of arguments:

- Input streams that contain read-only data for kernel processing.
- Output streams (marked with the `out` keyword) that store the result of a kernel computation.
- Gather streams which are declared as a C array with brackets, i.e. `gather[]`. A gather stream allows for arbitrary indexing of its elements.
- Non-stream arguments, which are read-only constants.

Due to the same GPU limitations experienced by Purcell et al., Brook does not

Listing 2.1: *Brook saxpy example*

```
kernel void saxpy(float a, float4 x<>, float4 y<>, out float4 z<>) {
    z = a*x + y;
}

void main(void) {
    float a;
    float4 X[100], Y[100], Z[100];
    float4 x<100>, y<100>, z<100>;
    // Omitted: Initialize a, X, Y
    streamRead(x, X);     // copy data from mem to stream
    streamRead(y, Y);
    saxpy(a, x, y, z);    // execute kernel on all elements
    streamWrite(z, Z);    // copy data from stream to mem
}
```

provide arbitrary writes, only arbitrary reads with the gather streams.

The Brook compilation and runtime system maps the language onto existing programmable GPU APIs, including OpenGL and DirectX. The system consists of two components: `brcc`, a source-to-source compiler and the Brook Runtime (BRT), a library that provides runtime support for kernel execution. `brcc` maps Brook kernels into Cg shaders which are then translated into GPU assembly by vendor-provided shader compilers. Additionally, it emits C++ code which uses BRT to invoke the kernels.

2.1.4 The Unified Shader Model

In November 2005, Microsoft launched the XBOX 360 console. A noteworthy aspect of this launch is that the console used the first unified shader architecture GPU on the market, the ATI Xenos [19]. Previously, GPUs had different processing units which either handled vertex or fragment shader operations. In the unified shader model, there is a single type of unit, called shader core, which can handle both type of operations. The main selling point of this change is that greater flexibility allowed for all the units to be used during rendering, no matter the type of workload. With the classical fixed shader model, heavy polygon scenes would leave the fragment units idle, while heavy pixel scenes would underutilize the vertex units. This issue is illustrated in figure 2.3. Figure 2.4 shows how the unified shader model is able to better utilize the available resources.

While ATI produced the first unified shader GPU for the XBOX 360, Nvidia was the one to release the model in PCs with the GeForce 8800 in November 2006. DirectX 10 had introduced Shader Model 4.0 which included a unified shader instruction set. Even

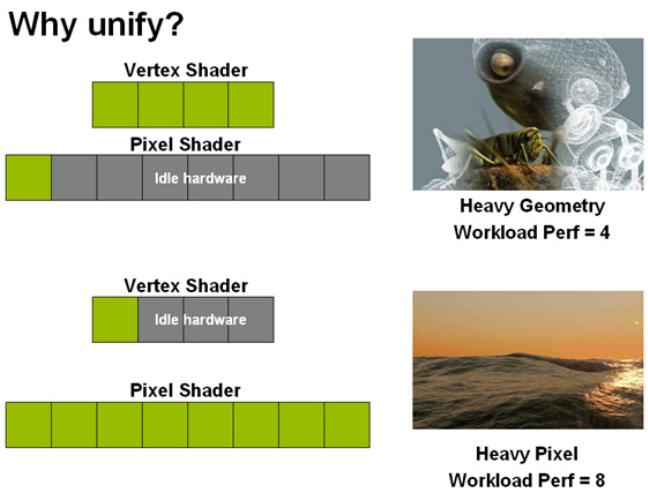


Figure 2.3: *Fixed shader model performance characteristics (Credit: Nvidia)*

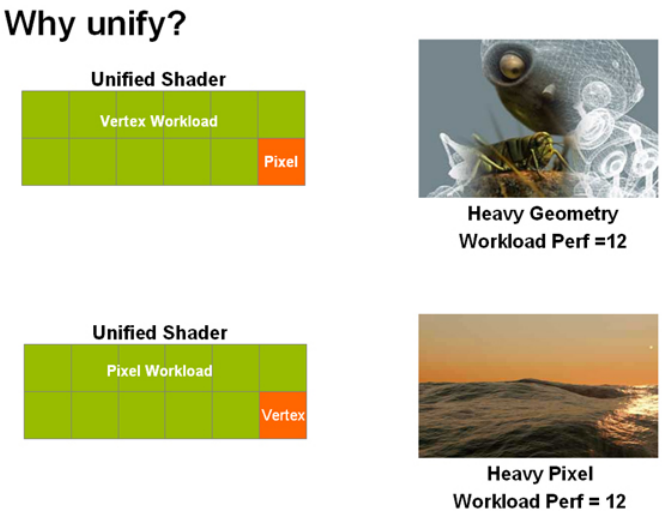


Figure 2.4: *Unified shader performance characteristics (Credit: Nvidia)*

though a unified architecture was not a requirement to use DirectX 10, it provided better efficiency, load-balancing and power utilization [20]. A block diagram of the GeForce 8800 can be seen in figure 2.5. The streaming processors (SP) marked in green are the units in charge of all the shader processing (the shader cores).

2.1.5 AMD Close To The Metal

ATI (from here on out referred to as AMD due to their acquisition in 2006) was also the first vendor to release direct support for GPGPU with its CTM or "Close To The Metal" system in late 2006. CTM provides raw assembly level access with its hardware abstraction layer (HAL). The compute abstraction layer (CAL) adds higher level

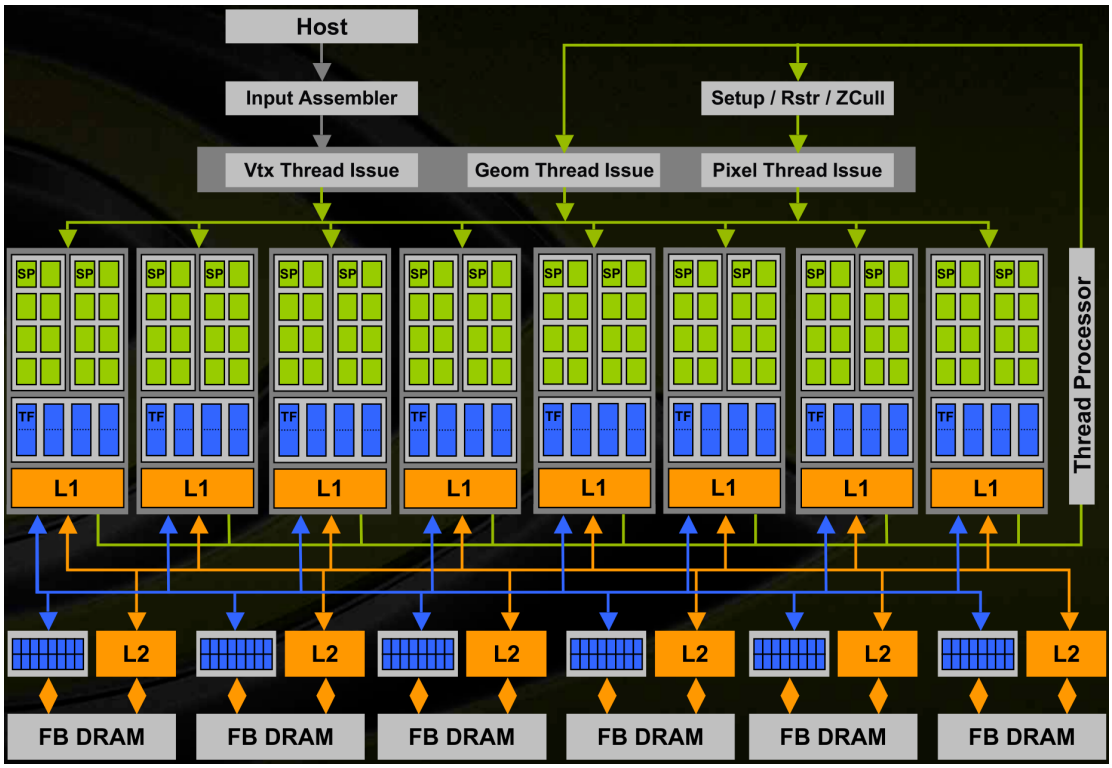


Figure 2.5: *GeForce 8800 GTX Block Diagram (Credit: Nvidia)*

constructs and a C API, however this only covers context, memory management and kernel execution, the kernel itself must still be written in a low level AMD intermediate representation [21]. For higher level programming, AMD also supported compilation of Brook programs directly to the hardware [22].

By providing a first party computing model, CTM eliminates the need for developers to work with graphics APIs and deal with a rendering pipeline. Instead of having to adapt algorithms work with textures, vertices, pixels and shaders, the developer can perform computation by binding memory as inputs and outputs to the stream processors directly. This includes Brook, which no longer has the requirement to work on top of a graphics API backend.

2.1.6 Nvidia CUDA

In June of 2007, Nvidia introduced CUDA [23]. The significant redesign that came with the adoption of unified shaders for the GeForce 8800 GTX, as well as the flexibility achieved by the final model, allowed Nvidia to develop a hardware and software solution for data-intensive computing.

The CUDA C++ language is a minimal extension over C++ adding parallelism fea-

tures. As opposed to AMD CTM, which combined the CAL C API with a low level kernel language, this programming model is single source. CPU (host) code and GPU (kernel) code are written in the same language and can be contained in the same file. When launching a kernel in CUDA, the user must provide an *execution configuration* with a custom <<< ... >>> syntax. The most important aspect of this configuration is the dimensions of the *grid* and *thread blocks*.

A thread block is a group of threads that is executed in a single streaming multiprocessor (SM), which is a group of stream processors (SP). As its definition indicates, a thread block must fit in a single SM and thus is limited by the hardware. Currently, this limits a thread block to contain a maximum of 1,024 threads. The dimensions of a thread block is given by a `dim3` which is a 3-dimensional vector, allowing the programmer to distribute the threads in a 3D configuration. Using 1D, 2D or 3D configurations is purely for convenience of the developer. They can use the one that best suits the kernel to run.

A grid is then a group of thread blocks. The maximum amount of blocks that can run in parallel is limited by the amount of SMs in a particular GPU, however this does not limit the dimensions of the grid. Due to memory latency, it is better to launch a large grid of blocks at the same time. Having a lot of blocks at its disposal allows the GPU to switch the blocks being executed depending on the contents of the memory. If a given block needs to load a piece of data from memory as it is not currently in the GPU registers, it can be swapped out until the data is loaded, hiding the overhead. Grid dimensions are also indicated with a `dim3`, for the same reasons as the blocks.

When running a kernel, the CUDA runtime assigns a unique ID to each execution of the kernel, according to its theoretical place in the grid and the block. On the device side, the programmer can query the block and thread ids as well as the block and thread dimensions in order to handle different sections of the data.

An example of CUDA code can be seen in listing 2.2.

When compared to AMD's offering, CUDA is a higher level interface than CAL, but it also provides more hardware access than Brook. In exchange for requiring more hardware knowledge, CUDA exposes multiple levels of memory hierarchy, per-thread registers, fast shared memory between threads in a block, board memory and host memory. In addition, while Brook only exposes a single dimension of parallelism, data parallelism via streaming, CUDA provides data parallelism and multithreading. CUDA kernels are also more flexible, as they allow the use of pointers, arbitrary writes and thread synchronization between threads in a single block [22].

Listing 2.2: *CUDA saxpy example*

```
__global__ void saxpy(int n, float a, float *x, float *y, float *z) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) z[i] = a * x[i] + y[i];
}

int main(void) {
  float a;
  float X[100], Y[100], Z[100];
  float *d_x, *d_y, *d_z;

  // Omitted: Initialize a, X, Y

  cudaMalloc(&d_x, 100 * sizeof(float));
  cudaMalloc(&d_y, 100 * sizeof(float));
  cudaMalloc(&d_z, 100 * sizeof(float));

  cudaMemcpy(d_x, X, 100 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, Y, 100 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_z, Z, 100 * sizeof(float), cudaMemcpyHostToDevice);

  // Launch kernel in a single block of 256 threads
  saxpy<<<1, 256>>>(100, a, d_x, d_y, d_z);

  cudaMemcpy(Z, d_z, 100 * sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_z);
}
```

Overall, CUDA's advantages over AMD's CTM gave Nvidia the upper hand in the GPGPU field and it is still in use to this day. By contrast, in 2008, AMD's CTO of graphics announced that the company was shifting its focus away from CTM and into the upcoming OpenCL standard [24], detailed in the following section.

2.2 OpenCL

Released in December 2008, Open Compute Language or OpenCL [25] is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. It provides a framework for parallel programming and includes a language, API, libraries and a runtime system to support software development. By leveraging OpenCL, an application can use a host and one or more OpenCL devices as a single heterogeneous parallel computer system.

The framework is comprised by the following components:

- **Platform layer**: Allows the host program to create contexts and discover OpenCL devices and their capabilities.
- **Runtime**: Allows the host program to manipulate contexts one they have been created.
- **Compiler**: Creates program executables that contain OpenCL kernels. Depending on the capabilities of a device, the compiler may build executables from either OpenCL C source strings, the SPIR-V intermediate language, or device-specific program binary objects. Some implementations may support other kernel languages or intermediate languages.

Unlike CUDA, where host and device code live in the same file and are written in the same language, OpenCL introduces a kernel specific language called OpenCL C [26] which runs on the device. Meanwhile, on the host side, OpenCL exposes functionality through a C library.

OpenCL C is based on the *ISO/IEC 9899:1999 - Programming languages - C* specification (also referred to as C99) [27], with the addition of some *ISO/IEC 9899:2011 - Information technology - Programming languages - C* specification (also referred to as C11) [28] features, plus some extensions and restrictions to support parallel kernels.

This dedicated kernel language allows the developer to write a single kernel code base and execute it in different devices. This ensures the *functional* portability of code across devices, eliminating the need for applications to be re-coded on a per-device or

Listing 2.3: *OpenCL saxpy example (kernel)*

```
__kernel void saxpy_kernel(
                    float a,
                    __global float *X,
                    __global float *Y,
                    __global float *Z)
{
    //Get the index of the work-item
    int index = get_global_id(0);
    Z[index] = a * X[index] + Y[index];
}
```

per-programming toolkit basis [29]. The same code can be distributed to any OpenCL device and is compiled by a device specific compiler at runtime (device specific binaries can also be distributed). However, portability issues may still arise if the hardware supports different versions of the standard. In addition, there can also be issues in terms of performance portability due to architecture differences and compiler optimizations available on each platform [29–31]. For maximum performance, some tweaking of the source code may still be necessary depending on which device is being targeted [32,33].

SPIR-V is an intermediate language which is also supported as an input language for OpenCL. Instead of distributing and ingesting OpenCL C in the host, developers can precompile their kernels into SPIR-V, allowing for faster kernel load times and avoiding directly exposed source code [34]. While SPIR-V is also supported by the Vulkan [35] graphics API, it uses a different execution mode of the language (**GLCompute** versus **Kernel** for OpenCL), so compute codes are not interchangeable [36]. Another caveat is that SPIR-V support is only mandatory for OpenCL 2.1 and 2.2 devices, support was made optional for OpenCL 3.0 devices [25]. This reduces the amount of devices which will guarantee SPIR-V compatibility, reducing the appeal of SPIR-V only distributions.

An example of OpenCL code can be seen in listings 2.3 and 2.4. Listing 2.3 shows a kernel written in OpenCL C which could be loaded as a file or a source string. Listing 2.4 shows the host side code, which is considerably abbreviated. OpenCL is highly verbose in order to provide low level control over the kernel execution. The programmer must manually choose on which of the available devices to run a particular kernel by creating different *command queues* onto which they can enqueue different operations.

Seventeen different companies (including Apple, Intel, AMD and Nvidia) distribute products conforming to the OpenCL standard [37]. This allows an OpenCL kernel to run on the majority of hardware available on the market, including CPUs, GPUs, FPGAs and more. As is illustrated in figure 2.6, a great number of applications use OpenCL as a backend to enable parallelism on this hardware.

Listing 2.4: *OpenCL saxpy example (host)*

```
// OpenCL C kernel as source string.
const char *saxpy_kernel = "...kernel...";

int main(void) {
  float a;
  float X[100], Y[100], Z[100];

  // Omitted: Initialize a, X, Y

  cl_uint num_devices;
  cl_device_id *device_list;
  // Omitted: Get platform and device information
  // Omitted: Get list of devices and choose device to run on

  cl_context context =
    clCreateContext(num_devices, device_list, ...);

  // Create a command queue
  cl_command_queue q =
    clCreateCommandQueue(context, device_list[0], ...);

  // Create memory buffers on the device for each vector
  cl_mem X_clmem =
    clCreateBuffer(context, CL_MEM_READ_ONLY, 100 * sizeof(float), ...);
  cl_mem Y_clmem =
    clCreateBuffer(context, CL_MEM_READ_ONLY, 100 * sizeof(float), ...);
  cl_mem Z_clmem =
    clCreateBuffer(context, CL_MEM_WRITE_ONLY, 100 * sizeof(float), ...);

  // Copy the Buffer X and Y to the device
  clEnqueueWriteBuffer(q, X_clmem, 100 * sizeof(float), A, ...);
  clEnqueueWriteBuffer(q, Y_clmem, 100 * sizeof(float), B, ...);

  cl_kernel kernel;
  // Omitted: Build program from source and create kernel

  // Set the arguments of the kernel
  clSetKernelArg(kernel, 0, sizeof(float), (void *) &a);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &X_clmem);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &Y_clmem);
  clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *) &Z_clmem);

  // Execute the OpenCL kernel on the vector
  size_t global_size = 100, local_size = 64;
  clEnqueueNDRangeKernel(q, kernel, &global_size, &local_size, ...);

  // Read the cl memory C_clmem on device to the host variable Z
  clEnqueueReadBuffer(q, Z_clmem, 100 * sizeof(float), C, ...);

  // Clean up and wait for all the comands to complete.
  clFlush(q);
  clFinish(q);

  // Omitted: Release all OpenCL allocated objects and host buffers
}
```

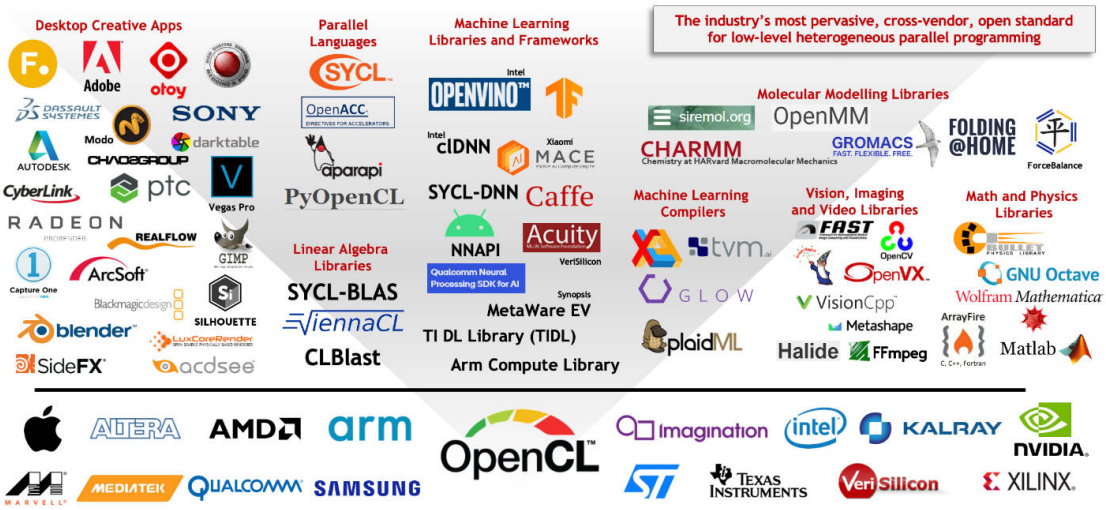


Figure 2.6: *OpenCL accelerated applications (Credit: Khronos Group)*

2.2.1 EngineCL

EngineCL [38], introduces a new object-oriented API on top of OpenCL. The EngineCL class provides a higher level view of the OpenCL context and management of the available devices. The engine in turn uses a Program object which internally manages all the data transfers between the device buffers, the user only needs to provide host input and output buffers, as well as the kernel arguments in order to begin execution. In addition, multiple devices can be used during a single run, the scheduling of which is handled by a Scheduler object. Different scheduling strategies are tested by the paper, with the best results achieved by the HGuided algorithm. HGuided is a dynamic algorithm which starts by assigning big block sizes to all devices and reducing the size of subsequent ones as the execution progresses. This reduces data transfer and synchronization overhead while allowing devices to finish simultaneously towards the end of the execution.

2.2.2 FluidiCL

FluidiCL [39] maintains the OpenCL API but implements it in such a way that the user can treat multiple devices as a single entity. Thus, it is very easy to adapt an existing OpenCL application to run using FluidiCL, as all function calls can remain unchanged. The paper considers the implementation running on an experimental system with a single GPU and CPU. At the time of setup, both kernel compilation and buffer writes are broadcasted to both devices. That is, the kernel is compiled for both and, likewise, the input data is transferred to both of their buffers. When the execution starts, the

GPU starts running the kernel with a decreasing order of work-group IDs, meanwhile the CPU executes smaller sub-kernels in increasing order of work-group IDs. At some point, when the work-groups IDs handled by both cross over, the work is finished and the results are merged on the GPU.

2.3 AMD ROCm

Initially announced in November 2015 as the "Boltzmann Initiative", AMD ROCm is an open-source software development platform for HPC GPU computing [40]. It is AMD's latest competitor to Nvidia's CUDA.

Since shifting their focus away from CTM in 2008 and mainly supporting the OpenCL standard, AMD still lost a significant market share to Nvidia in the GPGPU market [41]. Developers preferred the CUDA approach which, although constrained them to Nvidia products, provided a more convenient programming experience by following a single source model.

In order to further tap into the GPGPU market, mostly filled by CUDA developers, AMD implemented ROCm and, being more specific, HIP. HIP is a C++ Runtime API and kernel language very similar to CUDA that is portable across Nvidia and AMD GPUs. Porting a CUDA application to HIP does not involve much more than replacing all instances of **cuda** in the file with **hip**. AMD also provides the *Hipify* tool, which performs these operations automatically.

HIP code can be compiled for either AMD or Nvidia GPUs. Compilation was previously handled by `hcc` and `nvcc` respectively, but these tools have been replaced in ROCm v3.5 by the HIP-Clang compiler, which can compile code for both vendors.

When compared to OpenCL, HIP, like CUDA, has the advantage of a single source C++ programming model. Host and device code can live in the same file and use most of C++'s feature set including lambdas, templates and classes. Thus, porting CUDA code is significantly easier for HIP as mentioned previously. Porting from CUDA to OpenCL not only involves separating device code from host code, it also requires translating this device code from C++ to OpenCL C and modifying all CUDA function calls and keywords to their OpenCL counterparts.

As a case study, in SC16 Ben Sander from AMD made a presentation showing the work required to port CAFFE, a popular machine learning framework with 50,000 lines of CUDA code from CUDA to HIP [42]. As shown in figure 2.7, the HIP port of CAFFE only required changing 907 lines of code (LOC), out of which 688 (almost

76%) was done automatically. Manual changes to the code took a single developer less than a week to complete. By contrast the OpenCL implementation of CAFFE required changing more than 30,000 LOC.

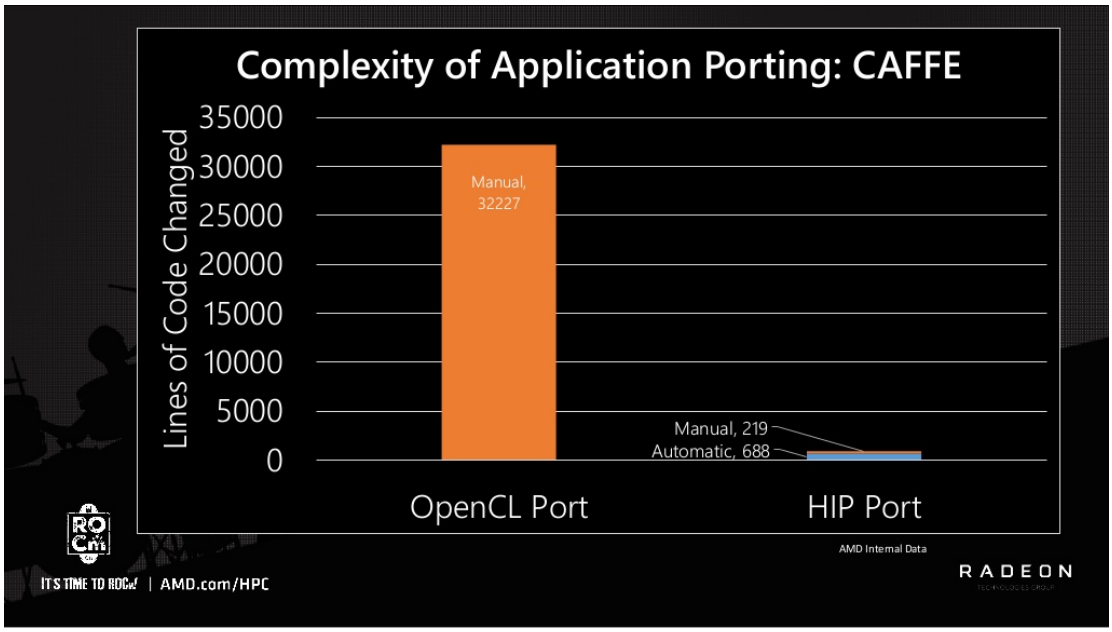


Figure 2.7: *Complexity of porting CAFFE, OpenCL vs HIP (Credit: AMD)*

[43] also presents a case of porting an application from CUDA to HIP. Tsai et al. found the process very smooth and were able to automatize most of the work by extending the initial Hipify script to their needs. Although HIP allowed them to run their application on both AMD and Nvidia, they still preferred to keep native support for CUDA and maintain two versions at the same time. This setup allowed them to take advantage of HIP's similarity to CUDA by sharing a third of the code base between both implementations. In the cases where they used CUDA specific functionality, they were able to replicate it in HIP and use a common API to call the appropriate implementation depending on the programming model used. Finally, they also included experiments to show the performance of HIP code compared to native CUDA on Nvidia GPUs. These tests showed that the overhead of HIP over CUDA is negligible, with 50% of the test cases showing less than 3% performance difference and 90% of the test cases showing less than 10% performance difference.

2.4 OpenMP & OpenACC

In this section we will tackle OpenMP and OpenACC in conjunction as they take very similar approaches.

Listing 2.5: *OpenMP saxpy example*

```
int main(void) {
    float a;
    float x[100], y[100], z[100];

    // Omitted: Initialize a, x, y

    #pragma omp target map(to:a, x[0:100], y[0:100]) map(from:z[0:100])
    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        z[i] = a * x[i] + y[i];
    }
}
```

Both projects are composed of a library and set of compiler directives that provide a model for parallel programming across different architectures. Support is provided for the C, C++ and Fortran languages. The directives extend the languages with useful constructs for parallelizing applications. Further control of the runtime environment is possible through the library [44, 45].

OpenMP and OpenACC allow for quick adaptation of existing single threaded code into a parallel execution model. This work requires a compiler which supports the standard, meaning that it is able to handle the directives and generate multithreaded code automatically.

An example of offloaded OpenMP code can be seen in listing 2.5. The code looks like a regular serial implementation of SAXPY for a general purpose processor, except for the two `#pragmas`. The first pragma instructs OpenMP to offload the following code onto an accelerator, and specifies which variables must be copied **to** the accelerator when entering the scope (a, x and y) and which must be copied **from** the accelerator when leaving the scope (only z). The second pragma indicates that the for loop can be executed in parallel, that way the compiler can be sure that all operations are independent and capable of running in separate threads.

Up until version 4.0, OpenMP only allowed this code to be compiled for and executed on the CPU. Version 4.0 (2013) introduced offloading of parallel code onto other devices, like GPUs or FPGAs [46]. Meanwhile, OpenACC focused on heterogeneous computing and accelerator offloading from the start [47], also treating the multicore CPU itself as a device.

[48] provides a comparison of both programming models in terms of programmability and expressiveness. Here the authors denote the differences between OpenMP and OpenACC when implementing common parallel programming patterns targeting

accelerators. Overall, the resulting code and directives used are mostly equivalent, with OpenACC having a slight advantage thanks to providing accelerator support since its inception. In terms of programmer effort, there is no significant difference. In terms of performance however, [49] shows that the code generated by OpenACC is able to utilize more memory bandwidth and thus perform better than OpenMP, specially when using a naïve approach. Still, both approaches fall behind a pure CUDA kernel.

Finally, the possibility to use both models at the same time exists. Works like [50] exploit parallelism on the CPU with OpenMP to schedule code to run on multiple GPUs. [51] also leverages this hybrid approach to run kernels which are more GPU friendly on the GPU using OpenACC while running less friendly kernels with OpenMP CPU parallelization.

2.5 SYCL

SYCL [52] is a C++ programming model for heterogeneous computing. It builds on the underlying concepts, portability and flexibility of parallel APIs or standards like OpenCL while adding the ease of use and flexibility of single-source C++. Initially, SYCL was released as OpenCL SYCL with a provisional specification in 2014, and acted as a higher level API to interact with OpenCL devices. Since the newest SYCL 2020 standard, the model has become more generalized, making OpenCL just one of the different potential programming models SYCL can be based upon [53].

In SYCL, device and host code live on the same file and can be written in C++ according to the C++17 standard (ISO/IEC 14882:2017 Programming languages — C++) [54] and also the newest C++20 standard (ISO/IEC 14882:2020 Programming languages — C++) [55]. For compatibility reasons, the entire set of C++ features is not available in device code. In particular, SYCL device code does not support virtual function calls, function pointers, exceptions, runtime type information or dynamic memory allocation. This still leaves a big portion of the standard which is compatible with host and device code alike, allowing the reuse of types, library code, templates and abstractions. In addition, the programmer does not need to switch between languages depending which part of the code base they are modifying. It is also important to note that, as long as there is no dependencies created with the underlying SYCL implementation, a standard C++ compiler can compile SYCL programs to run directly on the host CPU, without any external accelerator.

To facilitate integration, SYCL is designed to allow each source file to be passed through multiple different compilers, the outputs of which are combined into a single

Listing 2.6: *SYCL saxpy example*

```
using namespace sycl;
float A;
float h_X[100], h_Y[100], h_Z[100];

// Omitted: Initialize A, h_X, h_Y

queue q; // Queue to enqueue work to the default device

// Wrap the arrays in buffers
buffer<float,1> d_X { h_X, range<1>(100) };
buffer<float,1> d_Y { h_Y, range<1>(100) };
buffer<float,1> d_Z { h_Z, range<1>(100) };

q.submit([&](handler& h) {
    auto X = d_X.get_access<access::mode::read>(h);
    auto Y = d_Y.get_access<access::mode::read>(h);
    auto Z = d_Z.get_access<access::mode::read_write>(h);

    // Enqueue a parallel_for task with 100 work-items executing the saxpy
       kernel
    h.parallel_for(100, [=] (id<1> idx) {
        Z[idx] = A * X[idx] + Y[idx];
    });
});
q.wait();
```

source file. The programmer can then add SYCL code to an existing project and continue using their preferred host compiler while the SYCL tools handle compilation for the device code.

An example SYCL code snippet can be seen in listing 2.6.

The code within the the lambda argument to the `parallel_for` is the device kernel. This code will be compiled using the device compiler and executed on the device.

SYCL can be laid out on top of multiple backends. A backend exposes one or multiple SYCL platforms (collections of devices). For example, implementations can expose an OpenCL backend to give access to OpenCL devices, or a CUDA backend to give access to Nvidia GPUs. Apart from the generic API that all backends must implement in order to provide the basic device functionality, each backend can expose their specific features through interoperability headers to provide the developer with more control at the expense of portability.

When building a SYCL application, the user can choose which backends to use. This is the set of *active backends* for the application. The application can then be run on any host platform that supports at least one of the active backends. The subset of

active backends which are supported by the host platform at runtime are called the *available backends*.

A diagram with the SYCL implementations in development and their provided backends can be seen in figure 2.8. As shown, the available SYCL implementations cover a wide range of hardware. As long as the programmer does not use any backend-specific feature, their application can be executed in any available implementation without modifications.

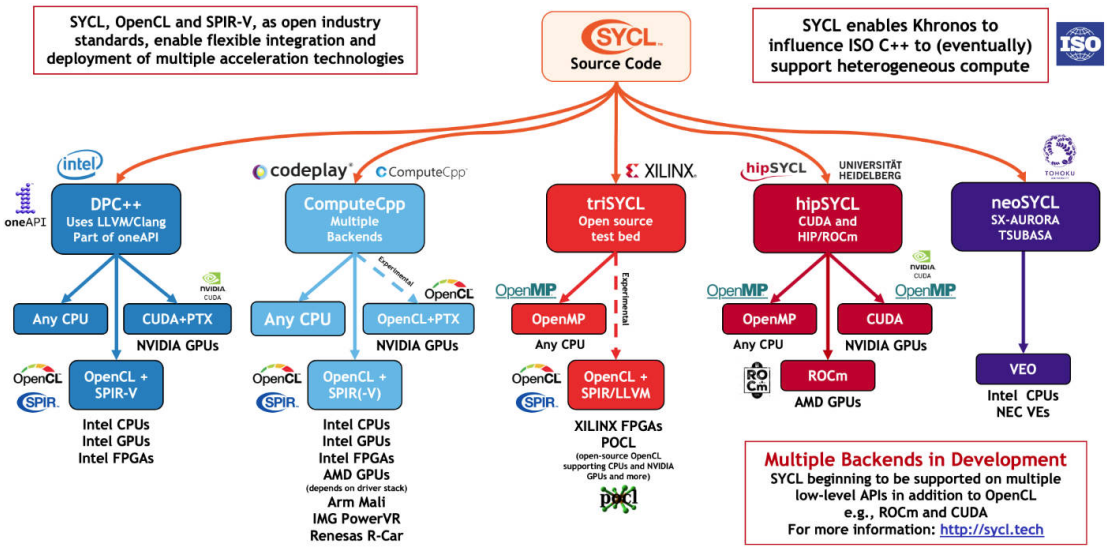


Figure 2.8: *SYCL implementations in development (Credit: Khronos Group)*

In [56] Reguly studies the performance of multiple different programming models on both CPUs and GPUs. These programming models include: OpenMP, OpenACC, CUDA, SYCL and Kokkos (covered in the following section). In the paper, two multi-material simulation problems are used as benchmarks. In multi-material simulations, each cell in the simulation domain can have one or multiple different materials. Given these multi-material cells, the three algorithms used to solve the proposed benchmark problems compute the density and pressure of each material over each cell and their neighboring cells. Performance is measured in terms of fraction of peak performance, where peak performance is the theoretical maximum memory bandwidth possible for each device. In GPUs, SYCL (in particular the hipSYCL implementation) is able to achieve very similar results to pure CUDA kernels, the best performing model, in both problems. The situation is different when looking at CPUs, where SYCL implementations are 5-15% slower than OpenMP ones.

[57] presents a more recent survey of SYCL performance. It evaluates the memory bandwidth achieved by three different applications:

- BabelStream: A popular measure of memory transfer speeds to and from global device memory.
- Heat: 5-point stencil which computes the weighted average between a cell and its four neighbors.
- CloverLeaf: Largest application in the group, with close to 8,000 lines of code. It is composed of around 11 routines, each one performing either point-wise or stencil updates over a 2D grid.

All these applications are memory bandwidth bound, as it is common in HPC applications today. In this study, SYCL proved to incur in very little overhead over the underlying implementations. Any performance problems shown in the applications were also reproduced in the direct implementations for the underlying programming models, so they are not SYCL level issues. What this study shows is that there is a clear need for better vendor support. All GPU implementations rely on open source projects which do not offer commercial support from the hardware vendors. Also, as SYCL is currently mostly built on top of other programming models, it relies on the maturity and support of their implementations as well.

2.5.1 Celerity

Celerity [58] is an API for programming distributed memory accelerator clusters. It is built on top of SYCL and extends its programming model, allowing the user to distribute the work and data across multiple nodes in an automatic way. Celerity also leverages MPI (Message Passing Interface) [59] to enable inter-node communication.

The Celerity extensions act as a wrapper over the SYCL runtime, handling inter-node communication and scheduling, while each individual node executes the SYCL kernel in parallel. In order to enable this, Celerity introduces a global distributed queue, replacing the device local queues of SYCL. It also adds the requirement to specify the data access of each kernel, for reading and writing to the buffers. This allows Celerity to know beforehand which pieces of data need to be present at each node when distributing the work, and also to reconstruct the output buffer when the execution finishes.

Celerity's execution model is arranged in a master/worker fashion. Each worker node encapsulates the available accelerator hardware and receives commands coming from a master node, which is responsible for scheduling the work. In order to construct a dependency graph of the kernels to execute, the Celerity runtime executes the programs twice. First, in the *pre-pass*, information about the kernel is collected, this

includes buffer accesses, how these accesses are mapped into outputs, and the global size of the kernel. With this data, the master can construct the graph and generate commands to distribute to the workers. These commands not only contain buffer data and the kernel function, but also dependency information. This allows the workers to perform peer-to-peer communication during the *live-pass* as necessary and start execution as soon as their dependencies are fulfilled, instead of requiring communication with the master at each instance. This means that Celerity does not induce any extra bandwidth overhead compared to a fully decentralized approach.

2.6 Kokkos

Kokkos [60] is a C++ library that enables applications and domain libraries to achieve performance portability on manycore architectures by unifying abstractions for both fine-grain data parallelism and memory access patterns.

As a library-only solution, Kokkos does not require external tools such as a specialized compiler or even compiler extensions. Kokkos can be linked as a regular C++ library with CMake.

One of the main features provided by this programming model, and the one that distinguishes it from the rest, are `View` objects. `Views` are used as storage for kernel computations. Instead of using raw pointers and arrays, where the indexing is done as a simple memory reference at a given offset, `Views` change their memory layout depending on the device the kernel is running on. Operations on a `View` are not affected at all by this change, allowing kernels to be portable across devices while also providing the optimal memory access pattern on each architecture.

To implement parallelism, Kokkos provides a set of functions that cover different *Execution Patterns*. These include parallel loops with `parallel_for`, reductions with `parallel_reduce`, scans with `parallel_scan` and spawning of individual tasks. These patterns can also be nested, for example one can execute a `parallel_reduce` inside a `parallel_for` or vice versa.

Kokkos is built on top of multiple backends. This means that it is not tied to any particular underlying implementation and can use the best performing programming model available on a particular platform. These backends include OpenMP, CUDA, ROCm, HPX [61], SYCL and pthreads.

In terms of performance, the initial implementation presented in [60] showed it was capable of achieving at least 90% of the performance of the architecture specific, op-

timized variants of the same benchmarks (i.e. comparing Kokkos against OpenMP for CPUs and CUDA for GPUs). [56] also evaluates the performance of Kokkos. As mentioned in the SYCL section, this paper uses two different multi-material problems as benchmarks. The overall performance of Kokkos is 6% lower for the first problem and 22% lower for the second problem. While CUDA and OpenMP implementations have the advantage, they also require different codebases, while Kokkos is completely portable across devices.

2.7 MANGO V1

The first version of the MANGO project, here referenced to as MANGO V1, presents a programming model that simplifies the development of parallel applications exploiting deeply heterogeneous architectures.

Providing a C and a C++ API, what separates MANGO from the previously mentioned alternatives is the inclusion of a resource manager (BarbequeRTRM, further detailed in section 3.4.1). The resource manager is capable of scheduling user's tasks over the system available accelerators according to user requirements and the current state of the system. A custom hardware architecture, consisting on multiple clusters of FPGA accelerators, and capable of fast reconfiguration, was developed along the MANGO software stack and utilized for the exploration of manycore architectures for HPC systems. As a result, MANGO V1 offers support for PEAK, nu+ and DCT accelerators, while also providing CPU emulation of these (GN), for development and testing purposes.

Listing 2.7: *MANGO GN saxpy example (kernel)*

```
#pragma mango_kernel
void saxpy(float a, float *x, float *y, float *out, int n) {
    for (int i=0; i < n; i++) {
      out[i] = a * x[i] + y[i];
    }
}
```

Listing 2.8: *MANGO saxpy example (host)*

```
#define KERNEL 1
#define B1 1
#define B2 2
#define B3 3

int main(int argc, char const *argv[])
{
```

```
    // Initialization
    mango::mango_init_logger();
    auto mango_rt = mango::Context("gn_saxpy", "test_manga");

    // Omitted: Initialize a, n, x, y, o

    char kernel_path[] = "/opt/mango/usr/local/share/gn_saxpy/saxpy";
    auto kf = std::make_shared<mango::KernelFunction>();
    kf->load(kernel_path, mango::UnitType::GN, mango::FileType::BINARY);

    // Register task graph
    auto k  = mango_rt.register_kernel(KERNEL, kf, {B1, B2}, {B3});
    auto b1 = mango_rt.register_buffer(B1, buffer_size, {}, {KERNEL});
    auto b2 = mango_rt.register_buffer(B2, buffer_size, {}, {KERNEL});
    auto b3 = mango_rt.register_buffer(B3, buffer_size, {KERNEL}, {});

    auto tg = mango::TaskGraph({k}, {b1, b2, b3});

    // Resource Allocation
    mango_rt.resource_allocation(tg);

    // Set up arguments
    auto argX = mango::BufferArg(b1);
    auto argY = mango::BufferArg(b2);
    auto argO = mango::BufferArg(b3);
    auto argA = mango::ScalarArg<float>(a);
    auto argN = mango::ScalarArg<int>(n);

    auto argsKERNEL =
        mango::KernelArguments({argA, argX, argY, argO, argN}, k);

    // Omitted: Write to buffers

    // Start kernel
    auto e = mango_rt.start_kernel(k, argsKERNEL);

    // Wait for kernel termination
    e->wait();

    // Omitted: Check results and free resources
}
```

Mainly due to its resource management capabilities, MANGO's programming model provides a level of abstraction over the hardware far greater than its competition, removing the querying of platform information from the list of user responsibilities. As seen in listing 2.8, the only architecture specific information required is a kernel compatible

with its target accelerator. Multiple kernel versions can be provided for a single task, leaving to the resource manager the decision of which accelerator performs its execution. In the provided example, the kernel in listing 2.7 targeting the GN architecture is used.

MANGO's API exposes a set of data structures for the specification of an user's application. A `TaskGraph` holds `Kernel`, `Buffer` and `Event` data structures, defining their dependencies. Once the task graph is defined, the resource allocation is performed. Kernel termination events are automatically generated for synchronization purposes, while users can specify custom events to use as desired. A variety of kernel arguments are supported, including scalar values, memory buffers and user events. Arguments are grouped in a `KernelArguments` structure, which is passed when a kernel is started.

MANGO V1 was the foundation for our work, in which we seek to further improve the platform through a number of additions that help close gaps left by other solutions in this field.

2.8 Gap Analysis

In this section, we will compare the programming models seen so far in order to analyze the present gaps in the heterogeneous computing space that MANGO can fill. As explained previously, MANGO is an ongoing project with the goal of enabling the development of user applications on heterogeneous HPC systems, while including a system resource manager. Our contributions to the project focused on the addition of a Python language API, Just In Time compilation of kernels, and GPU support. This work resulted in a new version of the system: MANGO V2.

For the following analysis, we consider MANGO V1 and MANGO V2 separately to further differentiate the capabilities that were already present in the first version of the system, from those introduced by our contributions.

The capabilities we are interested in for the presented programming models are:

- Device support, including CPU, GPU and FPGAs.
- Whether the model uses a single-source approach, where host and device code can coexist in the same source file, are written in the same programming language and can share common pieces of code.
- The portability of the device kernels, as in whether it is possible for a kernel

written to target a specific device, to be compiled for and executed on a different supported device.

- The ability to compile kernels at runtime (Just In Time compilation), according to their target accelerators.
- Resource management capabilities, allowing efficient use of the available devices depending on their current load, power consumption and relative performance.
- Support for control and distribution of the application along multiple clusters, each one with a set of diverse target devices.

A comparison of the capabilities of each programming model can be seen in table 2.1.

Model	*CPU*	*GPU*	*FPGA*	*Single source*	*Kernel portability*	*Kernel JIT compilation*	*Resource management*	*Multiple clusters*
MANGO V1	Yes[1]	No	Yes	No	No	No	Yes	Yes[3]
MANGO V2	Yes	Yes	Yes[2]	No	No	Yes	Yes	Yes[3]
CUDA	No	Yes	No	Yes	No	No	No	No
OpenCL	Yes	Yes	Yes	No	Yes	Yes	No	No
EngineCL	Yes	Yes	Yes	No	Yes	Yes	No	No
FluidiCL	Yes	Yes	Yes	No	Yes	Yes	No	No
ROCm	No	Yes	No	Yes	Yes	No	No	No
OpenMP	Yes	Yes	No	Yes	Yes	No	No	No
OpenACC	Yes	Yes	No	Yes	Yes	No	No	No
SYCL	Yes	Yes	Yes	Yes	Yes	No	No	No
Celerity	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Kokkos	Yes	Yes	No	Yes	Yes	No	No	No

[1] Mutually exclusive with FPGAs, only used for testing
[2] Minimal integration effort required
[3] Work in progress

Table 2.1: *Comparison of programming model capabilities*

In MANGO V2, GPUs were added to the pool of supported accelerators, and CPU devices are now handled independently (in the previous version, CPU and FPGA usage was mutually exclusive). As part of our work, we introduced a new module named Heterogeneous Hardware Abstraction Layer (HHAL), capable of handling communication with multiple target accelerators and providing an easy path for integrating new devices. We cover its implementation in section 3.5.

By looking at the table, we can see that there are two capabilities still not covered by MANGO V2: it is not a single source approach and also requires writing a separate kernel for each device.

We argue that, although at first glance introducing these capabilities would improve programmability, the performance gained by their absence is much more important to HPC.

First of all, using a single kernel programming model for multiple devices, even one that is open source, does not guarantee portability. We will take OpenCL as an example. As seen in section 2.2, it is the responsibility of the hardware vendors to provide support for any new feature of the standard. This introduces version differences between different platforms, for example Nvidia GPUs still only support OpenCL 1.2. While OpenCL 2.x introduced multiple useful features, the lack of support from Nvidia, which is one of its main targets, slowed down 2.0 adoption. The effect was so considerable that the 3.0 standard removed the requirement for most of the features introduced in 2.x.

If a developer has two target OpenCL devices for the same kernel, one with 1.2 capabilities and one with 2.0 capabilities one of two things can happen:

1. they limit themselves to 1.2 features, or
2. they write two separate kernels and possibly two separate host side implementations to exploit all the newest features on the device that supports the latest standard, while also maintaining compatibility with the 1.2 device.

As such, a great hurdle for kernel portability is the hardware vendors' unwillingness to work together [62]. This is perfectly reasonable, like in any other market, a company will gain an advantage over their competitors in whichever way they can and maintain it as long as possible. If Nvidia simply releases CUDA as an open source standard for all GPUs to utilize, they will lose their upper hand over AMD in the GPGPU market.

Even if, despite all their differences, all hardware vendors would work together on a single project, the differences in the hardware architecture will still be present. Indeed, the purpose of Heterogeneous Computing is to handle multiple devices with different memory access requirements, levels of parallelism, instruction sets and more. Although it may be possible to abstract all this away from the kernel programmer, it will not come without a cost in performance.

In order to squeeze all the performance out of the devices it is necessary to use a low-level language which takes into account the specific details of the hardware [63]. This does not mean that all kernels have to be written in assembly, but in a language that makes it possible to utilize all the capabilities offered by the target device. In consequence, it is inevitable that one or more of the available devices do not present a given capability. Thus, in today's world, achieving complete portability of a kernel

while also maintaining peak performance is impossible.

MANGO's plan is to embrace these differences in kernel implementations, allowing the developers to write their device code in the best suited programming language or framework available. Our programming model abstracts away all the *host side* code to be able to handle multiple different platforms, allowing for different types of implementations to coexist and communicate with each other, no matter what the chosen model for the device kernel is.

A rarely supported functionality is Just In Time compilation of kernels. With OpenCL and its users being the only ones to provide run-time compilation of kernels, in MANGO V2 we introduced this functionality as part of the HHAL module, which we named the Dynamic Compiler. The Dynamic Compiler is capable of compiling kernels at run-time for each of the supported devices, this way kernels are compiled from source as needed, according to their assigned accelerator. The main advantage this functionality gives is smoothing the development process, by allowing developers to work directly with kernel source code without requiring manual compilation for each target architecture.

The key advantage of MANGO over the other mentioned programming models is its integration with a resource manager, namely BarbequeRTRM. While the previous models focus on executing a single kernel or set of kernels as fast as possible on a given platform, they do not take into account the current state of the machine that they are running on. If multiple applications could be running at the same time, there is a chance that the device most suitable for a given application is currently busy serving something else. Depending on what other devices are available, their relative performance for the kernel, power consumption and other aspects, it may be more efficient to execute the work on a different device or just wait for the preferred one to be available. In addition, it could also be advantageous to suspend the execution of a given application in order to give priority to another which is more time sensitive.

Lastly, no other model except Celerity offers distributed execution of an application over multiple clusters. Since its inception, one of the goals of MANGO was to provide this capability, allowing it to exploit the heterogeneity of a large number of different machines. In conjunction with BarbequeRTRM, which also provides the capability to manage resources in a distributed system, it would be possible to launch a number of different applications on a multitude of computers and achieve an efficient and performant execution of all tasks. There are still advancements to be done in this front, as it was not the topic of our work, but many implementation decisions were driven by it.

CHAPTER 3

Architecture Description

In this Chapter, we provide a detailed description of the resulting MANGO software architecture, after the completion of our work. First, we provide an overall description of the system, followed by an explanation of the core elements. Then, we dive deeper into the inner workings of each MANGO module and their interaction with the rest of the system.

3.1 Overall description

The MANGO project aims at allowing developers to easily create applications that target different types of accelerator architectures. In particular, three types of accelerators are considered: symmetric multiprocessors, which are characterized by good capabilities in terms of OS support and execution flexibility (i.e. they are able to run a POSIX-compliant runtime); GPGPU-like accelerators, which are programmable but are not able to run a fully compliant POSIX runtime; and hardware accelerators, which do not need nor support any kind of software runtime. Applications may be developed either by domain experts with limited knowledge of parallel computing and program-

ming models, or by more experienced programmers.

The user-facing module (Libmango), therefore, needs to operate in a way that is akin to an intermediate language in a compiler: it must allow the software stack developers to easily map high-level programming models on the range of supported accelerators, while providing functional compatibility [1].

Depending on the individual capabilities of each accelerator, the accelerator-facing section of the system should also introduce optimizations and/or additional features.

In addition, developers must be capable of easily expanding the MANGO system by adding support of other architectures. The HHAL module, along with platform libraries (NAN, HN and GN), take care of these requirements.

Even though the current implementation runs entirely on a single machine, the internal communication of the system is designed to enable the expansion of the architecture into multiple distributed nodes.

User applications, however complex, must be scheduled to run on the system available resources. This task, also involving the resource allocation, can be performed with multiple optimization goals in mind, job that is carried out by the Resource Manager: BarbequeRTRM.

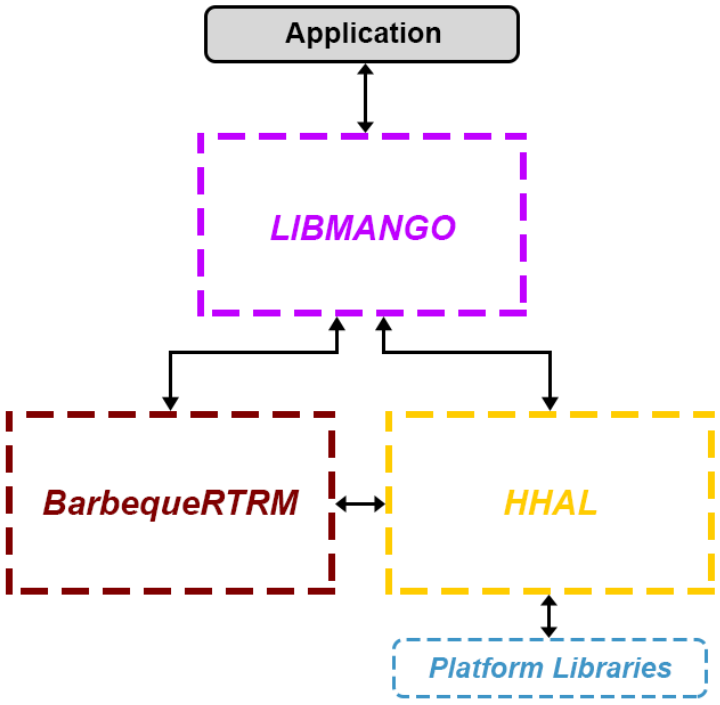


Figure 3.1: *MANGO system - architecture overview*

In figure 3.1, a complete overview of the MANGO system is presented, featuring the Libmango, BarbequeRTRM and HHAL modules, as well as platform libraries.

3.2 Core elements

Throughout the multiple MANGO modules, there are a few core elements often present in each module. These are the components necessary to specify and control an user application and its execution. Although their names may vary from module to module, here they are defined as Kernel, Memory Buffer, Event and Task graph.

3.2.1 Kernel

In computing, a compute kernel is a routine compiled for high throughput accelerators (such as graphics processing units (GPUs), digital signal processors (DSPs) or field-programmable gate arrays (FPGAs)), separate from but used by a main program (typically running on a central processing unit) [64].

The MANGO system manages user defined Kernels and their execution. For a particular Kernel, multiple sources (accelerator specific implementations of the kernel) can be specified. The architectures for which an implementation is available are considered in the kernel-accelerator assignment process by the resource manager.

As any computer program, a kernel must have the capability of interacting with the outside world in order to perform significant work. This is achieved through the support of kernel arguments.

Three types of kernel arguments are supported:

- Scalar Argument: A scalar value. For example, an integer.
- Buffer Argument: A pointer to a Memory Buffer.
- Event Argument: A pointer to an Event data type.

3.2.2 Memory Buffer

A Memory Buffer is a region of memory used to store data and through which communication to, from and among kernels is carried out.

As depicted in figure 3.2, Kernels read from Input Buffers and write to Output Buffers for inter-kernel and host-kernel data transferring.

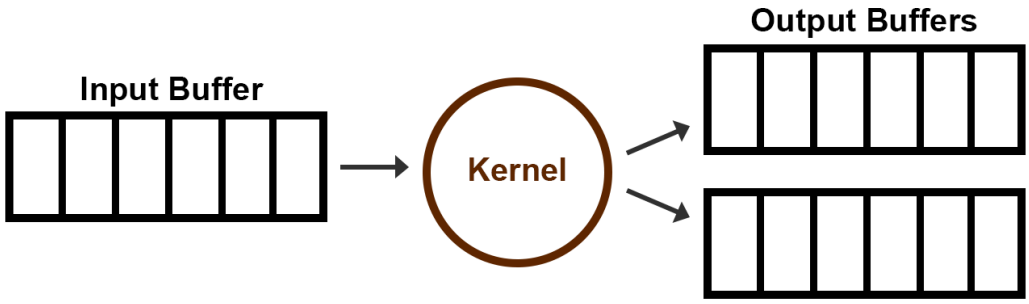


Figure 3.2: *Input and Output Buffers*

A Memory Buffer is defined by the user and allocated by MANGO at the target architectures, where a Kernel that makes use of said Buffer (as either input or output) is assigned.

3.2.3 Event

An event is a data structure utilized for communication and synchronization of different parts of the system.

User defined events can be accessed by Kernels through Event Arguments, providing the user with the necessary tools for the implementation of host-kernel or inter-kernel synchronization.

By default, MANGO utilizes kernel termination events for both internal and host synchronization.

3.2.4 Task Graph

The Task Graph gives a global picture of the application's behavior and represents data and control dependencies between Kernels, Memory Buffers and Events. Through the Task Graph, the resource manager obtains the necessary information to formulate the best possible resource allocation, also taking into account the requested performance and Quality of Service (QoS) levels.

The example in figure 3.3 demonstrates the Task Graph of an application consisting of four Kernels, their respective termination Events, and data dependencies among Kernels in the form of Buffers.

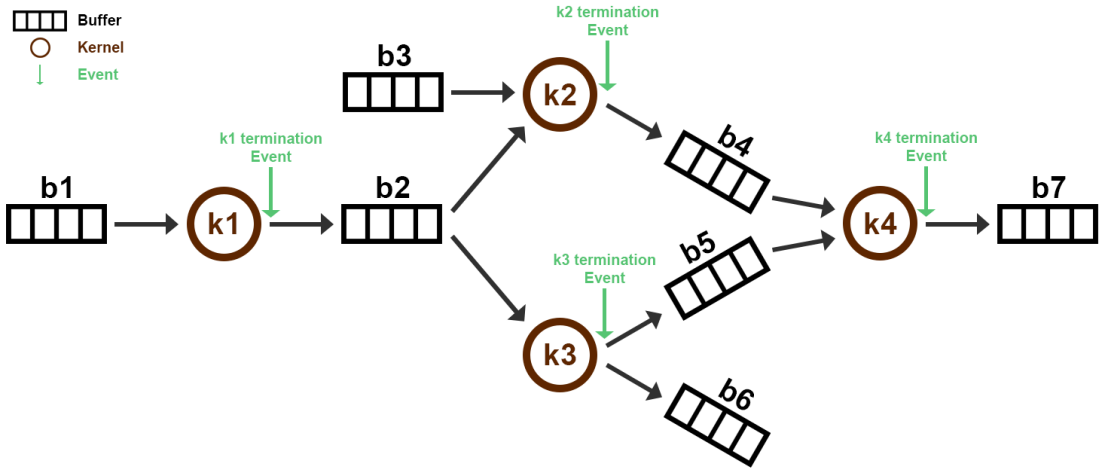


Figure 3.3: *Task Graph example*

3.3 Libmango

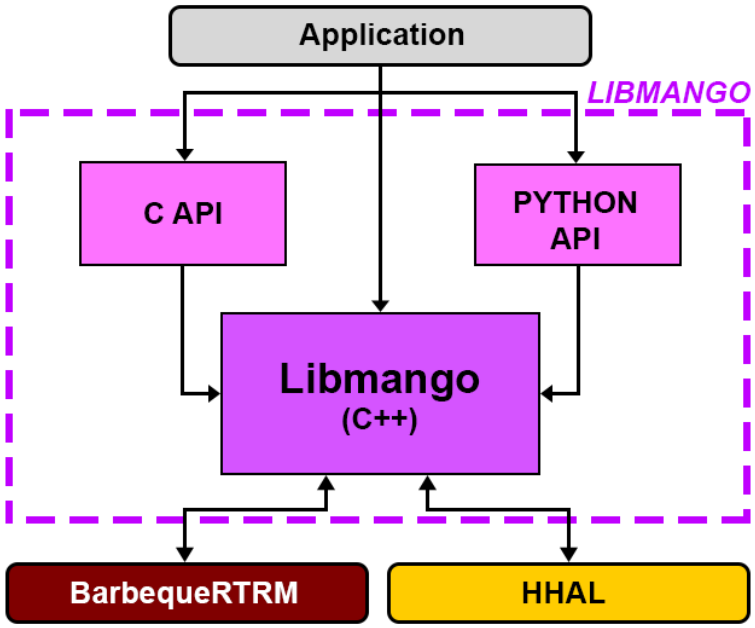


Figure 3.4: *Libmango module overview*

Libmango is the front-facing module of the MANGO system, hence providing the system's API for user interaction with the underlying components, and acting as an abstraction layer between user defined models and module specific requirements.

The goal of the Libmango module is to allow software stack developers to easily map high-level programming models on the range of supported accelerators. Through

the communication with the BarbequeRTRM and HHAL modules, user models are automatically mapped to the supported accelerators in a transparent manner, removing integration complexity from the user's hands.

The Libmango API allows developers to indicate to the runtime the components of their application, namely kernels, memory buffers and events. These are grouped in a task graph that represents the dependencies among the multiple components.

Libmango is implemented in the C++ programming language, so its native usage is through the C++ API. However, a set of wrappers for other languages are provided, namely the C and Python API wrappers.

3.3.1 Context

The Context is the main class in the Libmango module, it holds the state information of the host side runtime for a single application, and it is created by the user at the beginning of their interaction, with an application name and a recipe file needed by BarbequeRTRM for the resource allocation (further explained in the BarbequeRTRM section 3.4.3). Every subsequent component (kernels, buffers, events and task graph) has to be registered in the Context in order to be considered part of the application.

Once the application is specified, the task graph information is sent to the BarbequeRTRM for the resource allocation. After a successful resource allocation, the application is ready to run.

3.3.2 Kernel management

Libmango exposes functionalities and data structures that can be used to represent and manipulate kernels.

Kernels are identified by an user-provided integer. For a single kernel, multiple implementations can be specified, each one targeting a different supported accelerator and thus provided in their respective architecture's requirements, i.e. a kernel targeting an Nvidia GPU would require a CUDA implementation.

Kernel versions (implementations) are stored either in memory or in external files. According to the targeted architecture, multiple source types are supported. The kernel source can be a pre-compiled binary file or code in accelerator-supported language, provided via a memory-stored string or a source file, to be dynamically compiled as required. The resource manager relies on the available options for the assignment of kernels to accelerators.

Kernels can be manually started by the developer once the resource allocation is successfully completed.

Libmango supports three types of Kernel arguments: Scalar arguments, Buffer arguments and Event arguments. These act as wrappers of the HHAL kernel arguments.

Scalar argument

A Scalar argument consists of a scalar value. The types supported by Libmango are signed and unsigned integers of sizes 8, 16, 32 and 64 bits, as well as floating point values. When a Scalar argument is created, the provided value is copied and stored in memory, and later sent to the HHAL module when their respective kernel is ran.

Buffer argument

A Buffer argument consists of a Buffer integer identifier. The corresponding memory pointer in the accelerator's memory space is passed as an argument to the Kernel at execution time.

Event argument

An Event argument consists of an Event integer identifier. The corresponding Event is passed as an argument to the Kernel at execution time. Event implementation is architecture dependent, hence the data structure received by the kernel may vary for each architecture.

3.3.3 Buffer management

Libmango exposes functionalities and data structures that can be used to represent and manipulate Memory Buffers. Buffers are the main data communication instruments between the host and the executing Kernels.

A Buffer is identified by an user-provided integer. It consists of a pointer to a memory location where the Memory Buffer starts, and its size in bytes. For creating a Buffer, the user needs to register it to the application's Context. A Buffer is automatically allocated in the same accelerator where the Kernel that writes to, or reads from it, is assigned to by the resource manager. Once successfully allocated, Libmango permits the writing of the Buffer with host-side data, as well as reading from the Buffer into host memory.

3.3.4 Event management

To provide developers with synchronization capabilities, Libmango exposes Event functionalities.

An Event is identified by an integer generated by Libmango. Events are synchronization data structures with an internal value utilized to indicate the different stages in the process, or in the case of user-defined Events, any denotation the user gives it.

Libmango lets the user define their own Events. These can be passed as arguments to the Kernels (if the target architecture supports them), which allows for user-defined inter-Kernel or host-Kernel synchronization.

By default, Libmango utilizes Events for Kernel termination synchronization. For every registered Kernel, Libmango automatically generates a Kernel-termination Event, which can also be accessed by the developers for waiting until a started Kernel finishes.

3.3.5 API Wrappers

As the core implementation of Libmango is in the C++ programming language, a set of wrappers are provided to complement the C++ API and allow developers to make use of the MANGO system using their programming language of choice.

Besides the native C++ API, Libmango currently exposes C and Python API wrappers.

C API Wrapper

The C language API is a wrapper around the C++ API that is provided both for usage in C applications, and for compatibility with the early version of MANGO, which was developed in C.

All the data types in the function prototypes have been made opaque using specific typedef types. This hides to the application some specific types of the machine, such as the size of the memory addresses. In the current MANGO implementation, the used types are mostly uint32_t, due to the addressing size that is of 32-bit.

The API is divided in 8 groups: initialization and shutdown, kernel loading, task graph definition, task graph registration, resource allocation, kernel launching, synchronization primitives, and data transfer [1].

Python API Wrapper

The Python wrapper aims at expanding MANGO accessibility onto scripting programming languages. Due to the nature of the MANGO system, scripting languages are a perfect fit, as the computation-heavy part of the user application are often the kernels executed in the available accelerators.

Python's popularity as a programming/scripting language made it an ideal candidate to enable a great number of developers access to the MANGO platform. As observed in figure 3.5, Python was the second most popular programming language in the RedMonk's Programming Language Rankings [65], only falling behind JavaScript.

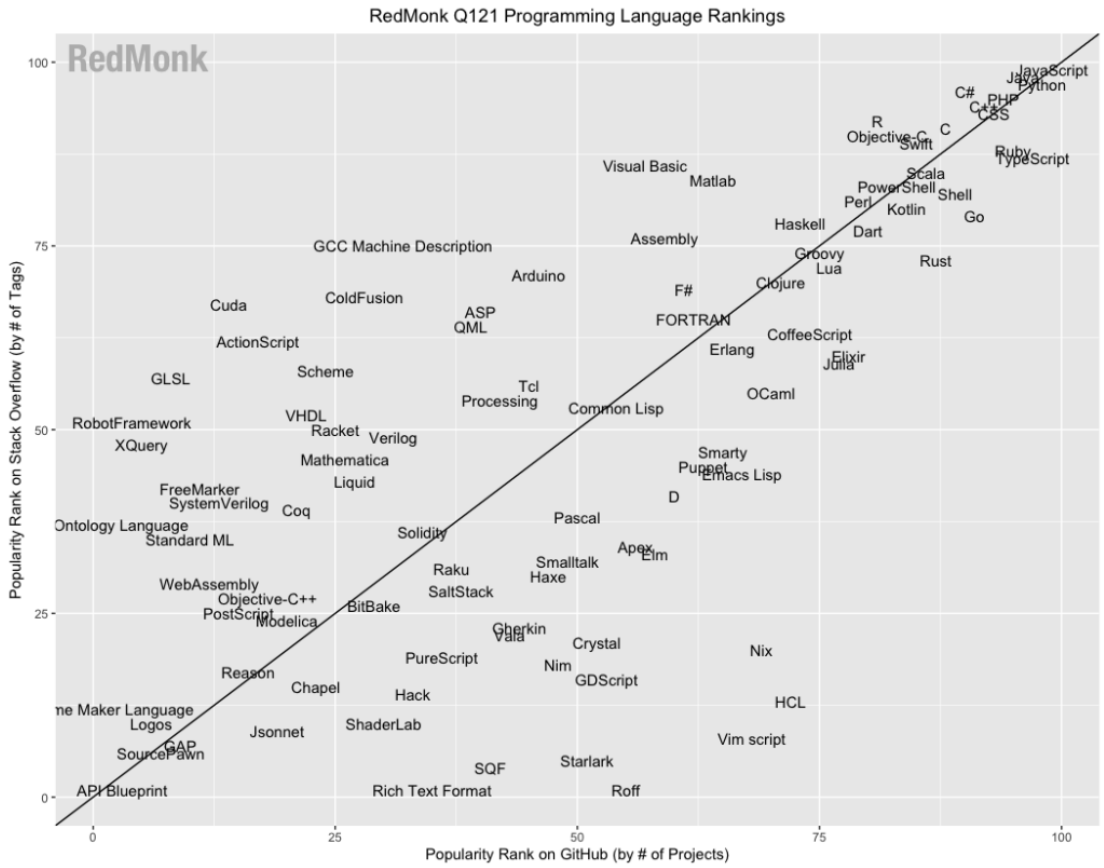


Figure 3.5: *RedMonk's January 2021 Programming Language Rankings*

The Python API wrapper is implemented using Cython, with support for Python 3. Cython is an optimizing static compiler that allows writing Python code that calls back and forth from and to C or C++ code natively at any point [66]. For the API user, Cython usage is completely transparent and not a requirement, so their code can be purely written in Python.

Listing 3.1: *Python API Example snippet*

```
# Omitted: Register kernel(k), buffers(b1,b2,b3) and build task graph(tg)
with ctx.resource_allocation(tg):
    arg1 = BufferArg(b1)
    arg2 = BufferArg(b2)
    arg3 = BufferArg(b3)
    arg4 = ScalarArg(size, ScalarType.INT)

    args = KernelArguments(k, arg1, arg2, arg3, arg4)

    b1.write(A.tobytes())
    b2.write(B.tobytes())

    end_event = ctx.start_kernel(k, args)

    end_event.wait()
    b3.read(C)
# Omitted: Check results
```

3.3.6 Sample Application

In this section we will go over a sample to showcase and explain the Libmango C++ API usage. The sample application consists on the computation of a SAXPY operation (z = ax + y) over two trivially pre-initialized arrays: x and y.

Listing 3.2: *saxpy.cu*

```
extern "C" __global__
void saxpy(float a, float *x, float *y, float *out, int n) {
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < n) {
    out[tid] = a * x[tid] + y[tid];
  }
}
```

The target architecture of the sample is CUDA, hence a precompiled binary of the shown CUDA kernel that performs the SAXPY operation is used.

Listing 3.3: *Sample - Includes*

```
#include <cstddef>
#include <iostream>
#include <memory>

#include <host/context.h>
#include <host/logger.h>
```

First we set up the sample with the necessary includes. Regarding Libmango, `context.h` is the required header that exposes the C++ API. `logger.h` is also included to access the Libmango logger.

Listing 3.4: *Sample - Definitions*

```
#define KERNEL 1
#define B1 1
#define B2 2
#define B3 3

// saxpy function matching the CUDA kernel, used to check the results
void saxpy(float a, float *x, float *y, float *o, float n) {
    for (size_t i = 0; i < n; ++i) {
        o[i] = a * x[i] + y[i];
    }
}
```

We now define the kernel and buffers integer identifiers, as well as a `saxpy` function that matches the **saxpy.cu** kernel seen in listing 3.2, which we will use to check the obtained results.

Listing 3.5: *Sample - Initialization*

```
int main(int argc, char const *argv[])
{
    // Initialization
    mango::mango_init_logger();
    auto mango_rt = mango::Context("cuda_simple", "test_manga_cuda");

    int n = 4096;
    size_t buffer_size = n * sizeof(float);
    float a = 2.5f;
    float *x = new float[n], *y = new float[n], *o = new float[n];

    for (size_t i = 0; i < n; ++i) {
        x[i] = static_cast<float>(i);
        y[i] = static_cast<float>(i * 2);
    }
```

At the beginning of the `main` function, we initialize the logger, and the application's `Context` is created. For the initialization of the `Context`, an application name is required: "cuda_simple", as well as a recipe file name for the BarbequeRTRM resource allocation: "test_manga_cuda".

Then, the three buffers we need for the operation are declared. `x` and `y` are the input buffers, so they are initialized with known values. `o` is the output buffer where we will

store the results, so there is no need to initialize its data.

Listing 3.6: *Sample - Kernel loading*

```
char kernel_path[] = "/opt/mango/usr/local/share/cuda_simple/saxpy";
auto kf = std::make_shared<mango::KernelFunction>();
kf->load(kernel_path, mango::UnitType::Nvidia, mango::FileType::BINARY);
```

To load the saxpy kernel binary file, we create a `KernelFunction` object, and then load the kernel through the `load()` function, specifying the target architecture and file type.

Listing 3.7: *Sample - TaskGraph registration and resource allocation*

```
// Registration of task graph
auto k  = mango_rt.register_kernel(KERNEL, kf, {B1, B2}, {B3});

auto b1 = mango_rt.register_buffer(B1, buffer_size, {}, {KERNEL});
auto b2 = mango_rt.register_buffer(B2, buffer_size, {}, {KERNEL});
auto b3 = mango_rt.register_buffer(B3, buffer_size, {KERNEL}, {});

auto tg = mango::TaskGraph({k}, {b1, b2, b3});

// Resource Allocation
mango_rt.resource_allocation(tg);
```

In order to realize the resource allocation, we first need to register the elements in the `Context` and create the `TaskGraph`. The kernel is registered in the Libmango `Context` by providing its id, kernel function and input and output buffers. The buffers are registered by providing their ids, size, kernels where they act as input and kernels where they act as output.

Finally, the `TaskGraph` is created with the previously registered elements and the resource allocation is performed over the specified `TaskGraph`.

Listing 3.8: *Sample - Arguments set up*

```
auto argX = mango::BufferArg(b1);
auto argY = mango::BufferArg(b2);
auto argO = mango::BufferArg(b3);
auto argA = mango::ScalarArg<float>(a);
auto argN = mango::ScalarArg<int>(n);

auto argsKERNEL = mango::KernelArguments({argA, argX, argY, argO, argN},
     k);
```

The `saxpy` kernel receives five arguments, two scalars and three buffers. A `BufferArg` is created for each buffer argument, and two `ScalarArg` are created with their respec-

tive types (float and int) for each of the scalar arguments.

A `KernelArguments` object groups the arguments to be passed to a given kernel in the stated order.

Listing 3.9: *Sample - Writing buffers*

```
std::cout << "Sample host: Writing to buffer 1..." << std::endl;
b1->write(x, buffer_size);

std::cout << "Sample host: Writing to buffer 2..." << std::endl;
b2->write(y, buffer_size);
```

Before launching the kernel, we need to write the data from the host buffers onto the registered buffers that where allocated at the target accelerators in the resource allocation. To write to a buffer, we use the `write()` function of the `Buffer` objects that were returned from the `Context` registration.

Listing 3.10: *Sample - Kernel launch*

```
std::cout << "Sample host: Starting kernel..." << std::endl;
auto e = mango_rt.start_kernel(k, argsKERNEL);

std::cout << "Sample host: Waiting for kernel completion..." << std::
    endl;
e->wait();
```

We are now ready to execute the kernel. The `start_kernel()` function takes the kernel and its arguments and returns a kernel termination `Event`. By calling the event's `wait()` function, the host execution is blocked until the kernel's termination is notified.

Listing 3.11: *Sample - Checking results*

```
b3->read(o, buffer_size);

float *expected = new float[n];
saxpy(a, x, y, expected, n);

bool correct = true;
for (int i = 0; i < n; ++i) {
    if (o[i] != expected[i]) {
        std::cout << "Sample host: Error!\n" << std::endl;
        correct = false;
        break;
    }
}
if(correct) {
    std::cout << "Sample host: SAXPY correctly performed!" << std::endl;
```

```
    }
```

Once the kernel terminated, we can read the results from the output buffer `b3`. By using the buffer's `read()` function, we can read the data from the accelerator's memory into the `o` host buffer.

We use the `saxpy` function defined at the beginning of the sample to check the correctness of the results.

Listing 3.12: *Sample - Teardown*

```
    mango_rt.resource_deallocation(tg);
    delete[] x;
    delete[] y;
    delete[] o;
    delete[] expected;

    return 0;
}
```

Before returning, we perform the resource deallocation of the `TaskGraph`, and free the host buffers.

3.4 BarbequeRTRM

The MANGO architecture is based on the idea of building energy efficient HPC (High Performance Computing) systems. In heterogeneous architectures, the resource management problem is especially complex due to the difficulty of scheduling tasks over multiple architectures with different instruction sets and requirements.

A resource manager performs the tasks' scheduling and assignment of resources to the available accelerators, according to the application's objective, like the minimization of power consumption or the minimization of execution time [1].

The framework used for resource management in the MANGO system is the Barbeque Run-Time Resource Manager (BarbequeRTRM). As described in its official documentation [67], BarbequeRTRM is a modular and extensible run-time resource manager, to manage the allocation of computing resources (i.e. CPU, GPU, memory, HW accelerators, etc.) to multiple concurrent applications. Its modular design allows the developers to add custom resource allocation policies, according to specific use-cases and target platforms [68, 69]. As BarbequeRTRM is a large system with multiple applications, in this section we will focus on it's usage in the MANGO system and the relevant components involved in the process.

3.4.1 Resource Manager Architecture

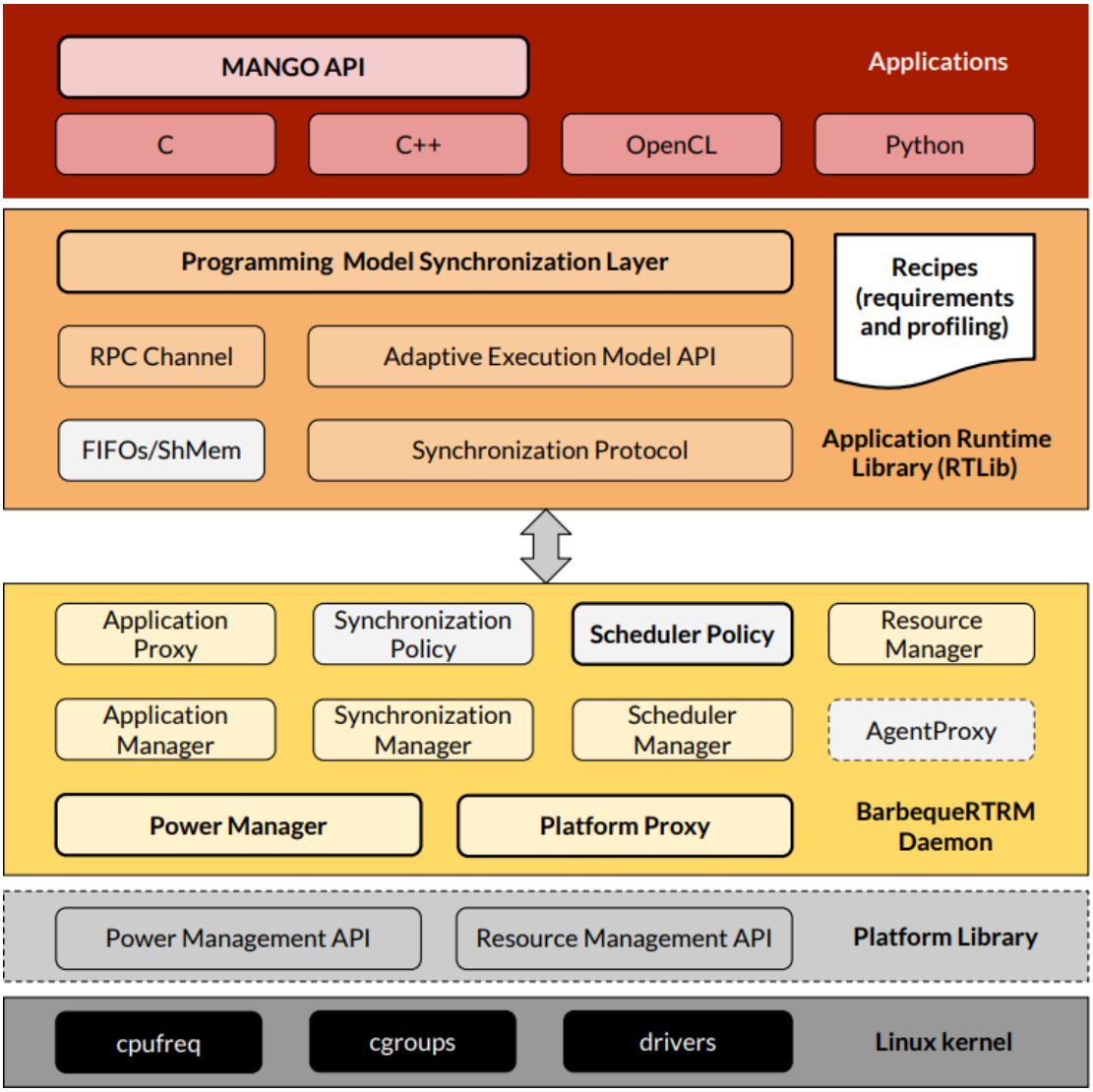


Figure 3.6: *Overview of the BarbequeRTRM [1]*

In figure 3.6, we can see an overview of the BarbequeRTRM framework. The interfacing with the rest of the MANGO system happens at application level, specifically Libmango natively communicates with the Application Runtime Library (RTLib) implemented in C++, which in turns acts as a client of the BarbequeRTRM Daemon running in the host system.

The Programming Model Synchronization Layer in the RTLib facilitates MANGO integration with the BarbequeRTRM by working as an abstraction layer that supports the MANGO programming model for a BarbequeRTRM managed application. The Libmango-BarbequeRTRM interaction happens through an **Application Controller**,

which is created when a Libmango **Context** is initialized.

The daemon and applications communicate through a Remote Procedure Call (RPC) based protocol. Data is mainly exchanged by using named pipes; a general one for the messages coming from the application to the resource manager, plus one pipe per application for application management purposes. A further communication channel, based on shared memory, has been introduced in MANGO to enable an efficient transfer of complex data structures, like for example the task-graph representations of the applications.

Regarding hardware support, suitable system interfaces for accessing low-level mechanisms are required. The two components that create an abstraction layer on top of the platform and resource specific interfaces are the **Platform Proxy** and the **Power Manager**. Examples of such interfaces are the Linux frameworks cgroup and cpufreq, used to bound the amount of CPU time, memory and number of CPU cores assigned to an application, as well as managing the clock frequency of cores. There are, of course, resources out of the Linux operating system control, in these cases platform-specific libraries are used, for example the Nvidia Management Library (NVML) [70] is utilized for controlling and obtaining runtime data of Nvidia GPUs [1].

The **Platform Proxy** also performs the resource assignment for its platform. This is done through functions exposed by the **HHAL API**, for which a specific Platform Proxy must have an HHAL Client.

In terms of task scheduling, the **Scheduler Manager** is responsible for loading a **Scheduling Policy** and executing the scheduling algorithm for a given task graph. Multiple resource allocation policies can be implemented to account for the specific requirements of the user's system and application, i.e. policies focusing on power and temperature control, execution time, isolation, etc. [1]. The utilized resource allocation policy is selected at installation time. Here its worth mentioning the **recipe** file (further explained in section 3.4.3), through which the user can specify the QoS required for their application.

3.4.2 Integration

As previously stated, The Libmango-BarbequeRTRM communication is done through an Application Controller belonging to the Programming Model Synchronization Layer, which is created when a Libmango Context is initialized.

When initializing an Application Controller, an application name and a recipe file are provided. The first one for identification purposes and the second one for applica-

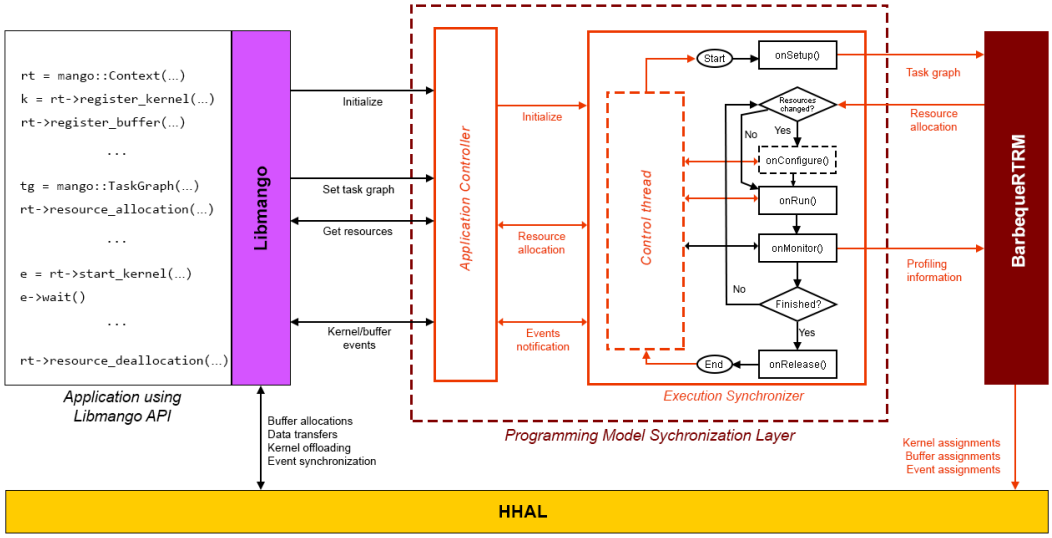


Figure 3.7: *Flow between BarbequeRTRM and the MANGO system (Based on: [1])*

tion requirements specification (more information about recipes in section 3.4.3).

As shown in figure 3.7, the next step after initialization is the resource allocation, for this, a task graph representing dependencies between the application components needs to be created and passed in the `resource_allocation()` function call. The task graph is similar to the Libmango one covered in section 3.2.4, requiring a simple translation before being sent for resource allocation.

Once the resource allocation is completed, the **Execution Synchronizer** locks on the `onConfigure()` function, waiting for kernels to start. During this time, applications can proceed by loading kernels into their assigned devices and writing into input buffers. Once a kernel is run, the `onConfigure()` function unlocks, spawning a monitor thread for each running kernel in order to keep track of execution time and throughput of each of them. For BarbequeRTRM to be up to date with the application's execution status, event notifications for events such as kernel start and stop, and buffer writes and reads are notified by Libmango.

Finally, when all kernels terminate, the resource deallocation is performed and the Application Controller is deleted, releasing all the assigned resources in the `onRelease()` function.

3.4.3 Recipe file

The **Recipe** file has already been mentioned a few times in the document. A recipe file path is passed to the Application Controller at initialization, and it is used to specify the application requirements to be taken into account in the resource allocation procedure.

The recipe is an XML [71] format file named with a `.recipe` extension, through which developers can request the QoS or performance level that their application can achieve when a certain set of resources is assigned by the resource manager. These requirements are internally referred to as Application Working Modes (AWM) [1]. However, to further fit MANGO requirements, a resources section was introduced.

As observed in listing 3.13, the `application` tag is the root tag from which the application information is defined, where a `priority` value can be specified, with `0` as the highest priority.

Under the `platform` section, the `id` value identifies the architecture targeted in the enclosed specification. Specifications for multiple targets can be provided, leaving to BarbequeRTRM the decision of choosing the target to be utilized.

Referring back to AWMs, in the `awm` section, the following attributes are available:

- `id`: A number identifier.
- `name`: A name used for logging purposes.
- `value`: A performance level.
- `config-time`: The configuration time profiled when the application switches to the given AWM.

The `resources` subsections contain the resource assignment configurations. Under `cpu`, the CPU time quota `pe` is expressed in percentage, the amount of memory is defined in the `mem` tag, the number of accelerator cores as `pe` under `acc` and the network bandwidth as `net`.

Since due to the MANGO programming model, per-task and per-buffer resource mapping is required, the AWM specification is not enough to cover the MANGO needs. The `tg` section was introduced to allow specification of task graph related mapping information. As seen in listing 3.13, under the `reqs` tag, `task` information is provided for each task (kernel) in the application. The attributes are:

- `id`: A number identifier.

- `name`: A name used for logging purposes.
- `hw_prefs`: Ordered list of accelerator preferences.
- `inbw_kbps`: Read bandwidth.
- `outbw_kbps`: Write bandwidth.
- `grid_dim`: Grid dimensions for an Nvidia kernel.
- `block_dim`: Block dimensions for an Nvidia kernel.

Listing 3.13: *BarbequeRTRM Recipe file - Saxpy Sample in GN and Nvidia*

```
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
  <application priority="4">
    <platform id="bq.*">
      <awms>
        <awm id="0" name="OK" value="100">
          <resources>
            <cpu>
              <pe qty="100"/>
            </cpu>
            <mem qty="20" units="M"/>
          </resources>
        </awm>
      </awms>
      <tg>
        <reqs>
          <task name="t1" id="1" hw_prefs="nvidia" grid_dim="32,1,1"
              block_dim="256,1,1"/>
          <task name="t2" id="2" hw_prefs="gn"/>
        </reqs>
      </tg>
    </platform>
  </application>
</BarbequeRTRM>
```

3.4.4 Resource Scheduling and Platform Support

As a MANGO developer, there are two main aspects of the BarbequeRTRM to keep in mind when adding support of new architectures and extending MANGO capabilities. These are the scheduling policies and the integration with new platforms.

To implement a new scheduling policy, a version of the `SchedulerPolicyIF` class must be introduced. The scheduler is in charge of the main functionality of the

resource manager: allocating resources according to the application's characteristics, the current status of the system and the user requirements. The central method of the scheduler is the `Schedule()` function, which receives references to the available resource interfaces, as well as information on the applications. Regarding QoS and task graph requirements, the user specifications loaded from the recipe file (task performed by the `RecipeLoader`) can be accessed by the scheduler to take into account in the scheduling process.

To fulfill its role, the scheduler inherently needs platform-specific information upon which resource allocation decisions can be made. The `PlatformProxy` is the class that acts as an abstraction layer between a platform library and the BarbequeRTRM, providing a set of functions for the scheduler to query the available accelerators. For each supported architecture, a different `PlatformProxy` must be introduced. Optionally, a `PowerManager` that manages power related accelerator information can also be implemented for a given architecture.

However, there is another primary job the `PlatformProxy` carries out. After the scheduling procedure is completed, the `MapResources()` function of each `PlatformProxy` is called. Here the assignments of kernels, buffers and events are done. Since we are particularly dealing with MANGO supported architectures, the assignments are performed through the HHAL module, covered in the following section.

3.5 HHAL

The Heterogeneous Hardware Abstraction Layer (HHAL) is the module of the MANGO system that takes care of the communication with the multiple accelerator libraries. As such, HHAL abstracts accelerator specific information in a manner that allows the resource manager to fully exploit architecture specific features, while freeing Libmango from the inherent complexity of handling multiple architectures.

Due to the fact that multiple modules, running as independent processes or potentially on distributed nodes, need to make use of the Heterogeneous Hardware Abstraction Layer, HHAL works in a client-server manner. The server is run as a daemon, and the client is used by other modules to interact with it through sockets. In this way, both BarbequeRTRM and Libmango can make use of a common instance of the HHAL module.

HHAL exposes an architecture-agnostic API with the necessary functionalities to permit the managing of kernels, buffers and events on any of the supported architectures. The module is structured as a front-facing API and a set of architecture specific

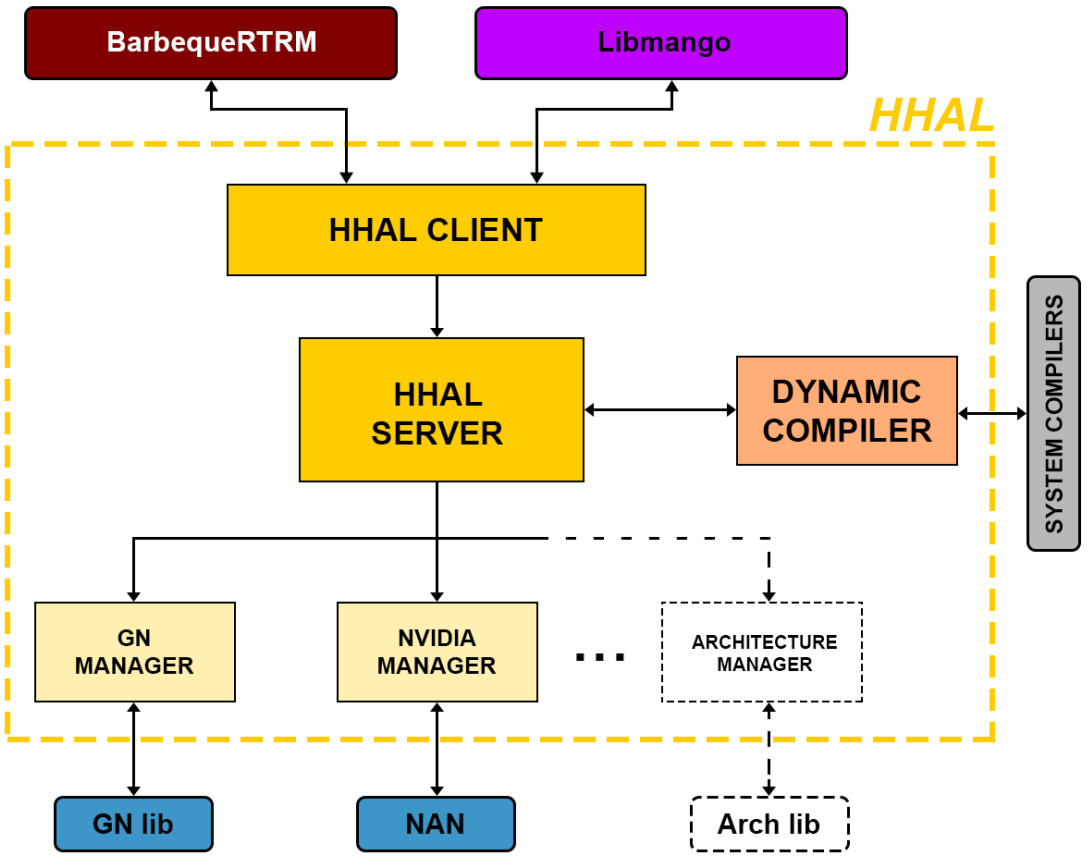


Figure 3.8: *HHAL module overview*

managers that implement the required functionalities for their specific architecture.

3.5.1 Abstracting architecture-specific information

Due to the fact that a single API is utilized for interacting with every supported architecture, underlying architecture managers require a mechanism that allows the description of architecture specific information by external modules, specifically the resource manager.

For each resource, there is a base structure that contains the minimal information required by the front-facing API, namely the resource's identification integer.

Listing 3.14: *HHAL API - Base structures*

```
typedef struct hhal_kernel_t {
    int id;
} hhal_kernel;

typedef struct hhal_buffer_t {
    int id;
```

```
} hhal_buffer;

typedef struct hhal_event_t {
    int id;
} hhal_event;
```

Architecture managers can expand these base structures to add architecture specific information that they may require. For example, the following are the structures used by the Nvidia Manager.

Listing 3.15: *HHAL Nvidia Manager - Extended structures*

```
typedef struct nvidia_kernel_t {
    int id;
    int gpu_id;
    int mem_id;
    uint32_t grid_dim_x;
    uint32_t grid_dim_y;
    uint32_t grid_dim_z;
    uint32_t block_dim_x;
    uint32_t block_dim_y;
    uint32_t block_dim_z;
    int termination_event;
} nvidia_kernel;

typedef struct nvidia_buffer_t {
    int id;
    int gpu_id;
    int mem_id;
    size_t size;
    std::vector<int> kernels_in;
    std::vector<int> kernels_out;
} nvidia_buffer;

typedef struct nvidia_event_t {
    int id;
} nvidia_event;
```

The general API essentially acts as a dispatcher, and passes the casted structure pointer when calling the corresponding manager function. A diagram of the dispatcher flow is shown in figure 3.9.

The following code snippet is for example purposes and not part of the final implementation.

Listing 3.16: *HHAL API Example - Dispatching architecture-specific structures*

```
void HHAL::example_function(Unit unit, hhal_kernel *info) {
```

```
    switch (unit) {
        case Unit::GN:
            GN_MANAGER.example_function((gn_kernel *) info);
            break;
        case Unit::NVIDIA:
            NVIDIA_MANAGER.example_function((nvidia_kernel *) info);
            break;
        case Unit::TEST:
            TEST_MANAGER.example_function((test_kernel *) info);
            break;
    }
}
```

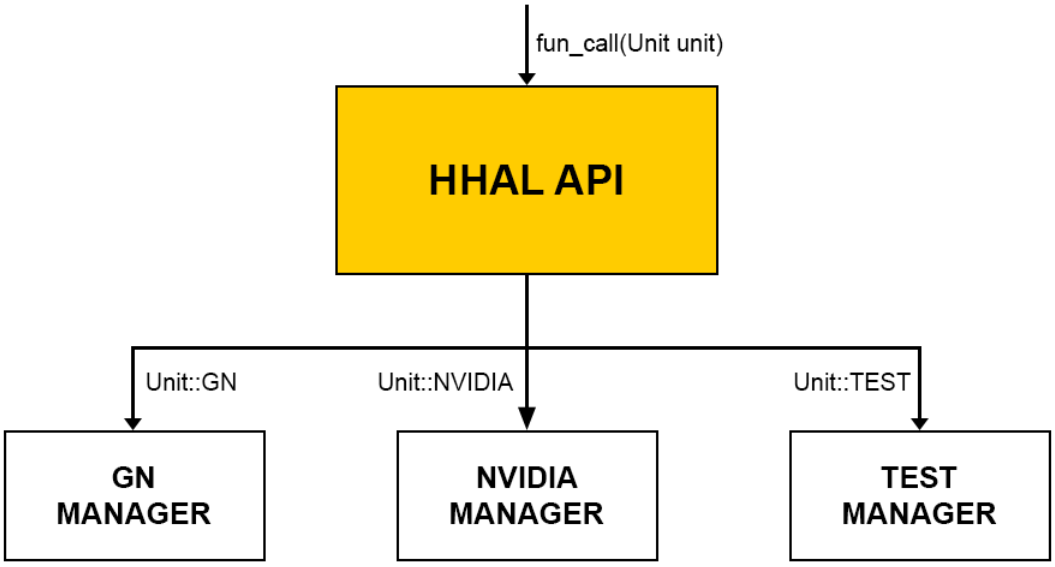


Figure 3.9: *HHAL API dispatching flow (test manager added for example purposes)*

3.5.2 HHAL API

The API exposed by HHAL provides a set of functionalities for managing resources. Here we explain in detail each functionality and how they relate to each component of the user's application: kernels, buffers and events.

Resource Assignment

Each resource composing the overall user application is assigned to an architecture by the resource manager. The resource assignment takes place before the allocation in its assigned target accelerator, and its the step in which the resource's information is made

known to HHAL.

In the assignment process, an identification integer is provided for the resource being assigned. This same id is utilized in successive operations to map a resource with the information provided at assignment time.

To assign a resource, a supported unit where the resource is to be assigned is provided in the function call. Further usage of the resource will automatically be handled by the corresponding architecture manager.

Listing 3.17: *HHAL API - Assign functions*

```
HHALExitCode assign_kernel(Unit unit, hhal_kernel *info);
HHALExitCode assign_buffer(Unit unit, hhal_buffer *info);
HHALExitCode assign_event (Unit unit, hhal_event *info);

HHALExitCode deassign_kernel(int kernel_id);
HHALExitCode deassign_buffer(int buffer_id);
HHALExitCode deassign_event(int event_id);
```

Once a resource is no longer required by the application, it may be deassigned and thus removed from the HHAL module runtime.

Resource Allocation

Every assigned resource has to eventually be allocated in its target accelerator in order for their associated kernel/s to be able to run.

Both allocation and release (deallocation) function calls are simple, due to the fact that the required information regarding the resources is provided beforehand at assignment time. Thus, only the resource's identification integer is needed.

Listing 3.18: *HHAL API - Allocation functions*

```
HHALExitCode allocate_kernel(int kernel_id);
HHALExitCode allocate_memory(int buffer_id);
HHALExitCode allocate_event(int event_id);

HHALExitCode release_kernel(int kernel_id);
HHALExitCode release_memory(int buffer_id);
HHALExitCode release_event(int event_id);
```

The resource's target architecture manager is dispatched the allocation/deallocation action and it takes care of the accelerator specific requirements for allocating resources.

Buffer Actions

Memory Buffers that are allocated in an accelerator have to be capable of being written and read. The HHAL API provides write and read functions that take the buffer's id, a pointer to a source/destination buffer in the host memory space and the size of the buffer.

Listing 3.19: *HHAL API - Buffer actions*

```
HHALExitCode write_to_memory(int buffer_id, const void *source, size_t
    size);
HHALExitCode read_from_memory(int buffer_id, void *dest, size_t size);
```

Communication with the corresponding accelerator is handled by the architecture's manager.

Event Actions

Events that are allocated in an accelerator require the capability of being written and read. The HHAL API provides a write function that takes the event's id and the data to write, and a read function that takes the event's id and a pointer to host memory where to read the data into.

Listing 3.20: *HHAL API - Event actions*

```
HHALExitCode write_sync_register(int event_id, uint32_t data);
HHALExitCode read_sync_register(int event_id, uint32_t *data);
```

Once again, accelerator specifics are handled by the architecture's manager. In some cases (like Nvidia), events are handled entirely on the Nvidia Manager, as the Nvidia Architecture Node does not provide events support.

Kernel Actions

Before a Kernel is run, its source first has to be written into the accelerator's memory. The HHAL API provides a kernel write function that takes the kernel's id and a map of unit to sources, out of which the previously assigned unit's source is taken.

Multiple kernel's source types are supported by HHAL, this is further explained in section 3.5.3, Dynamic Compiler.

Listing 3.21: *HHAL API - Kernel actions*

```
HHALExitCode kernel_write(int kernel_id, const std::map<Unit,
    hhal_kernel_source> &kernel_sources);
HHALExitCode kernel_start(int kernel_id, const Arguments &arguments);
```

After a Kernel is written and its dependencies are correctly set up, it can be run, or "started" via the kernel start function. As kernels require argument support, this function not only takes the kernel's id but also an **Arguments** object which consists of a vector of kernel arguments.

Three types of arguments are supported by HHAL: Scalar arguments, Buffer arguments and Event arguments. These are the arguments wrapped by Libmango for the user to provide (as seen in section 3.3.2).

Scalar arguments consist of a scalar value belonging to one of the supported types. These are signed and unsigned integers of sizes 8, 16, 32 and 64 bits, as well as floating point values.

The Buffer and Event arguments consist of the respective's Buffer or Event id, which is used to pass the corresponding structure information to the Kernel.

3.5.3 Dynamic Compiler

In previous implementations of the MANGO system, a user provided Kernel had to be pre-compiled into a binary file (or the format required by the target accelerator) before its utilization in the MANGO system. Despite the optimality of this process, it puts limits on the agile and iterative nature of the development of software solutions.

Giving developers the ability to work directly with kernel source code greatly facilitates the development process. Furthermore, having access to the source code would ultimately enable MANGO to apply optimizations by leveraging runtime system information.

The Dynamic Compiler included in the HHAL module offers the functionality of "dynamic" compilation of kernels. That is, the developer provides a kernel's source code to be compiled as required before being written into its target accelerator's memory.

Usage and Implementation

As previously mentioned, HHAL supports multiple types of kernel sources:

Listing 3.22: *HHAL API - Kernel source types*

```
enum class source_type {
    BINARY,
    SOURCE,
    STRING
};
```

The `BINARY` type is for pre-compiled kernels in binary format. While `SOURCE` and `STRING` refer to source code in a file or in a string in memory respectively.

The Dynamic Compiler is automatically used by HHAL when either a SOURCE or a STRING source type is provided in the kernel write function call.

The usage of the Dynamic Compiler is simple, requiring only the instantiation of a `Compiler` object, and a single `get_binary()` function call to obtain a kernel's binary from its source.

Listing 3.23: *HHAL Dynamic Compiler - Compiler class*

```
class Compiler {
    public:
        Compiler();
        const std::string get_binary(const std::string source, hhal::Unit
            arch);

    // Omitted: Private definitions
};
```

Internally, the Dynamic Compiler has a set of alternatives to work with, depending on its configuration and the target architecture of a specific source.

For kernel sources belonging to the C family, the Clang frontend for the LLVM project [72] is utilized if enabled and installed in the user's machine.

Otherwise, compilation is done through system calls as specified in the configuration, utilizing a compiler tool installed in the user's machine.

In the case of CUDA kernels (targeting Nvidia accelerators), the CUDA Compiler tool (covered in section 3.6.1) from the Nvidia Architecture Node is utilized for compilation. This tool exploits the CUDA runtime compilation library NVRTC to compile CUDA kernels into a ptx format as required by the Nvidia Architecture Node.

Configuration

On initialization, the Dynamic Compiler reads a configuration file located in its installation directory. Through this configuration, the user can specify the tools to utilize for the compilation process.

Listing 3.24: *HHAL Dynamic Compiler - Configuration example*

```
[compiler]
expiration=86400

[GN]
```

```
libclang=false
path=cc

// Syntax
[ARCHITECTURE_NAME]
libclang=true_or_false
path=path_to_compiler
```

In the previous configuration example, under `[compiler]`, the `expiration` time for compilation related kernel files is specified in seconds, this parameter is further explained in the following subsection, Caching.

Then, for each supported architecture (`[ARCHITECTURE_NAME]`), two parameters can be specified:

- **libclang:** Whether the Clang compiler should be used when compiling kernels for this architecture.
- **path:** The path to a compiler tool installed in the user's machine to be used when compiling kernels for this architecture. Also works as a fallback option if libclang is not available in the system.

Caching

The Dynamic Compiler implements a caching mechanism to avoid the re-compilation of previously used kernels, exchanging disk space for processing time.

When a kernel is compiled, the compilation output is saved into a file under the caching directory specified at installation. Each time a kernel is sent to the Dynamic Compiler for compilation, an internal check is performed to see whether the source file has already been compiled. This is done by checking if there is a compiled kernel file with the same name as the source file, and if the source file has been modified since.

Listing 3.25: *HHAL Dynamic Compiler - Save kernel string to file*

```
const std::string save_to_file(const std::string kernel_string);
```

If the kernel was loaded as a memory string, HHAL first calls the `save_to_file()` utility function, provided by the Dynamic Compiler, which saves the kernel string as a source file, and then uses this file for compilation. The saved source file is named with the hash of the input string, which allows for fast checking if a kernel source has already been saved.

To prevent the problem of ever-growing disk space usage, every time a kernel is sent

for compilation, the Dynamic Compiler runs a procedure to delete unused (expired) files from the caching directory. The `expiration` time can be specified in the Dynamic Compiler configuration file, defaulting to three days (an example can be seen in listing 3.24).

Kernel entry generation

For Kernels that fall under the GN architecture group, an entry point (main function) receiving the kernel's arguments as string values is required, since kernels are executed through a system call.

The Dynamic Compiler is capable of automatically generating the GN entry point, given that the user annotates the source kernel code accordingly, like in the following sample.

Listing 3.26: *HHAL Dynamic Compiler - Kernel source annotation sample*

```
#include "dev/mango_hn.h"
#include "dev/debug.h"
#include <stdlib.h>

#pragma mango_gen_entrypoint

#pragma mango_kernel
void kernel_function(int a, float *x, float *y, float *out, int n) {
    for (int i=0; i<n; i++) {
      out[i] = a * x[i] + y[i];
    }
}
```

Two pragmas are required for the entry point generation.

The first `#pragma mango_gen_entrypoint` indicates that the entry point is to be generated. The second, `#pragma mango_kernel` must be placed a line before the function to be called by the generated entry point, as its arguments are the ones received by the main function.

The final result is shown in the following listing.

Listing 3.27: *HHAL Dynamic Compiler - Entrypoint generation result*

```
#include "dev/mango_hn.h"
#include <stdlib.h>
extern void kernel_function(int a, float *x, float *y, float *out, int n);

int main(int argc, char **argv){
  mango_init(argv);
```

```
  int a = strtol(argv[6],NULL,16);
  float * x = (float *)mango_memory_map(strtol(argv[7],NULL,16));
  float * y = (float *)mango_memory_map(strtol(argv[8],NULL,16));
  float * out = (float *)mango_memory_map(strtol(argv[9],NULL,16));
  int n = strtol(argv[10],NULL,16);

  kernel_function(a, x, y, out, n);

  mango_close(42);
}
```

3.5.4 Manager example: NVIDIA Manager

HHAL was developed from the ground up with the goal of facilitating its extension via the addition of support for new architectures.

Each new architecture requires a manager that implements the HHAL API function calls for that architecture, acting as a bridge between the MANGO system and the particular target accelerator's library.

Modifications to the existing HHAL code resolve to the addition of the new architecture type and calls to the manager when dispatching the multiple API actions.

In this section, we will go over the Nvidia Manager to exemplify the implementation of an HHAL Manager. The Nvidia Manager communicates with the Nvidia Architecture Node (further details in section 3.6), referenced in code as `CudaApi`, which is the library implemented for launching kernels in Nvidia GPUs.

Listing 3.28: *HHAL Nvidia Manager - Manager Class*

```
class NvidiaManager {
    public:
        NvidiaManagerExitCode assign_kernel(nvidia_kernel *info);
        NvidiaManagerExitCode assign_buffer(nvidia_buffer *info);
        NvidiaManagerExitCode assign_event(nvidia_event *info);

        NvidiaManagerExitCode deassign_kernel(int kernel_id);
        NvidiaManagerExitCode deassign_buffer(int buffer_id);
        NvidiaManagerExitCode deassign_event(int event_id);

        NvidiaManagerExitCode
            kernel_write(int kernel_id, std::string image_path);
        NvidiaManagerExitCode
            kernel_start(int kernel_id, const Arguments &arguments);

        NvidiaManagerExitCode allocate_memory(int buffer_id);
```

```
        NvidiaManagerExitCode allocate_kernel(int kernel_id);
        NvidiaManagerExitCode allocate_event(int event_id);

        NvidiaManagerExitCode release_memory(int buffer_id);
        NvidiaManagerExitCode release_kernel(int kernel_id);
        NvidiaManagerExitCode release_event(int event_id);

        NvidiaManagerExitCode
            write_to_memory(int buffer_id, const void *source, size_t size);
        NvidiaManagerExitCode
            read_from_memory(int buffer_id, void *dest, size_t size);
        NvidiaManagerExitCode
            write_sync_register(int event_id, uint32_t data);
        NvidiaManagerExitCode
            read_sync_register(int event_id, uint32_t *data);

    private:
        void launch_kernel(int kernel_id, char *arg_array, int arg_count,
            char* scalar_allocations);

        std::map<int, nvidia_kernel> kernel_info;
        std::map<int, nvidia_buffer> buffer_info;
        std::map<int, nvidia_event> event_info;

        std::map<int, std::string> kernel_function_names;
        ThreadPool thread_pool;
        EventRegistry registry;
        CudaApi cuda_api;
};
```

The public function definitions mirror the HHAL API functions, which delegate the execution to the indicated manager.

Observing the assignment functions, the received arguments are pointers to structures defined by the Nvidia Manager. As mentioned in the HHAL API section 3.15, these structures are extensions of the base structures used in the HHAL API, and they represent the three core elements of the MANGO system (kernels, buffers and events) with the addition of the information required by the Nvidia Architecture Node.

The Manager keeps track of the assigned elements in three different maps, one for each element, mapping the element's id to their respective extended data structure.

Listing 3.29: *HHAL Nvidia Manager - Kernel assignment*

```
NvidiaManagerExitCode NvidiaManager::assign_kernel(nvidia_kernel *info)
{
    kernel_info[info->id] = *info;
```

```
    return NvidiaManagerExitCode::OK;
}
```

In assign/deassign functions, elements get added or removed from the maps respectively.

Buffers

Buffer allocation follows a simple procedure. The Nvidia Architecture Node takes a buffer id and size as parameters for the memory allocation call, so the manager just delegates the allocation given the information received at assignment time.

Listing 3.30: *HHAL Nvidia Manager - Kernel assignment*

```
NvidiaManagerExitCode NvidiaManager::allocate_memory(int buffer_id) {
    nvidia_buffer &info = buffer_info[buffer_id];
    CudaApiExitCode err = cuda_api.allocate_memory(info.mem_id, info.size);
    //error handling...
}
```

A similar course of action is followed in memory read and write operations, delegating their execution to the architecture library.

Events

Since the Nvidia Architecture Node does not support events, the Nvidia Manager handles them internally. This is carried out using an event registry, which implements the basic event functionalities.

Listing 3.31: *HHAL Nvidia Manager - Event registry*

```
class EventRegistry {
    public:
        EventRegistryExitCode add_event(int event_id);
        EventRegistryExitCode remove_event(int event_id);
        EventRegistryExitCode read_event(int event_id, uint32_t *data);
        EventRegistryExitCode write_event(int event_id, uint32_t data);
    private:
        std::mutex registers_mtx;
        std::map<int, uint32_t> registers;
};
```

Since event arguments are not supported for Nvidia kernels, only kernel termination events are handled.

Kernels

Aside from assignment and allocation, there are two main actions the manager must implement regarding kernels: write and start.

`kernel_write()` writes the kernel image in the accelerator's memory space. The Nvidia Manager also extracts the function name from the provided kernel, as it is required on launch.

`kernel_start()` starts kernel execution in the target accelerator, with two relevant considerations: argument handling and non-blocking kernel launch.

The target architecture's library likely define their own kernel arguments, as is the case for the Nvidia Architecture Node, so they must be translated from the HHAL version into their Nvidia Architecture Node counterpart.

Listing 3.32: *HHAL Nvidia Manager - Kernel arguments translation*

```
NvidiaManagerExitCode NvidiaManager::kernel_start(int kernel_id, const
    Arguments &arguments) {
    // Omitted: Function setup
    char *current_arg = arg_array; //Base of arguments array to be sent to
        NAN
    for(auto &arg: args) {
        switch (arg.type) {
            case ArgumentType::BUFFER: {
                auto &b_info = buffer_info[arg.buffer.id];
                // Omitted: Check if buffer is an input buffer...
                auto *arg_x = (cuda_manager::BufferArg *) current_arg;
                *arg_x = {cuda_manager::BUFFER, b_info.id, is_in};
                current_arg += sizeof(cuda_manager::BufferArg);
                break;
            }
            case ArgumentType::SCALAR: {
                hhal::scalar_arg scalar = arg.scalar;
                auto *arg_a = (cuda_manager::ScalarArg *) current_arg;
                // Omitted: Allocate scalar value locally so it's not
                    deallocated until kernel terminates...
                *arg_a = {cuda_manager::SCALAR, (void *)allocated_scalar};
                current_arg += sizeof(cuda_manager::ScalarArg);
                break;
            }
            // Omitted: EVENT argument translation and kernel launch
}
```

Since the nature of MANGO requires kernel launching to be non-blocking, for each kernel start action a task executing the private `launch_kernel()` procedure

is added to an internal thread pool.

Listing 3.33: *HHAL Nvidia Manager - Kernel launch*

```
NvidiaManagerExitCode NvidiaManager::kernel_start(int kernel_id, const
    Arguments &arguments) {
    // Omitted: Argument translation

    // Push launch_kernel task
    thread_pool.push_task(std::bind(&NvidiaManager::launch_kernel, this,
        kernel_id, arg_array, arg_count, scalar_allocations));

    return NvidiaManagerExitCode::OK;
}

void NvidiaManager::launch_kernel(int kernel_id, char *arg_array, int
    arg_count, char* scalar_allocations) {
    nvidia_kernel &info = kernel_info[kernel_id];

    CudaResourceArgs r_args = {info.gpu_id,
                    {info.grid_dim_x, info.grid_dim_y, info.grid_dim_z,
                    {info.block_dim_x, info.block_dim_y, info.block_dim_z}};

    auto &termination_event = event_info[info.termination_event];
    CudaApiExitCode err = cuda_api.launch_kernel(kernel_id,
        kernel_function_names[kernel_id].c_str(), r_args, arg_array,
        arg_count);

    // Omitted: Deallocation and error handling...

    // Notify kernel termination
    write_sync_register(termination_event.id, 1);
}
```

Worker threads continuously execute pending tasks from the tasks queue managed by the thread pool. The `launch_kernel()` function calls the Nvidia Architecture Node (blocking) procedure to execute the kernel and passes the necessary arguments. Once the execution finishes, the kernel's termination event is notified.

3.6 Nvidia Architecture Node

The Nvidia Architecture Node (NAN) is the bridge between the MANGO system, specifically the HHAL module, and the low-level Nvidia libraries utilized to communicate with Nvidia GPUs.

The main focus of the NAN is enabling MANGO to launch CUDA kernels in Nvidia

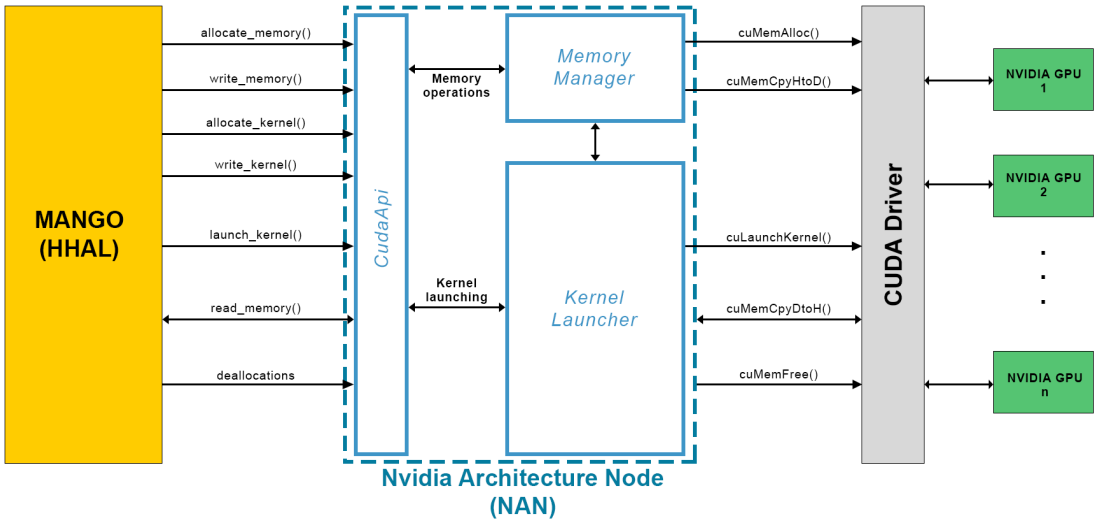


Figure 3.10: *Nvidia Architecture Node flow*

accelerators. As shown in figure 3.10, the NAN exploits the CUDA Driver API [73] library, which exposes a variety of functions that allow low-level interfacing with the system-available Nvidia devices.

3.6.1 CUDA Kernels

Nvidia GPUs are capable of executing kernels written in the CUDA programming language, which are compiled into the PTX [74] format. PTX stands for Parallel Thread Execution, and is an Instruction Set Architecture (ISA) for using Nvidia GPUs as data-parallel computing devices.

Naturally, users of the MANGO system targeting Nvidia GPUs must provide CUDA kernels in their PTX compiled form when loading kernels of the binary type. Alternatively, if kernels are loaded as source code, the Nvidia Architecture Node provides the **CUDA Compiler** tool, which makes use of the CUDA compilation library NVRTC [75] to compile CUDA kernels from source into a PTX format for its later execution. The CUDA Compiler is used by the Dynamic Compiler of the HHAL Module for this very purpose.

3.6.2 NAN API

Listing 3.34: *Nvidia Architecture Manager - CudaApi*

```
class CudaApi {
public:
  CudaApi();
```

```
  ~CudaApi();

  CudaApiExitCode allocate_memory(int buffer_id, size_t size);
  CudaApiExitCode deallocate_memory(int buffer_id);
  CudaApiExitCode
    write_memory(int buffer_id, const void *data, size_t size);
  CudaApiExitCode
    read_memory(int buffer_id, void *dest_buffer, size_t size);

  CudaApiExitCode allocate_kernel(int kernel_id, size_t size);
  CudaApiExitCode deallocate_kernel(int kernel_id);
  CudaApiExitCode
    write_kernel(int kernel_id, const void *data, size_t size);

  CudaApiExitCode launch_kernel(int kernel_id, const char *function_name,
      CudaResourceArgs resource_args, const char *args, int arg_count);
};
```

The Nvidia Architecture Manager exposes its API through the `CudaApi` class. As observed in the listing 3.34, the API presents a small number of functions that directly correspond with the basic functionality needed by MANGO for a single architecture.

The first four listed functions are related to memory buffers management. `allocate_memory()` and `deallocate_memory()` perform the allocation and deallocation of a memory buffers in the target device. A buffer is identified by an user-provided id integer, from the time of a buffer's allocation this id is used for further operations regarding said buffer until deallocation.

`write_memory()` and `read_memory()` allow writing to and reading from an allocated buffer, in each case a source or destination memory location is required.

The following functions work in a similar manner but focus on kernel allocation and writing. The `write_kernel()` function takes a pointer to the start of the kernel PTX loaded in memory and its size.

The `launch_kernel()` function starts the execution of a kernel in the target device, taking the kernel entry function name and a set of arguments regarding both the resource requirements and the user arguments for the kernel itself. In the following subsections, resource and kernel arguments are explained in further detail.

Resource arguments

Listing 3.35: *Nvidia Architecture Manager - Resource arguments*

```
// Dimensions for grid and blocks
```

```
struct CudaDims {
  uint32_t x;
  uint32_t y;
  uint32_t z;
};

// Resource specific arguments for kernel launch
struct CudaResourceArgs {
  int device_id;
  CudaDims grid_dim;
  CudaDims block_dim;
};
```

Resource arguments, grouped in the `CudaResourceArgs` structure as shown in listing 3.36, allow the specification of thread block and grid dimensions when launching a kernel, as well as the target device's id. An explanation of the execution configuration of CUDA kernels is available in section 2.1.6 of the State of the Art chapter, covering Nvidia CUDA.

Kernel arguments

Listing 3.36: *Nvidia Architecture Manager - Kernel arguments*

```
enum ArgType {
  BUFFER,
  SCALAR
};

struct Arg {
  ArgType type;
};

struct ScalarArg {
  ArgType type;
  void *ptr;
};

struct BufferArg {
  ArgType type;
  int id;
};
```

The `const char *args` argument of the `launch_kernel()` API function is a pointer to an array of kernel arguments in the order the kernel receives them. Two types of kernel arguments are supported by the NAN, `BUFFER` and `SCALAR` argu-

ments. Scalar arguments contain a reference to the scalar value allocated in host memory, while buffer arguments consist of the integer id of an allocated memory buffer. When parsing arguments before being sent to the GPU, arguments are interpreted as the base struct `Arg`, and then casted to either `ScalarArg` or `BufferArg` depending on their type.

3.7 HN and GN

In previous versions of MANGO, the HN module acted as the hardware abstraction layer akin to the current HHAL module.

The HN module consists of an HN library that exposes the HN API, and an HN daemon that manages the concurrent access to the HN supported platforms and provides the physical connection the the HN system. Other modules of the MANGO system that interact with HN must link to the HN library, which acts as a client of the daemon.

3.7.1 Supported architectures

HN was developed to support heterogeneous systems consisting of FPGA clusters containing PEAK, nu+ and DCT accelerators.

PEAK

PEAK stands for Partitioned Enabled Architecture for Kilocores and it is a research manycore prototype for generic computing. The main goal of PEAK within MANGO is to offer a configurable processor able to be adapted to different configurations and capabilities, thus enabling exploration of adaptations to the different target applications in the project [2]. The processor has the following key characteristics:

- Runs a large set of MIPS R32 ISA instructions.
- Can be instantiated to any given number of cores, restricted only to the resources available in the targetFPGA.
- Implements private L1 caches to each core and a shared bank set of L2 caches.
- Supports shared memory by implementing an invalidation-based coherence protocol (MESI).
- Implements a sophisticated Network-on-Chip enabling communication between cores, L1 caches, L2 banks, configuration registers, and memory.

- Supports exceptions and interrupts.

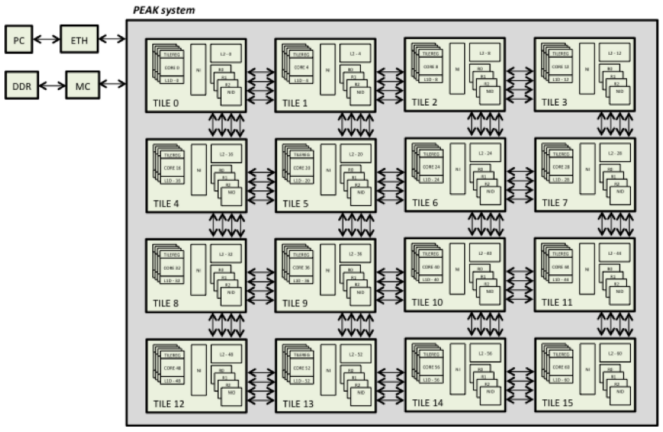


Figure 3.11: *PEAK architecture [2]*

nu+

As described in [2], nu+ is a complex and configurable GPU-like accelerator core, allowing flexible customization driven by application requirements. It is designed to support the exploration of advanced architecture features deviating from current general-purpose heterogeneous architectures. In particular, it offers the following key characteristics:

- Data-level parallelism through large-size vector/SIMD support.
- Multi-core organization allowing non-SIMT execution.
- Advanced mesh-based Network-on-Chip.
- Lightweight control flow constructs exposed to the programmer.
- Hybrid memory hierarchy providing both coherent caches and non-coherent scratch-pad memory.
- Non-standard floating-point precision values as well as dedicated functions like fused operators.

A nu+ compiler based on the LLVM project was developed from scratch, as explored in [76].

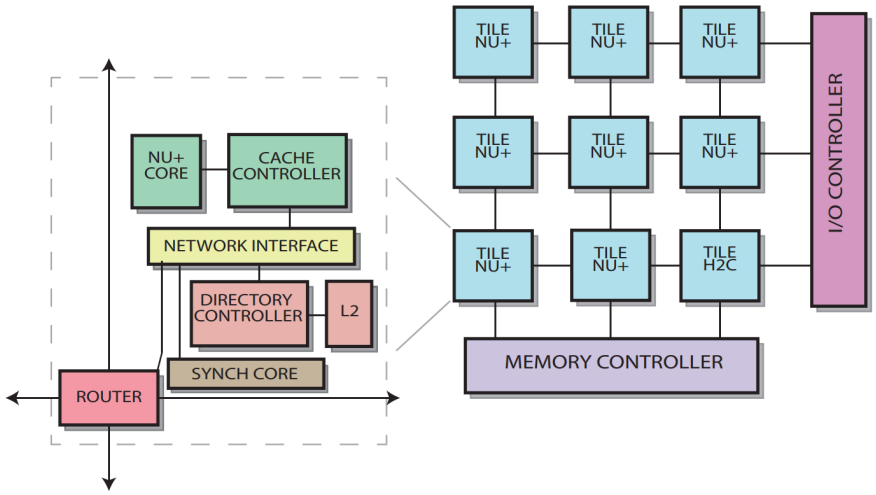


Figure 3.12: *nu+ architecture [2]*

DCT

DCT accelerators are passive accelerators designed to run a single kernel, so their initialization and usage differ from the other supported accelerators. Because of its passive nature, the accelerator requires input buffers, output buffers and synchronization memory locations to start at a pre-defined virtual memory address, which has to be retrieved through the HN API [76].

3.7.2 Hardware Emulation

The GN library emulates the presence of hardware accelerators by running kernels in the host CPU. GN was developed as an HN emulator, so it respects the same API as the HN library, which allows to use them interchangeably.

Kernels targeting the GN platform require binaries compiled for the system's CPU architecture, so their implementation language is irrelevant as long as a working binary is generated.

Furthermore, kernel execution is performed through system calls. As a consequence, the kernel entry function must receive its parameters as an array of characters.

As previously covered, the Dynamic Compiler (section 3.5.3) of the HHAL module is capable of generating kernel entrypoints for kernels written in the C language .

CHAPTER 4

Experimental Results

In this Chapter we will cover the experimental phase of our work. First comes a description of our setup and methodology, covering the hardware and software utilized as well as the chosen benchmarks and metrics. Then we will present our results and provide our analysis in order to fuel further research and development in the MANGO software stack.

During this experimental campaign, our goals were to:

- Compare MANGO with other available programming models, in terms of performance and programmability.
- Analyze where the main overheads of the current MANGO implementation are and how they can be reduced.
- Evaluate our communication scheme between the Libmango API and the HHAL client, and how it can be improved.

4.1 Setup and Methodology

All our tests were run on a Dell XPS 9570 laptop. This particular model is equipped with an Intel Core i7 8750H CPU, 16 GB RAM and an Nvidia GeForce GTX 1050 Ti GPU with Max-Q Design (4GB VRAM). In terms of Operating System, the machine runs Ubuntu 18.04.2 LTS 64-bit (Kernel Version: 5.4.0-48-generic).

For reproducibility, the utilized software versions were the following:

- CUDA 11.0
- OpenCL 1.2 CUDA (Driver version: 450.51.06)
- gcc 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)
- clang 6.0.0-1ubuntu2
- Python 3.6.9
- Cython 0.29.23

In order to compare the performance of MANGO to CUDA and OpenCL we used a subset of the Rodinia Benchmark Suite (Version 3.1) [77], provided by the University of Virginia. In particular, we chose the HotSpot and PathFinder benchmarks.

HotSpot (HS) is a thermal simulation tool, presented in [78], which is used for estimating processor temperature based on an architectural floor plan and simulated power measurements. The benchmark includes the 2D transient thermal simulation kernel of HotSpot, which iteratively solves a series of differential equations for block temperatures. The inputs to the program are power and initial temperatures of a square N x N grid. Each output cell in the grid represents the average temperature value of the corresponding area of the chip.

During our experiments, the size of the Grid (N) was progressively increased in order to increase the load on the GPU. The number of threads spawned for computation as well as the size of the input buffers and output buffers scale with $O(N^2)$. The number of kernel executions is determined by the amount of iterations desired. This parameter is kept fixed, resulting in 16 kernel executions to compute 128 iterations of the algorithm for all HotSpot benchmarks.

PathFinder (PF) uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node

picks a neighboring node in the previous row that has the smallest accumulated weight, and adds its own weight to the sum.

Like with HotSpot, we also increased the size of the input grid (N) in order to increase the GPU load when running the PathFinder benchmarks. The dimensions of the grid were also kept symmetrical, as in the number of columns is equal to the number of rows (N x N). In this case, increasing the number of columns increases the number of threads spawn to compute the shortest path. The number of rows, on the other hand, increases the number of kernel executions. In addition, the size of the grid increases the dimensions of the input buffer with $O(N^2)$, the output however only increases linearly, with O(N), as only the values of the columns are needed.

During our testing we realized that PathFinder most certainly lacks global GPU memory access optimizations. This is indicated by its poor memory bandwidth performance coupled with the fact that the algorithm requires continuous access to the input rows in order to calculate each iteration. In addition, the operations performed by pathfinder are simple comparisons and a single integer addition. It is important to also note that these comparisons are integer minimum operations, which do not generate branches [79].

We still decided to keep PathFinder in our comparison as it is a good example of what happens when there is a need to run a kernel multiple times in a single benchmark and how the different systems scale with increasing numbers of kernel launches.

Finally, in order to cover our lack of memory bound applications due to the issues mentioned with PathFinder, we added an AXPY benchmark. AXPY stands for "A X plus Y", as noted by the name, it performs the following operation:

$$z \leftarrow a * x + y$$

This same operation is also implemented in BabelStream [80], a memory bandwidth benchmark for heterogenous systems based on STREAM [81], which names it "Triad".

A breakdown of the three benchmarks chosen can be seen in table 4.1.

Benchmark	Dwarves	Performance characteristic
HotSpot	Structured Grid	Compute intensive
PathFinder	Dynamic Programming	Multiple kernel launches
AXPY	Basic Linear Algebra	Memory intensive

Table 4.1: *Breakdown of benchmarks used*

4.1.1 Performance metrics

To evaluate performance in both HotSpot and PathFinder, we will measure:

- Total execution time: Time between the beginning of the proper benchmark (i.e. after all input initialization and I/O) and the end (after all resource deallocations, but before checking output results).
- Kernel execution time: Time between the launch of a kernel and the end of its execution.
- Buffer write time: Time to write data to the device, i.e transfer data from the host to the device.
- Buffer read time: Time to read data from the device, i.e. transfer data from the device to the host.

For AXPY, we will measure the performance of the different systems as the percentage of theoretical peak bandwidth they can achieve. We will take into account both GPU memory bandwidth, measuring accesses to GPU global memory, and PCI-E bandwidth, measuring data transfers between the Host and the Device and vice versa.

For GPU memory, the theoretical peak bandwidth is given by plugging the memory clock rate and bus width into the following formula:

$$GBW_{peak} = \frac{C * 10^6 * (B/8) * 2}{10^9}$$

Where:

- GBW_{peak} is the theoretical peak GPU VRAM bandwidth in GB/s.
- C is the memory clock rate in MHz.
- B is the memory bus width in bits.

With a memory clock rate of 3504 MHz and 128-bit of bus width, the GTX 1050 Ti in our test bench achieves a theoretical peak of 112.128 GB/s.

To measure the effective bandwidth achieved by each model we use the following formula:

$$GBW_{effective} = \frac{R_B + W_B}{t * 10^9}$$

Where:

- $GBW_{effective}$ is the effective GPU VRAM bandwidth in GB/s.
- R_B is the number of bytes read per kernel.
- W_B is the number of bytes written per kernel.
- t is the kernel execution time in seconds.

The two previous formulae were taken from [82].

In terms of PCI-E bandwidth, our video card presents a PCI-E 3.0 x16 connection which could achieve a peak bandwidth of 15.754GB/s.

In a very similar way as how we measure effective bandwidth for GPU memory, we can measure the effective transfer bandwidth:

$$TBW_{effective} = \frac{T_B}{t * 10^9}$$

Where:

- $TBW_{effective}$ is the effective transfer bandwidth in GB/s.
- T_B is the number of bytes transferred.
- t is the transfer time in seconds.

In AXPY we are not interested in the scaling over different input sizes but in the bandwidths achieved with large inputs. A large input maximizes the amount of memory to transfer and access, meaning that a larger part of the available bandwidth can be exploited.

It is important to note that for all benchmarks in both CUDA and OpenCL we execute the set of benchmarks twice. Once to measure the total execution time of the entire benchmark and another to measure each individual component. As all our measurements in these models are done externally (i.e. we cannot make intrusive profiling modifications like with MANGO) they could incur in extra overhead due to the need of forcing synchronization between the host and the device in order to make measurements. For example, OpenCL buffer writes are usually enqueued in a `CommandQueue` and executed asynchronously, to measure them we forced a synchronous transfer by calling `clFinish` on the `CommandQueue`.

In the case of MANGO, to evaluate its performance and compare it directly with CUDA and OpenCL we perform profiling of the application at the HHAL server, disregarding the overhead added by IPC communication.

4.1.2 IPC

By striving towards a distributed architecture, communication overhead is inevitably introduced in the interaction between MANGO modules. Hence, when considering the platform in its entirety, results are expected to be inferior to the alternatives (CUDA and OpenCL), which work in a local manner.

As previously mentioned, the HHAL library is implemented in a client-server arrangement, where the communication between client and server is performed via an IPC socket, enabling the possibility of running the HHAL in a separate cluster in the future.

To evaluate the communication scheme between Libmango and HHAL, we performed measurements directly on the Libmango API side. These measurements will take the IPC overhead into account, which will be compared directly with the metrics obtained on the HHAL server. The discrepancies between the two sets of values will provide an overview of the aspects of the system that are most affected by IPC communication and how they could be improved.

4.1.3 Programmability

To compare the programmability of each model we count the number of lines of code (LOC) required for each implementation of the HotSpot and PathFinder benchmarks. To compute the LOC we do not take into account comments or blank lines. Also, we ignore any line of code related to debugging, extra code needed to profile the OpenCL and CUDA benchmarks or checking computation results. This last point is particularly significant on the MANGO benchmarks which usually compare their results with the CUDA benchmarks to ensure that the implementation is working correctly. Finally, the original code from the Rodinia Benchmarks was adapted in order to follow a consistent style across all implementations.

In addition to the LOC metric, to have a more direct comparison of the size of the MANGO implementations to the CUDA and OpenCL ones we also use a Relative Difference metric (RD). RD calculates the percentage difference in LOC between MANGO and a different implementation. We define RD as follows:

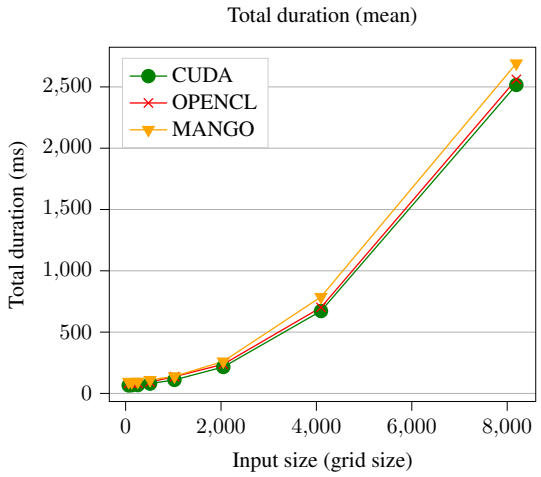


Figure 4.1: *Mean total execution time for HotSpot*

$$RD = \frac{LOC_{impl} - LOC_{MANGO}}{LOC_{MANGO}} * 100$$

Thus, a positive RD indicates that a given implementation requires more LOC than MANGO, while a negative RD indicates that the implementation requires less LOC.

4.2 Results

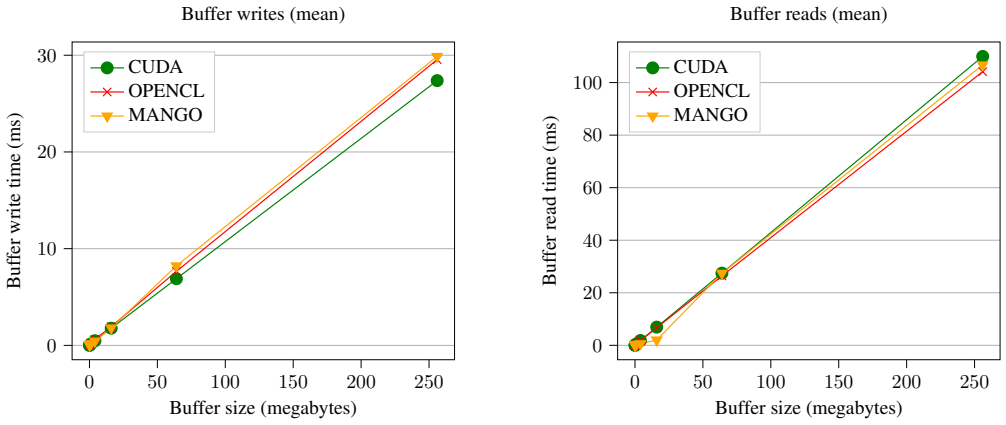
In this section we will present the results of our experiments. First, we will make performance comparisons against CUDA and OpenCL, then we will cover the overhead experimented by the client due to IPC with the HHAL server. Finally, we will analyze the programmability improvements achieved by MANGO over CUDA and OpenCL.

4.2.1 Performance

HotSpot

Starting with HotSpot, we can see that performance across all metrics is very similar for all of the different programming model implementations. In addition, all models follow the same scaling as input size increases.

The overhead presented by MANGO in this benchmark is noticeable, but not significant. For example, when running with a grid size of 8192 x 8192, CUDA is the fastest completing the entire execution in ~2516ms while MANGO takes ~2692ms, a difference of only 6.5%.



(a) *Mean buffer write time*

(b) *Mean buffer read time*

Figure 4.2: *Mean buffer transfer times for HotSpot*

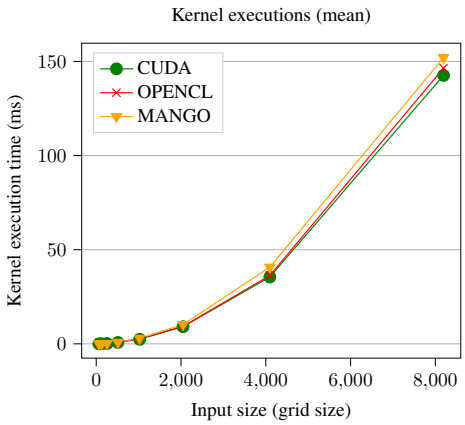


Figure 4.3: *Mean kernel execution time for HotSpot*

When decomposing the benchmark into its different aspects we can see that the overhead is most significant for buffer writes, as seen in figure 4.2. Here CUDA is 8.4% faster than MANGO for the same 256 megabyte transfer. Still, MANGO keeps up the pace with OpenCL, which is only 1% faster.

The most important overhead is actually the one linked to kernel executions, which are performed more frequently than buffer writes (only 2 buffer writes versus 16 kernel executions) and also individually require more time to be performed. In this case, CUDA and OpenCL are 6.2% and 3.7% faster respectively for the largest grid size used. This can be seen in figure 4.3.

The benchmark breakdown figure 4.4 clearly shows the influence of kernel execution time in the overall running time. This breakdown compares how the total running time of the benchmark is distributed among the different aspects of the execution, plus some miscellaneous operations which are not particularly relevant for this benchmark. To

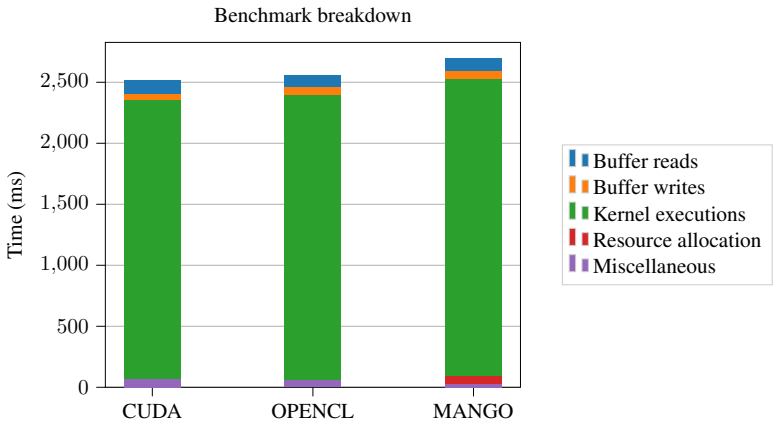


Figure 4.4: *Benchmark breakdown for HotSpot*

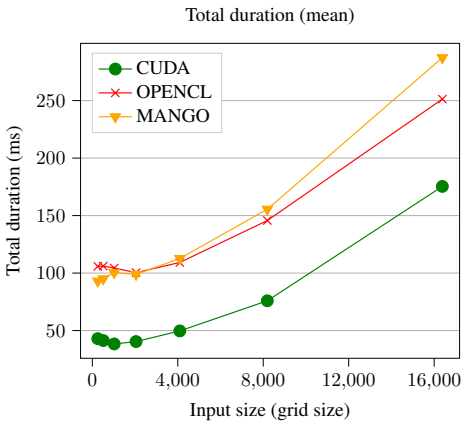


Figure 4.5: *Mean total execution time for PathFinder*

generate these metrics we used the measurements obtained when running the biggest input size for HotSpot, 8192 x 8192. The figure clearly pictures how kernel execution time takes up more than 90% of the total benchmark time.

PathFinder

Moving on to PathFinder, we can see a much more significant difference in the performance of MANGO and OpenCL with respect to CUDA in figure 4.5. In addition, a clear increase in the steepness of the curve can be seen in the case of MANGO, as it steers away from the OpenCL implementation as the input size increases. This would indicate that there is some noticeable overhead present in the kernel executions, as they increase in number along with the input size of PathFinder.

Figure 4.6 confirms our suspicion, as there is a clear separation between the kernel execution times of MANGO and the ones of CUDA and OpenCL, which are quite similar with each other. Comparing the the time taken for the biggest input size, CUDA

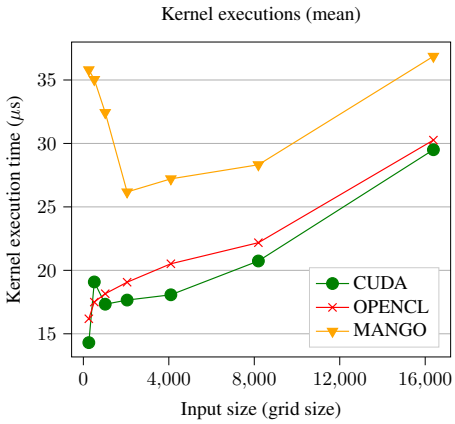
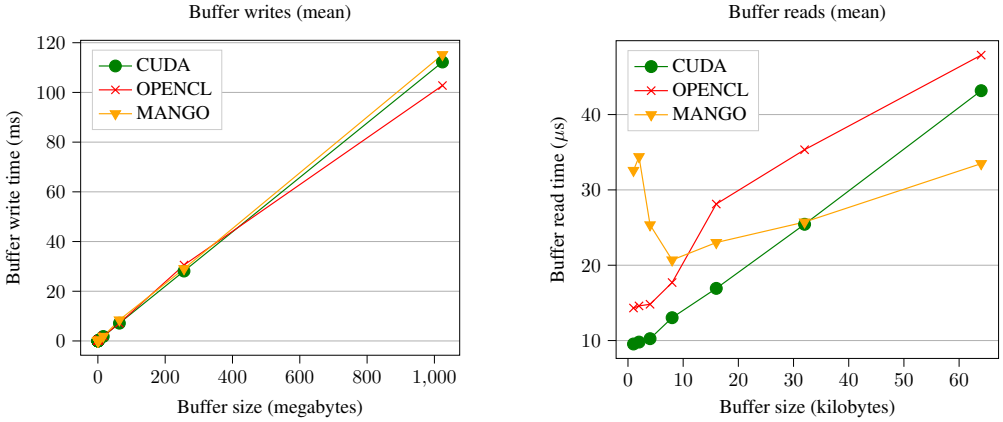


Figure 4.6: *Mean kernel execution time for PathFinder*



(a) *Mean buffer write time*

(b) *Mean buffer read time*

Figure 4.7: *Mean buffer transfer times for PathFinder*

is 20% faster than MANGO for each kernel executed, while OpenCL is 17.9% faster.

Focusing now on buffer transfers, as seen in figure 4.7, OpenCL gains the advantage for the bigger buffer sizes written by PathFinder. Meanwhile MANGO stays in line with the pure CUDA implementation, although with some overhead still present. In terms of buffer reads, MANGO pulls ahead with the largest buffer size, with CUDA and OpenCL being 28.9% and 43% slower respectively. However it is important to note that the buffers read as a result are significantly smaller than the ones written as input, for larger buffer sizes this initial differences could become negligible, as seen for HotSpot in figure 4.2.

Finally, we move on to the benchmark breakdown which, similarly to the one for HotSpot, is generated with the largest input size for PathFinder, 16384 x 16384. Here in figure 4.8 we can clearly see that the factor that most contributes to the difference in performance between the different implementations are actually the miscellaneous

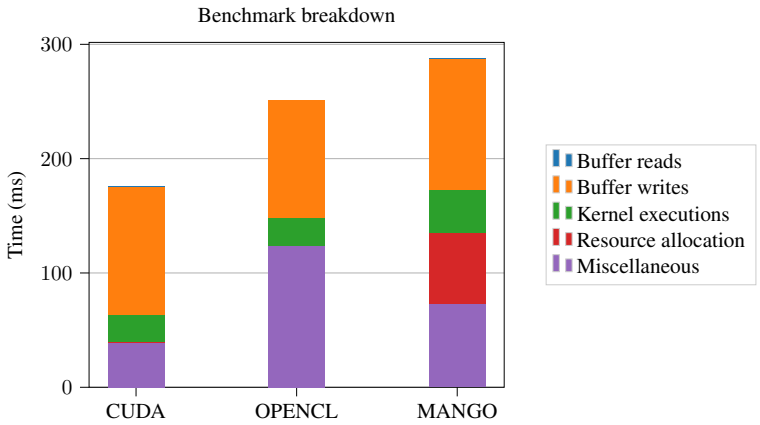


Figure 4.8: *Benchmark breakdown for PathFinder*

tasks (plus resource allocation for MANGO), taking up around 50% of the running time. These include all the boilerplate and preparations required to setup the different execution environments. While this is very minimal in the case of CUDA, as it is a single-source model, where kernel and host code are precompiled and loaded together, the setup is much more significant for MANGO and OpenCL. Both OpenCL and MANGO require loading a separate kernel file, requiring I/O operations. After that OpenCL also adds kernel compilation and device setup, while MANGO requires communication with BarbequeRTRM to perform resource allocation and deallocation. These aspects are not significant for HotSpot as the total running time is much longer and the overhead of the setup is constant, no matter the input size.

AXPY

In AXPY we move straight to the mean kernel execution time in figure 4.9 where we can see that CUDA performs 12.4% faster than MANGO, while OpenCL is only 7% faster for the same size input.

As we know that AXPY is purely memory bandwidth bound, we can translate this kernel execution times to memory bandwidth measurements as seen in figure 4.10. In this figure we can see the percentage of theoretical peak bandwidth achieved by each programming model.

What is interesting is that if we look at the maximum bandwidth achieved by each model MANGO is able to beat both CUDA and OpenCL, being 1% and 4.5% faster than each programming model respectively. This indicates that our implementation does not inherently pose a limit to the performance that could be achieved by the GPU when compared to pure CUDA and the overhead is likely affected by varying external

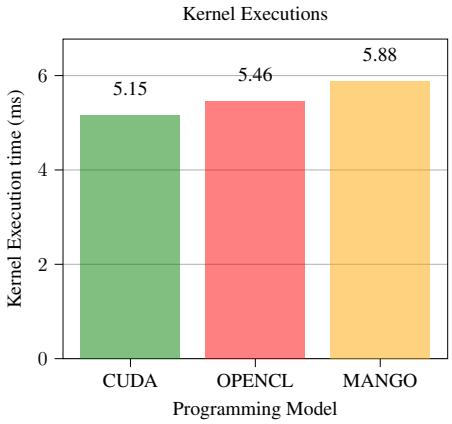


Figure 4.9: *Mean kernel execution times for AXPY*

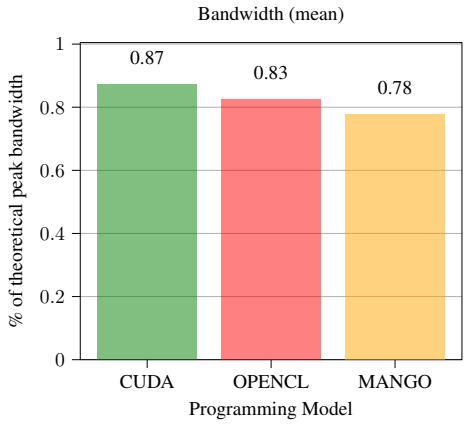


Figure 4.10: *Mean GPU memory bandwidth for AXPY*

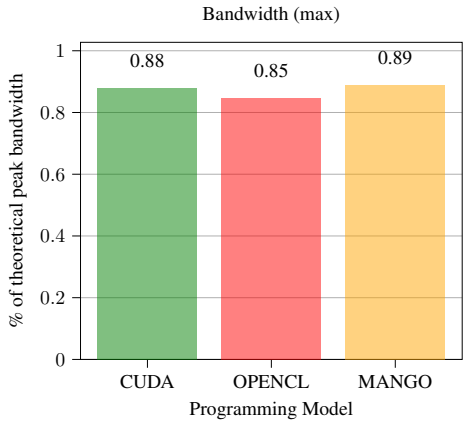
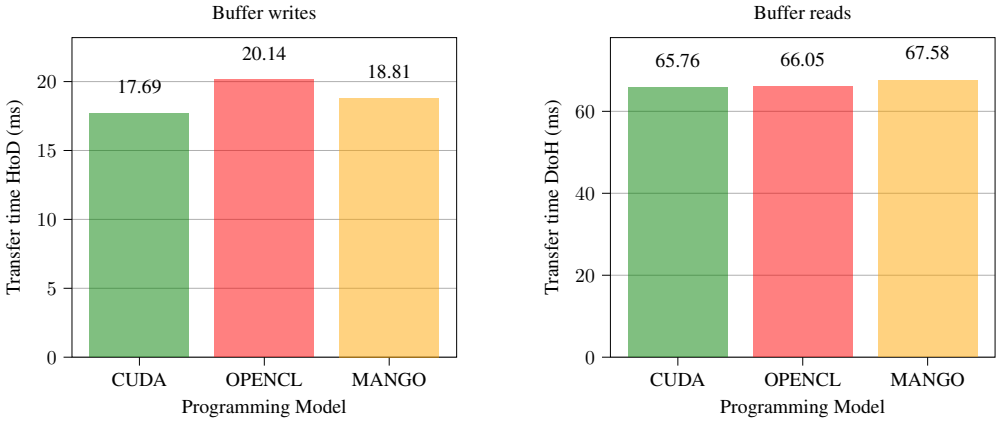


Figure 4.11: *Max GPU memory bandwidth for AXPY*



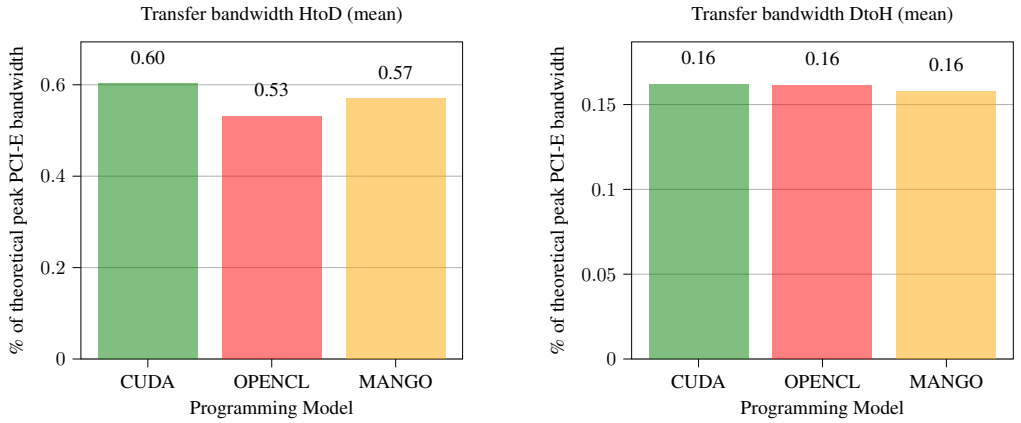
(a) *Mean buffer write time*

(b) *Mean buffer read time*

Figure 4.12: *Mean buffer transfer times for AXPY*

factors, such as background processes or Operating System context switches.

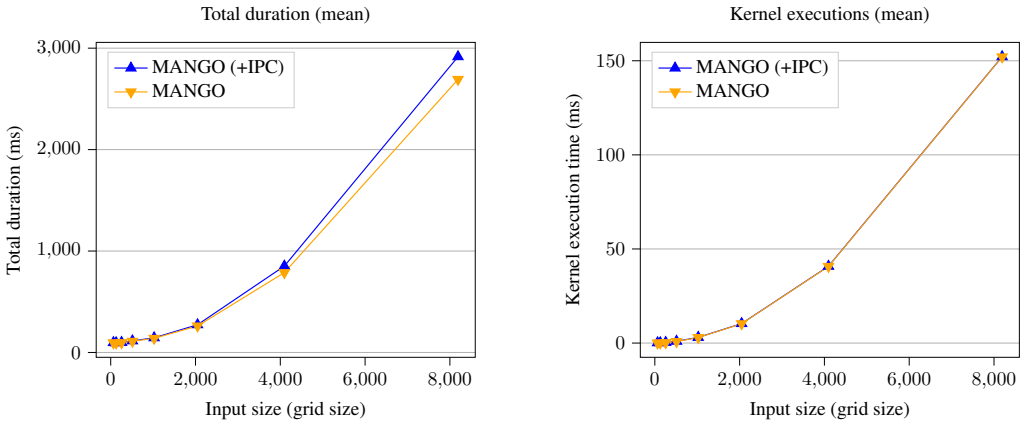
Finally, we look at the transfer times achieved between the host and the device and vice versa, and how these translate to the percentage of PCI-E bandwidth achieved. These comparisons can be seen in figures 4.12 and 4.13 respectively. In terms of device-to-host transfers, the difference in performance is negligible, although none of the models provide an optimal speed for the available PCI-E connection. For host-to-device transfers, MANGO gets second place, with CUDA being 6% faster and OpenCL 7% slower. Still, none of the models achieve close to maximum usage of the PCI-E connection.



(a) *Mean host to device transfer bandwidth*

(b) *Mean device to host transfer bandwidth*

Figure 4.13: *Mean buffer transfer bandwidths for AXPY*



(a) *Mean total execution time*

(b) *Mean kernel execution time*

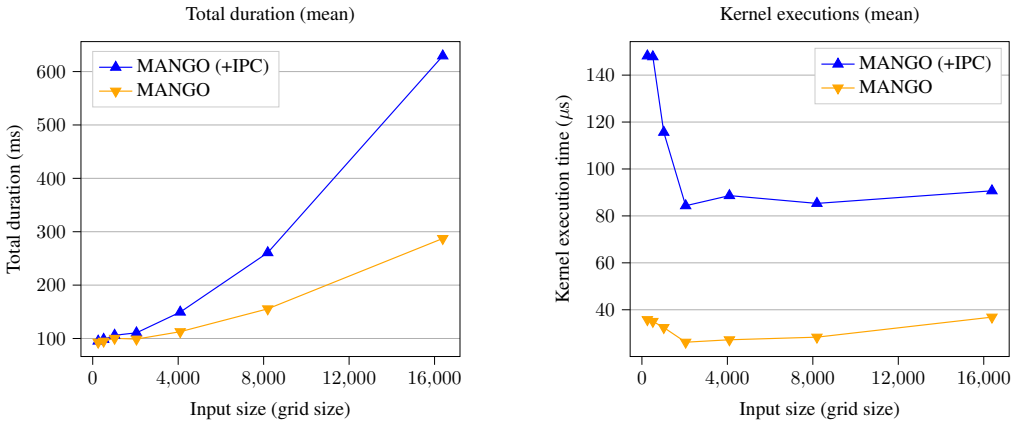
Figure 4.14: *Mean total and kernel execution times for HotSpot*

4.2.2 IPC

As mentioned in section 4.1, the previous benchmarks did not take into account the overhead added by inter process communication between the libmango client and the HHAL server. Here we will focus on this part separately to evaluate our communication scheme and how it can be improved.

Starting with figure 4.14 we can see how the total duration of the HotSpot benchmark increases when taking IPC into account, and how this difference scales with larger input sizes. This however is not linked with the kernel execution time, which is perfectly in line with the benchmark without any IPC.

For the same metrics, PathFinder sees a much steeper increase in total running time



(a) *Mean total execution time*

(b) *Mean kernel execution time*

Figure 4.15: *Mean total and kernel execution times for PathFinder*

along with the execution time. As can be see in figure 4.15b, this is more closely linked with the kernel execution time which here shows a clear separation between IPC and no IPC. In addition, the PathFinder kernel is run more times as the input size increases, so this overhead piles up with multiple executions.

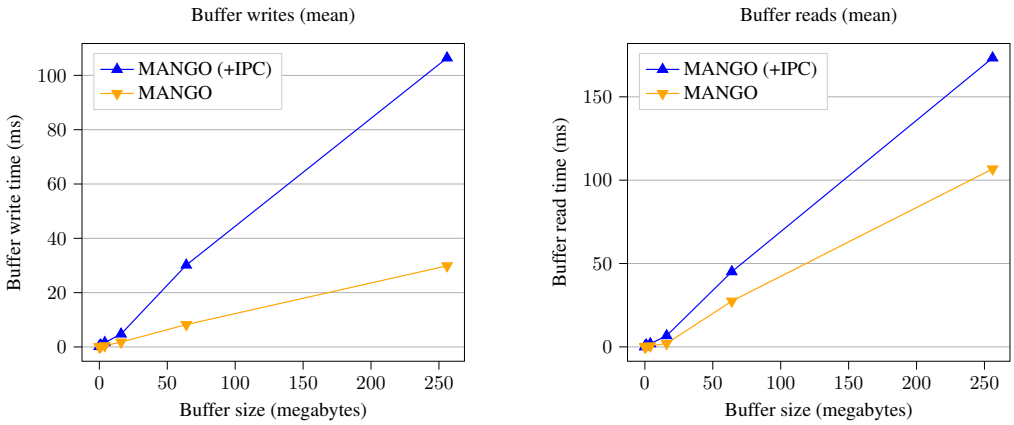
If we compare these results with the ones in HotSpot, we can conclude that the overhead experienced with IPC is fixed and does not scale with input size. This is supported by the fact that HotSpot kernel executions are an order of magnitude larger than PathFinder executions (150ms for HotSpot versus 90/40 μs for PathFinder).

Buffer transfers, as expected, are affected significantly more by IPC as the amount of data that needs to be transferred is much higher than what is needed to launch a kernel. Both figure 4.16 and figure 4.17 show similar scaling for buffer writes. There is a clear discrepancy between the buffer reads, however this can be attributed to the considerable difference in buffer sizes.

4.2.3 Overall Performance Analysis

At first glance, if we include the IPC overhead, our initial implementation of the HHAL and restructuring of the MANGO stack is certainly not competitive with state-of-the-art programming models such as CUDA and OpenCL.

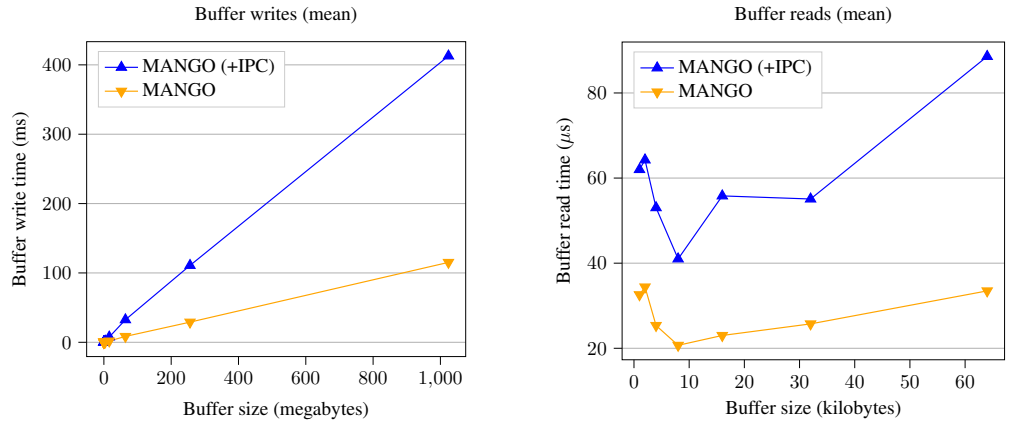
Communicating between multiple processes through sockets unnecessarily reduces performance when working within the same cluster, as more efficient communication could be adopted. If the HHAL, BarbequeRTRM and Libmango run on the same machine, all communication could be done through shared memory instead of a socket.



(a) *Mean buffer write time*

(b) *Mean buffer read time*

Figure 4.16: *Mean buffer transfer times for HotSpot*



(a) *Mean buffer write time*

(b) *Mean buffer read time*

Figure 4.17: *Mean buffer transfer times for PathFinder*

In addition, kernel executions and buffer transfers could be done entirely through a library linked directly to the MANGO host, without the need of any IPC. Allowing the computer to work on in this sort of "hybrid" mode would let MANGO achieve throughput inline with the initial metrics we presented, which do not take into account IPC. Although this discussed approach works for a single cluster, it is not extensible to a distributed system. Hence, to accomplish a flexible starting point the current implementation of the system was carried out using IPC socket communication.

For this reason, we first presented our measurements without taking into account IPC, as this is a fairer comparison between the different programming models. The CUDA and OpenCL implementations assume that the host will be connected directly to the accelerators, thus they do not need to implement any kind of external communication. However, once we introduce a second machine the overhead will be similar, if not exactly the same, for any model used, as coordination between the host and device across a distributed system will be a requirement.

By implementing the HHAL in a client-server fashion we followed the approach used when designing the initial HN library, which allows the HHAL server to be easily distributed and run on a separate cluster. Once the platform matures, however, the improvements mentioned previously will be very important for the cases when the MANGO host and the HHAL do run on a single machine. This topic is discussed further in section 5.2.5.

Improvements can also be made on the HHAL side, without taking into account IPC. First, the overheads experienced for kernel executions can be attributed to the CUDA device context assignment which is currently repeated for each kernel launch. This is necessary, as we want MANGO to be able to run a kernel on any of the available GPUs. However, as the CUDA device context is set on a per-thread basis, this overhead could be avoided. In principle it would be possible to create a separate thread for each available device and assign a fixed context to each one as soon as the HHAL server is up and running.

Second, buffer transfers could be made asynchronous. By making the HHAL server multithreaded and enqueueing buffer transfers in different CUDA streams a bigger portion of the available PCI-E bandwidth could be exploited, as well as allowing for the ability to overlap data transfers and kernel executions as seen in [83].

Model	*Kernel (LOC)*	*Kernel (RD)*	*Host (LOC)*	*Host (RD)*	*Total (LOC)*	*Total (RD)*
MANGO	84	0%	190	0%	274	0%
CUDA	84	0%	166	-13%	250	-9%
OpenCL	84	0%	212	12%	296	8%

Table 4.2: *Lines of code for HotSpot benchmark*

Model	*Kernel (LOC)*	*Kernel (RD)*	*Host (LOC)*	*Host (RD)*	*Total (LOC)*	*Total (RD)*
MANGO	59	0%	133	0%	192	0%
CUDA	59	0%	108	-19%	167	-13%
OpenCL	59	0%	188	41%	247	29%

Table 4.3: *Lines of code for PathFinder benchmark*

4.2.4 Programmability

The results for the HotSpot benchmark can be seen in table 4.2 while the ones for PathFinder are in table 4.3.

When comparing kernel LOC we can see that they are equal across all implementations of the same benchmark. As MANGO uses the same kernel code as the CUDA implementation, this is expected. In the case of OpenCL, its kernels are also optimized for GPU execution and thus follow the same access pattern as a CUDA kernel. The only differences among the kernels being CUDA and OpenCL specific keywords and functions, which have a 1-to-1 mapping between each other.

On the host side, the CUDA implementation gains the upper hand on MANGO, needing 9% less code for HotSpot and 13% less code for PathFinder. On the other hand OpenCL requires 8% and 29% more code for each respective benchmark. These differences are mostly related to the following factors:

Kernel function calls: the single source nature of CUDA code allows these implementations to call a kernel like any other function. Meanwhile MANGO requires creating multiple argument objects, one for each of the kernel inputs and grouping them into a `KernelArguments` object to be able to start the kernel execution. OpenCL faces a similar issue as it also needs explicit `setKernelArg` calls for each kernel input. This gives a big advantage to CUDA, with MANGO and OpenCL being very similar to each other.

Kernel loading: both MANGO and OpenCL need to load kernel code from an external file. For MANGO this only requires creating a `KernelFunction` object, loading a file (single call to the `load` function of a `KernelFunction`) and registering the kernel to the `Context`. Meanwhile OpenCL requests that the kernel code is provided as a C string, leaving the file loading code to the user, who also needs to make

function calls to build the program and create a kernel object. The CUDA implementation is not required to do any of the previous work, as the kernel code can be called directly and is compiled along with the host code.

Resource deallocations: while all programming models need to explicitly request resource allocation for input and output buffers, MANGO does not leave the user with the responsibility to perform the inverse operations, at least not individually. When the MANGO implementations finish, they can simply call the context deallocation function and all requested resources will be correctly released. On the other hand, both OpenCL and CUDA require explicit release of resources. In the case of CUDA, this only entails calling `cudaFree` for each `cudaMalloc` call, for OpenCL it is much more involved, however. The programmer needs to release, on top of the allocated buffers, the following resources: `CommandQueue`, `Kernel`, `Program`, `Context`, a buffer for the program source string, a buffer for querying device ids and a buffer for querying platform ids.

Device selection: OpenCL requires manual querying of the available platforms on the host as well as the available devices in order to select the device to use for the benchmark. Both are multi-step processes requiring multiple function calls and host heap memory allocations. As CUDA only runs on Nvidia GPUs, one only needs to select a particular device ID among the ones available, but that can also be skipped (and was, for our tests) if there is only a single GPU available or there is no preference in which GPU to run the kernel. Finally, MANGO does all this work automatically, needing no input from the user apart from providing the device kernels along with information about which platform they target (in this case Nvidia). In the background, BarbequeRTRM and the HHAL will make sure to assign the kernel to run on the most suitable device. This approach is significantly more optimal in terms of resource management, at no extra code cost for the MANGO user. Neither CUDA nor OpenCL provide similar functionalities, so an equivalent implementation would require considerably more programmer effort.

Error handling: as both CUDA and OpenCL work with exit codes in order to communicate runtime errors, there is a need to perform explicit error handling at each function call in their benchmark implementations. For brevity, most of this error handling was extracted into preprocessor macros, so they do not contribute significantly to the total LOC. Some OpenCL calls however cannot be easily handled by a macro and need extra error handling code. Meanwhile, the last layer of the MANGO API maps runtime errors into exceptions so they can be handled more easily by the user. Note that the error handling code added for both CUDA and OpenCL follows a "fail-fast"

Model	*Kernel call*	*Kernel load*	*Deallocations*	*Device select*	*Error handling*	*Total*
MANGO	20	3	1	0	0	24
CUDA	7	0	3	0	7	17
OpenCL	16	16	10	14	14	70

Table 4.4: *Size of relevant sections of code in HotSpot benchmark*

Model	*Kernel call*	*Kernel load*	*Deallocations*	*Device select*	*Error handling*	*Total*
MANGO	15	3	1	0	0	19
CUDA	6	0	3	0	7	16
OpenCL	11	16	10	14	14	65

Table 4.5: *Size of relevant sections of code in PathFinder benchmark*

approach, essentially converting an error code coming from the library into a program crash, similar to throwing an exception, like MANGO does.

A breakdown of LOC differences for these particular sections of the code in both benchmarks can be seen in tables 4.4 and 4.5 which refer to HotSpot and PathFinder respectively. As shown here, the main disadvantage of MANGO over CUDA is due to the kernel call code. This could be improved by leveraging C++'s templates, allowing for a variety of argument types and quantities to be supplied directly to MANGO's `kernel_launch` call. This will turn the implementation of the function significantly more complex, but will provide a much more concise API for the users of the platform.

CHAPTER 5

Conclusions and Future Work

In this Chapter we provide a final overview of our work, as well as some future challenges for MANGO according to our vision of its development.

Section 5.1 contains our conclusions about the work on this thesis, covering our entire journey in chronological order from the initial contribution of the Python wrapper to the final experimental campaign covered in this document.

Section 5.2 is destined to the next developers and researchers working on the system. Here we hope to communicate our vision of the future of MANGO and provide a starting point for improvements.

5.1 Conclusions

In this work we explored our contributions to the MANGO software stack, extending the capabilities of the platform and restructuring its design to improve its future extensibility. This covered a wide range of modifications across most of the MANGOLIBS codebase [84], and the addition of multiple new libraries to the system.

Our work started with the implementation of a Python wrapper, aiming to provide users with a simpler API in a language that is widely used for data science and machine learning applications. As these types of applications are one of the main drivers for HPC development, a Python API allows our programming model to be more easily accessible and usable for new users.

After that, the addition of Just-In-Time (JIT) compilation allowed users to load kernels from source code without the need of pre-compilation. This not only improves the feedback loop by decreasing the time to deploy a new kernel modification, but will also allow, in the future, for optimizations over the kernel code which need runtime device capability and availability information.

Our main contribution involved the extension of the MANGO supported platforms, expanding the system with the capability of exploiting GPU accelerators for general purpose applications. This was achieved by the implementation of the Nvidia Architecture Node (NAN), which leverages the CUDA Driver API to execute kernels in any of the connected devices.

During the integration of the NAN with the existing MANGO stack, there was a significant hurdle when adapting the Libmango code to support this new platform. Hurdle that was a clear consequence of the dependence between Libmango and the underlying architectures running the kernels. Even though the initial abstraction layer of the HN library and daemon allowed for abstraction over the custom MANGO hardware, it was not easily extendable to other devices, such as a GPUs.

This finally concluded in our implementation of the Heterogeneous Hardware Abstraction Library (HHAL), which also involved an internal restructuring of the Libmango library. With the new HHAL library, we aimed to allow for simple extension of the software stack to support new devices, while also maximizing the performance that can be exploited from each individual architecture. By keeping the hardware abstractions at the appropriate level, platform specific improvements can be made without affecting the rest of the interconnected libraries, while still achieving greater performance benefits. Furthermore, the client-server approach used when designing the HHAL will allow for a straightforward adaptation of MANGO into a distributed system, spanning multiple clusters.

Closing out our thesis, we moved forward with an experimental campaign to evaluate the new capabilities added by the NAN and the HHAL libraries, as well as how the MANGO project as a whole compares with other, more established, programming models in terms of performance and programmability. Even though maximizing per-

formance was not our main priority for this initial implementation, the results of our experiments show a promising outline for the future of the project.

The performance overheads found in MANGO when compared to other programming models were not negligible, but also easily identifiable and, in principle, solvable thanks to the final state of the software stack. To facilitate this, and also to further improve the ease of use and programmability of the system, we include our suggestions and vision for future work in the following section.

5.2 Future Work

During our work on the project, we thought of multiple ideas to improve the platform, but were not able to cover ourselves in order to keep a reasonable scope for this thesis. Here we detail these ideas, in hope that they are considered by up and coming developers, looking to delve into the MANGO software stack and continue growing the platform.

5.2.1 Task Graph Automatic Execution

In the search of making MANGO's usage as seamless and easy for developers as possible, a great addition to the framework would be the automatic execution of task graphs.

By automatic execution, we refer to the execution of the entirety of an application's task graph from a single start signal, leaving to MANGO the responsibility of handling buffer data transference and kernel execution from their dependencies.

All the necessary information to make this happen is already being provided by users of Libmango through the task graph definition. As a result, the only Libmango API change this might entail is the addition of a start task graph call.

This functionality can be fully implemented in the Libmango module, which based on its task graph knowledge, is capable of keeping track of kernel dependencies and perform buffer read/write and kernel start calls to the HHAL module as required.

5.2.2 GPU Kernel Parallelism Optimization

When it comes to optimization, there is an endless number of approaches to investigate with the goal of achieving a higher level of system efficiency, as seen in the multiple resource allocation policies that are already implemented in the BarbequeRTRM module.

Focusing on the GPU landscape, optimizing the use of their parallelization capabilities is the key to achieve better results. In 2019, a prototype for workload-aware efficient multiprocessing for modern GPGPUS: Slate [85], was presented. In this work, the authors introduce the idea of modifying kernels source code to allow for control of their parallelization extent at all times, while also maintaining their correct functionality. This way, the GPU resources (i.e. number of cores) an executing kernel is using can be modified "on the go", allowing the reconfiguration of allocated GPU resources to either reduce or increase the number of running tasks at any certain time. This job is carried out by a system-aware resource manager akin to MANGO's BarbequeRTRM.

Since BarbequeRTRM does not currently support reconfiguration of allocated resources once an application is running, there is clearly a great amount of work to be done on this front before a Slate-type optimization could be implemented.

5.2.3 Native CPU Support

In the current implementation, the GN module is used to run kernels on the user's host system CPU. The GN module was developed as an HN emulator and therefore its execution model is not optimized for efficiently running kernels on CPU devices, as its API is limited to that of HN and mimics its behavior.

To properly support CPUs as target accelerators, a new architecture node must be introduced, along with an HHAL manager that interacts with it.

5.2.4 HNlib Integration

The restructuring of the MANGO software stack, centered around the introduction of the Heterogeneous Hardware Abstraction Layer, forced a rework of the Libmango module. This entailed the removal of the Libmango-HNlib interaction, hence effectively eliminating the HN accelerators from the pool of MANGO supported architectures. With accelerator communication now taking place through the HHAL module, an HN Manager should be added to HHAL in order to reestablish it. As we did not have physical access to the custom MANGO hardware, designed to work with the HN library, no integration effort was carried out.

5.2.5 HHAL "hybrid" mode communication

As seen in our experimental results in chapter 4, the performance of MANGO is hurt by the client-server approach of the HHAL. Although this aspect is key for a distributed

system, it can be improved when dealing with a system that runs the entire MANGO stack on a single cluster.

Our proposed "hybrid" mode introduces a new type of communication between Libmango, the HHAL and BarbequeRTRM, provided that they all run on the same machine. For necessary bookkeeping between BarbequeRTRM and other modules, the solution would switch from pure socket connections to a mix of shared memory and socket usage. While all types of IPC would be bypassed when dealing with kernel executions and buffer transfers, by directly linking HHAL to Libmango.

This solution would provide a significant performance improvement for memory-bound applications in cases where socket communication is not required.

5.2.6 Multithreaded CUDA Management

In order to improve the performance of MANGO when running CUDA applications, it is important to implement multithreading in the Nvidia Architecture Node library running on the HHAL server. As mentioned in chapter 4, it would be possible to avoid the overhead of switching device contexts between each kernel launch by creating a separate thread for each CUDA device.

In addition, if the CUDA stream API is used, it would allow for the asynchronous transfer of data to and from device buffers, consequently enabling the overlap of data transfers and kernel executions.

By carrying out this implementation on the HHAL module, the changes would be completely transparent to Libmango, and by extension, the user.

Bibliography

[1] G. Massari, G. Agosta, A. Pupykina, F. Reghenzani, M. Zanella, W. Fornaciari, J. Flich, R. Tornero, J. M. Martinez, M. Zapater, I. P. Fernandez, D. Atienza, A. Cilardo, M. Gagliardi, A. Dray, and G. Guillaume, "Mango - exploring manycore architectures for next-generation hpc systems - wp4," 02 2019.

[2] G. Massari, G. Agosta, A. Pupykina, F. Reghenzani, M. Zanella, W. Fornaciari, J. Flich, R. Tornero, J. M. Martinez, M. Zapater, I. P. Fernandez, D. Atienza, A. Cilardo, M. Gagliardi, A. Dray, and G. Guillaume, "Exploring manycore architectures for next-generation hpc systems through the mango approach," *Microprocessors and Microsystems (2018)*, p. 6, 05 2018.

[3] Mark Harris, "A Brief History of GPGPU," May 2015.

[4] Nick England, "Ikonas Graphics Systems - The world's first GPGPU." `http://www.graphics-history.org/ikonas/`, Accessed: Apr 2021.

[5] T. Van Hook, "Real-time shaded nc milling display," *SIGGRAPH Comput. Graph.*, vol. 20, p. 15–20, Aug. 1986.

[6] G. Kedem and Y. Ishihara, "Brute force attack on UNIX passwords with SIMD computer," in *8th USENIX Security Symposium (USENIX Security 99)*, (Washington, D.C.), USENIX Association, Aug. 1999.

[7] K. E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized voronoi diagrams using graphics hardware," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, (USA), p. 277–286, ACM Press/Addison-Wesley Publishing Co., 1999.

[8] "OpenGL." `https://www.opengl.org/`, Accessed: Apr 2021.

[9] "OpenGL Wiki: Fixed Function Pipeline." `https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline`, Accessed: Apr 2021.

[10] "NVIDIA nfiniteFX Engine: Programmable Pixel Shaders," 2001.

[11] "NVIDIA nfiniteFX Engine: Programmable Vertex Shaders," 2001.

[12] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, (New York, NY, USA), p. 55, Association for Computing Machinery, 2001.

[13] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, (Goslar, DEU), p. 109–118, Eurographics Association, 2002.

[14] "TechPowerUp: ATI Radeon 9700 PRO." https://www.techpowerup.com/gpu-specs/radeon-9700-pro.c50, Accessed: Apr 2021.

[15] "TechPowerUp: NVIDIA GeForceFX." https://www.techpowerup.com/gpu-specs/radeon-9700-pro.c50, Accessed: Apr 2021.

[16] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, (Goslar, DEU), p. 41–50, Eurographics Association, 2003.

[17] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, (New York, NY, USA), p. 896–907, Association for Computing Machinery, 2003.

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, p. 777–786, Aug. 2004.

[19] "Xbox 360 Technical Specifications (Wayback Machine Archive Aug 2008." https://web.archive.org/web/20080822024003/http://www.xbox.com/en-AU/support/xbox360/manuals/xbox360specs.htm, Accessed: Apr 2021.

[20] "NVIDIA GeForce 8800 GPU Architecture Overview," Nov 2006. https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf, Accessed: Apr 2021.

[21] "AMD Compute Abstraction Layer (CAL) Programming," Dec 2010. https://developer.amd.com/wordpress/media/2012/10/AMD_CAL_Programming_Guide_v2.0.pdf, Accessed: Apr 2021.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[23] "CUDA Toolkit Archive." https://developer.nvidia.com/cuda-toolkit-archive, Accessed: Apr 2021.

[24] "Tom's Hardware: AMD Ditches Close-To-Metal, Focuses On DX11 And OpenCL," Aug 2008. https://www.tomshardware.com/news/AMD-stream-processor-GPGPU,6072.html, Accessed: Apr 2021.

[25] "The OpenCL Specification v3.0.6," Dec 2020. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[26] "The OpenCL C Specification v3.0.6," Dec 2020. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf.

[27] "ISO/IEC 9899:1999 - Programming languages - C," Dec 1999.

[28] "ISO/IEC 9899:2011 - Information technology - Programming languages - C," Dec 2011.

[29] S. Pennycook, S. Hammond, S. Wright, J. Herdman, I. Miller, and S. Jarvis, "An investigation of the performance portability of opencl," *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1439–1450, 2013. Novel architectures for high-performance computing.

[30] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 1–13, 2019.

[31] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, and S. Parker, "Performance portability evaluation of opencl benchmarks across intel and nvidia platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 330–339, 2020.

[32] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Optimizing opencl code for performance on fpga: k-means case study with integer data sets," *IEEE Access*, vol. 8, pp. 152286–152304, 2020.

[33] M. Nourian, M. E. Zarch, and M. Becchi, "Optimizing complex opencl code for fpga: A case study on finite automata traversal," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 518–527, 2020.

[34] "SPIR Overview." `https://www.khronos.org/spir/`, Accessed: Apr 2021.

[35] "Vulkan." `https://www.khronos.org/vulkan/`, Accessed: Apr 2021.

[36] "SPIR-V Specification 1.5 Revision 5," Jan 2020. `https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html`.

[37] "OpenCL: API Adopters." `https://www.khronos.org/conformance/adopters/conformant-companies`, Accessed: Apr 2021.

[38] R. Nozal, J. Bosque, and R. Beivide, "Enginecl: Usability and performance in heterogeneous computing," *Future Generation Computer Systems*, vol. 107, 02 2020.

[39] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), p. 273–283, Association for Computing Machinery, 2014.

[40] "AMD Launches 'Boltzmann Initiative' to Dramatically Reduce Barriers to GPU Computing on AMD FirePro™ Graphics," Nov 2015. `https://www.amd.com/en/press-releases/boltzmann-initiative-2015nov16`, Accessed: May 2021.

[41] N. Otterness and J. H. Anderson, "AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)* (M. Völp, ed.), vol. 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 10:1–10:23, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[42] "SC16: HIP and CAFFE Porting and Profiling with AMD's ROCm," 2016. `https://www.youtube.com/watch?v=I7AfQ730Zwc`, Accessed: May 2021.

[43] Y. Tsai, T. Cojean, T. Ribizel, and H. Anzt, "Preparing ginkgo for amd gpus – a testimonial on porting cuda code to hip," 06 2020.

[44] "OpenMP API Specification 5.1," Nov 2020. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`.

[45] "OpenACC API Specification 3.1," Nov 2020. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf`.

[46] "OpenMP Accelerator Support for GPUs." `https://www.openmp.org/updates/openmp-accelerator-support-gpus/`, Accessed: Apr 2021.

[47] "OpenACC API Specification 1.0," Nov 2011. `https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf`.

[48] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller, "A pattern-based comparison of openacc and openmp for accelerator computing," in *Euro-Par 2014 Parallel Processing* (F. Silva, I. Dutra, and V. Santos Costa, eds.), (Cham), pp. 812–823, Springer International Publishing, 2014.

[49] M. Khalilov and A. Timoveev, "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA v100 GPU," *Journal of Physics: Conference Series*, vol. 1740, p. 012056, jan 2021.

[50] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-gpus using openmp and openacc-based hybrid model," *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, 05 2013.

[51] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, and C. Silvano, "Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes," *The Journal of Supercomputing*, vol. 75, pp. 3374–3396, Jul 2019.

[52] "SYCL 2020 Specification (revision 3)," Mar 2021. `https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf`, Accessed: Apr 2021.

[53] "SYCL 2020 - What you need to know," 2020. `https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know`, Accessed: May 2021.

[54] "ISO/IEC 14882:2017 Programming languages — C++," Dec 2017.

[55] "ISO/IEC 14882:2020 Programming languages — C++," Dec 2020.

[56] I. Z. Reguly, "Performance portability of multi-material kernels," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 26–35, 2019.

[57] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of hpc-style sycl applications," in *IWOCL '20*, (United States), Association for Computing Machinery (ACM), Apr. 2020. International Workshop on OpenCL, IWOCL ; Conference date: 27-04-2020 Through 29-04-2020.

[58] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-level c++ for accelerator clusters," in *Euro-Par 2019: Parallel Processing* (R. Yahyapour, ed.), (Cham), pp. 291–303, Springer International Publishing, 2019.

[59] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems* (K. M. Decker and R. M. Rehmann, eds.), (Basel), pp. 213–218, Birkhäuser Basel, 1994.

[60] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[61] H. Kaiser, P. Diehl, A. Lemoine, B. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhanan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "Hpx - the c++ standard library for parallelism and concurrency," *Journal of Open Source Software*, vol. 5, p. 2352, 09 2020.

[62] D. Airlie, "But Mummy I don't want to use CUDA - Open source GPU compute," Jan 2019. `https://www.youtube.com/watch?v=ZTq8wKnVUZ8`, Accessed: Apr 2021.

[63] J. Lebar, "CppCon 2016: "Bringing Clang and C++ to GPUs: An Open-Source, CUDA-Compatible GPU C++ Compiler"," 2016. `https://www.youtube.com/watch?v=KHa-OSrZPGo`, Accessed: Apr 2021.

[64] "Wikipedia: Compute Kernel." `https://en.wikipedia.org/wiki/Compute_kernel`, Accessed: Apr 2021.

[65] "The RedMonk Programming Language Rankings: January 2021." `https://redmonk.com/sogrady/2021/03/01/language-rankings-1-21/`, Accessed: May 2021.

[66] "Cython C-Extensions for Python." `https://cython.org`, Accessed: May 2021.

[67] "The Barbeque Run-Time Resource Manager Open-Source Project (BOSP)." https://bosp.deib.polimi.it/, Accessed: May 2021.

[68] P. Bellasi, G. Massari, and W. Fornaciari, "Effective runtime resource management using linux control groups with the barbequertrm framework," 03 2015.

[69] P. Bellasi, G. Massari, and W. Fornaciari, "A rtrm proposal for multi/many-core platforms and reconfigurable applications," 2012.

[70] "NVIDIA Management Library (NVML)." https://developer.nvidia.com/nvidia-management-library-nvml, Accessed: May 2021.

[71] "Extensible Markup Language (XML)." https://www.w3.org/XML/, Accessed: May 2021.

[72] "Clang: a C language family frontend for LLVM." https://clang.llvm.org, Accessed: Apr 2021.

[73] "CUDA Toolkit Documentation - CUDA Driver API." https://docs.nvidia.com/cuda/cuda-driver-api/index.html, Accessed: May 2021.

[74] "CUDA Toolkit Documentation - Parallel Thread Execution ISA." https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, Accessed: May 2021.

[75] "CUDA Toolkit Documentation - NVRTC." https://docs.nvidia.com/cuda/nvrtc/index.html, Accessed: May 2021.

[76] G. Massari, G. Agosta, A. Pupykina, F. Reghenzani, M. Zanella, W. Fornaciari, J. Flich, R. Tornero, J. M. Martinez, M. Zapater, I. P. Fernandez, D. Atienza, A. Cilardo, M. Gagliardi, A. Dray, and G. Guillaume, "Mango - exploring manycore architectures for next-generation hpc systems - wp2," 04 2018.

[77] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[78] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotspot: a compact thermal modeling methodology for early-stage vlsi design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 501–513, 2006.

[79] "Parallel Thread Execution ISA Version 7.3." https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[80] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *High Performance Computing* (M. Taufer, B. Mohr, and J. M. Kunkel, eds.), (Cham), pp. 489–507, Springer International Publishing, 2016.

[81] J. D. McCalpin, "Sustainable Memory Bandwidth in Current High Performance Computers," Oct 1995. http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html, Accessed: May 2021.

[82] M. Harris, "How to Implement Performance Metrics in CUDA C/C++," Nov 2012. https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/, Accessed: May 2021.

[83] M. Harris, "How to Overlap Data Transfers in CUDA C/C++," Dec 2012. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/, Accessed: June 2021.

[84] "Mangolibs bitbucket repository." https://bitbucket.org/mango_developers/mangolibs.

[85] T. Allen, X. Feng, and R. Ge, "Slate: Enabling workload-aware efficient multiprocessing for modern gpgpus," 2019.