POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING

TITLE

Author:
Manuel Pedrozo
Student ID:
Numbers

Author:
Tomás Perez Molina
Student ID:
Other numbers

Supervisor:
PhD. Giovanni Agosta

Co-supervisor:
PhD. Giuseppe Massari

A.Y. 2020/2021

Contents

List of Figures

List of Tables

Acknowledgements

We would like to thank... pipo.

Abstract

ABSTRACT goes here.

Sommario

THE Italian sommario goes here.

CHAPTER 1

Introduction

1.1 Mangolibs

1.1.1 libmango

1.1.2 BarbequeRTRM

1.1.3 HN library

1.1.4 GN Emulation

1.2 Thesis Contributions

1.2.1 HHAL

1.2.2 GPU Support

1.2.3 JIT Compilation

1.2.4 Python Wrapper

1.3 Document Structure

CHAPTER 2

State of the Art

Text goes here.

Focus on multiprocessing models which focus on performance improvements.

- Single source kernel: OpenCL, HIP.
- Language extensions for high level parallelism: OpenMP, OpenACC, SYCL, C++ AMP.
- Proprietary solutions: CUDA.

Other types of models which focus on scalability, but not covered [1]:

- Actor model
- MPI

Section 2.1 covers the history of advancements in GPGPU technology and is based on [2], a talk by NVIDIA Software Engineer Mark Harris.

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.

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 [3]. Van Hook followed this up the next year with a paper on solid modelling with the Ikonas [4].

In August 1999, Kedem et al. [5] 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.

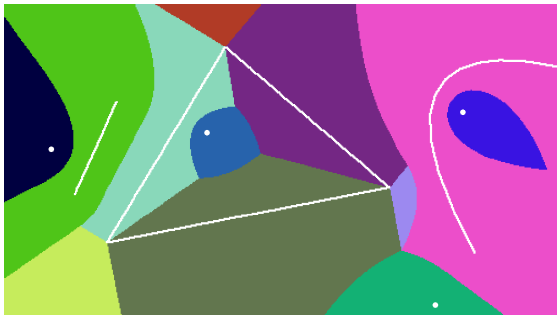


Figure 2.1: *Generalized Voronoi diagram computed interactively on PC (Credit: Hoff et al.)*

Also in 1999, Hoff et al. [6] managed to perform computations of generalized Voronoi diagrams using graphics accelerators, such as the NVIDIA TNT2, connected to a PC [2], as opposed to the specialized hardware in the PixelFlow. This was achieved by using the OpenGL API [7]. 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 [8].

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 [9, 10]. 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 [11]. 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.

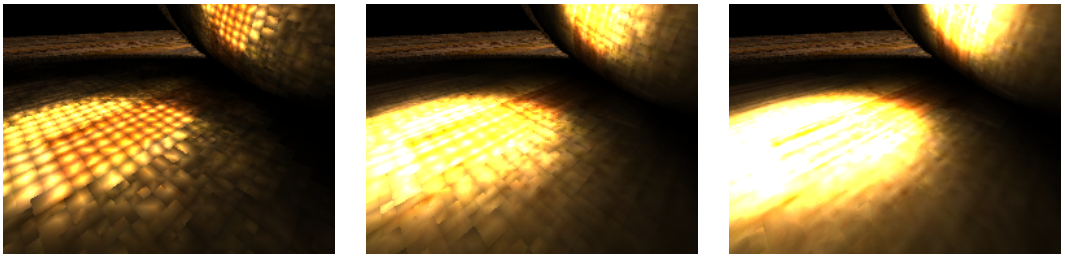
Approximate simulations of natural phenomena were achieved on the GPU by Harris et al. [12]. This included interactive visualizations of convection, reaction-diffusion and boiling. As the end effect of the simulation was to display visuals, this application had the advantage that data did not need to 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" [2]. 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 [13]. NVIDIA followed up with their own DX9 GPU, the GeForceFX, in January 2003 [14].

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 [15], 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 [16]. 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

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>;
    // 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
}
```

programs. In August 2004 researchers at Stanford University presented Brook [17], 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 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 [18]. 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.

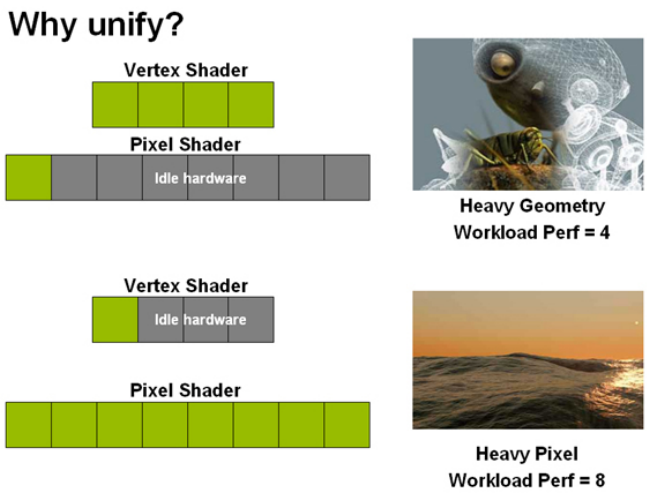


Figure 2.3: *Fixed shader model performance characteristics (Credit: NVIDIA)*

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 though a unified architecture was not a requirement to use DirectX 10, it provided better efficiency, load-balancing and power utilization [19]. 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).

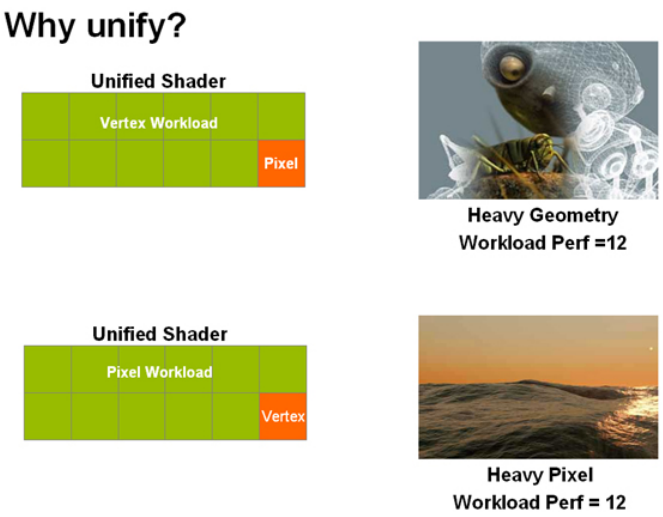


Figure 2.4: *Unified shader performance characteristics (Credit: NVIDIA)*

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 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 [20]. For higher level programming, AMD also supported compilation of Brook programs directly to the hardware [21].

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 [22]. 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 features. As opposed to AMD CTM, which combined the CAL C API with a low level

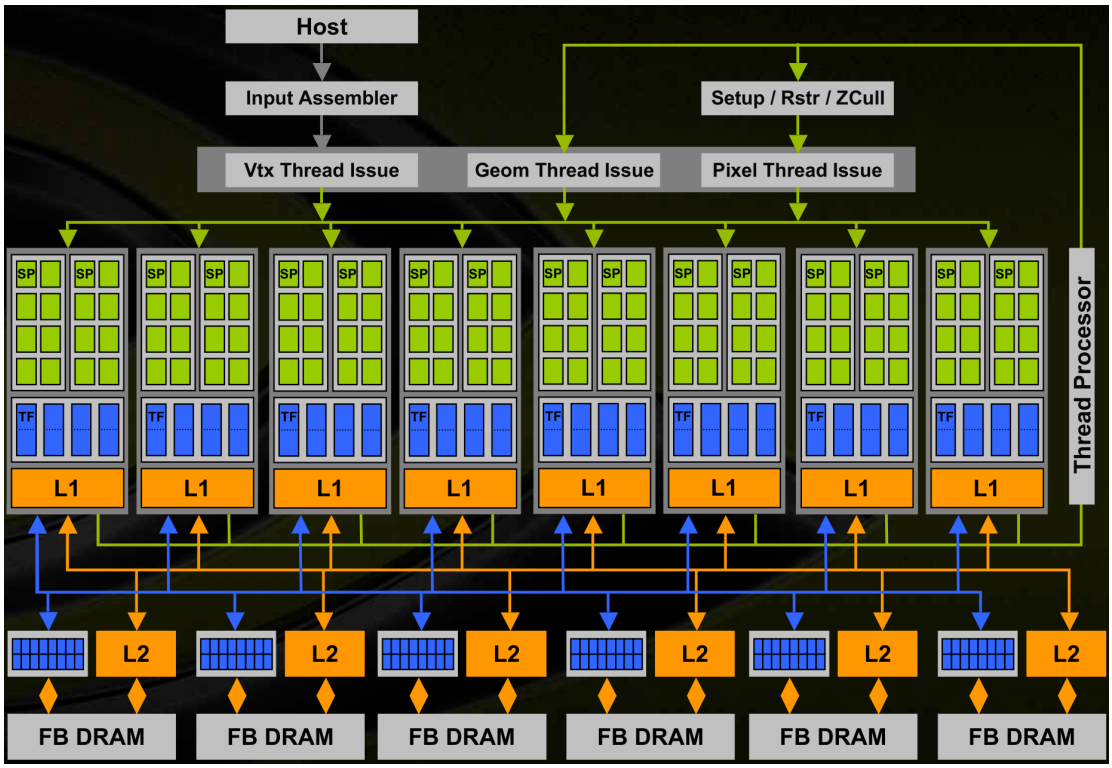


Figure 2.5: *GeForce 8800 GTX Block Diagram (Credit: NVIDIA)*

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. 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

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;

  // 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);
}
```

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 [21].

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 [23], detailed in the following section.

2.2 OpenCL

Released in December 2008, Open Compute Language or OpenCL [24] 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 [25] 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) [26], with the addition of some *ISO/IEC 9899:2011 - Information technology - Programming languages - C* specification (also referred to as C11) [27] 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 per-programming toolkit basis [28]. 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

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];
}
```

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 [28–30]. For maximum performance, some tweaking of the source code may still be necessary depending on which device is being targeted [31,32].

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 [33]. While SPIR-V is also supported by the Vulkan [34] graphics API, it uses a different execution mode of the language (**GLCompute** versus **Kernel** for OpenCL), so compute codes are not interchangeable [35]. 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 [24]. 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 [36]. 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.

2.2.1 EngineCL

EngineCL [37], introduces a new object-oriented API on top of OpenCL. The EngineCL class provides a higher level view of the OpenCL context and management

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];

  // Initialize a, X, Y ...

  cl_uint num_devices;
  cl_device_id *device_list;
  // Get platform and device information ...
  // 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;
  // 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);

  // Release all OpenCL allocated objects and host buffers...
}
```

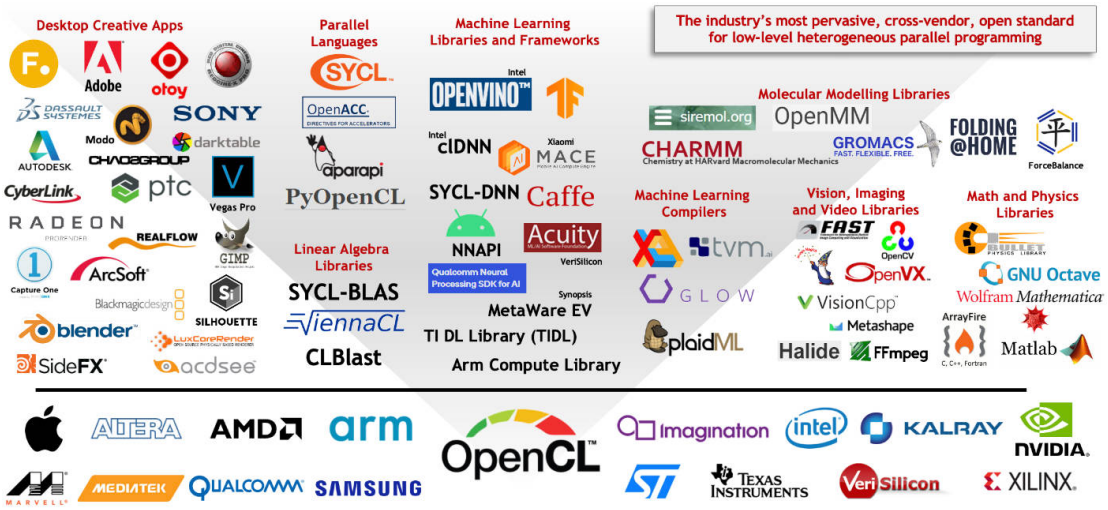


Figure 2.6: *OpenCL accelerated applications (Credit: Khronos Group)*

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 [38] 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 [39]. 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 [40]. 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. 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 modify all CUDA function calls and keywords to their OpenCL counter parts.

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 [41]. As shown in figure 2.7, the HIP port of CAFFE only required changing 907 lines of code (LOC), of which 688 or 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 a more than 30,000 LOC.

[42] 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

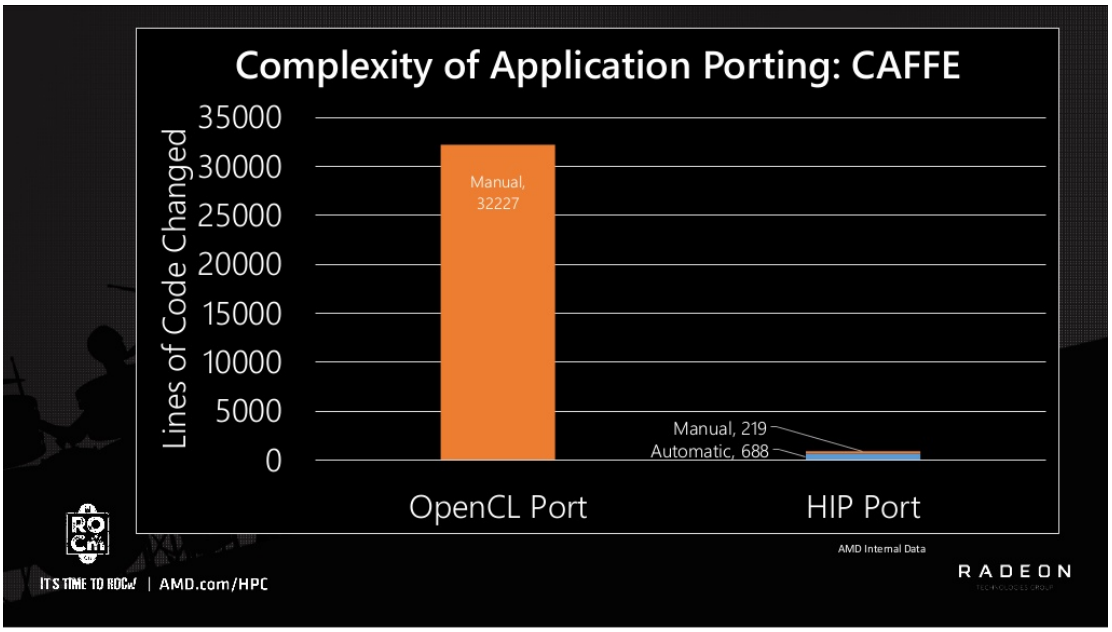


Figure 2.7: *Complexity of porting CAFFE, OpenCL vs HIP (Credit: AMD)*

CUDA and maintain two versions at the same time. Still, 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% of performance difference.

2.4 OpenMP & OpenACC

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

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 [43, 44].

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

Listing 2.5: *OpenMP saxpy example*

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

    // 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];
    }
}
```

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 [45]. Meanwhile, OpenACC focused on heterogeneous computing and accelerator offloading from the start [46], also treating the multicore CPU itself as a device.

[47] 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, [48] 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 [49] exploit parallelism on the CPU with OpenMP to schedule code to run on multiple GPUs. [50] 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 [51] 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 [52].

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++) [53] and also the newest C++20 standard (ISO/IEC 14882:2020 Programming languages — C++) [54]. 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 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

Listing 2.6: *SYCL saxpy example*

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

// 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();
```

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.

In [55] 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

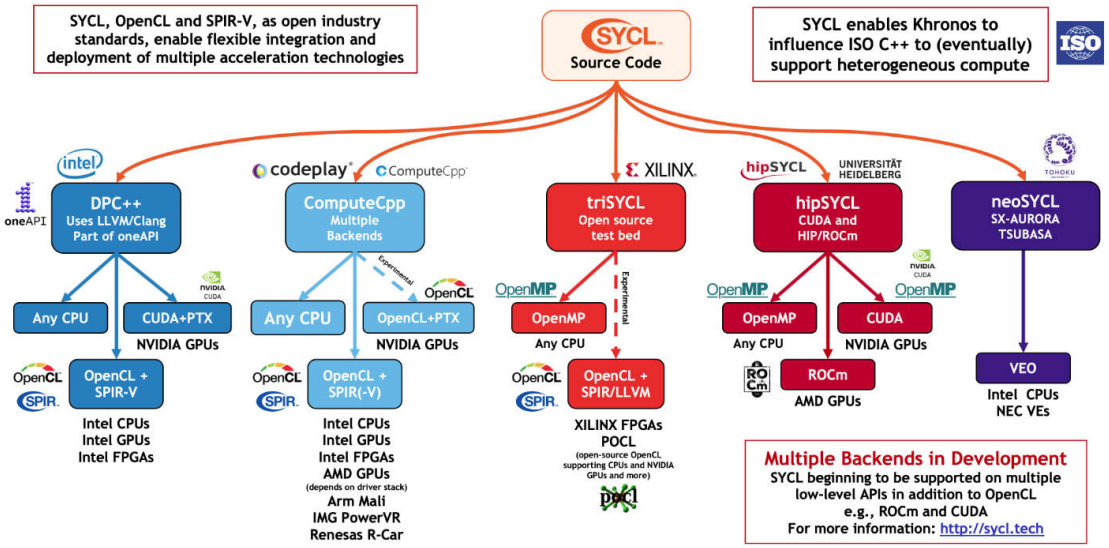


Figure 2.8: *SYCL implementations in development (Credit: Khronos Group)*

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.

[56] 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 their four neighbors.
- CloverLeaf: largest application in the group, measuring at about 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 requires the maturity and support of their implementations as well.

2.5.1 Celerity

Celerity [57] 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) [58] 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.

The execution model of Celerity 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, allowing the workers, during the *live-pass*, to perform peer-to-peer communication 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 [59] 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 [60], SYCL and pthreads.

In terms of performance, the initial implementation presented in [59] showed it was capable to achieve at least 90% of the performance of the architecture specific, optimized variants of the same benchmarks (i.e. comparing Kokkos against OpenMP for CPUs and CUDA for GPUs). [55] also evaluates the performance of Kokkos. As mentioned in the SYCL section, this paper uses two different multi-material problems as benchmarks. The performance of Kokkos is overall 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.

CHAPTER 3

Gap Analysis

Using a single programming model, even open source ones does not guarantee portability due to hardware vendor's unwillingness to work together [61].

In order to squeeze all the performance out of the hardware it is necessary to use a low-level language which takes into account the specific details of the hardware [62].

CHAPTER 4

Architecture Description

4.1 Overall description

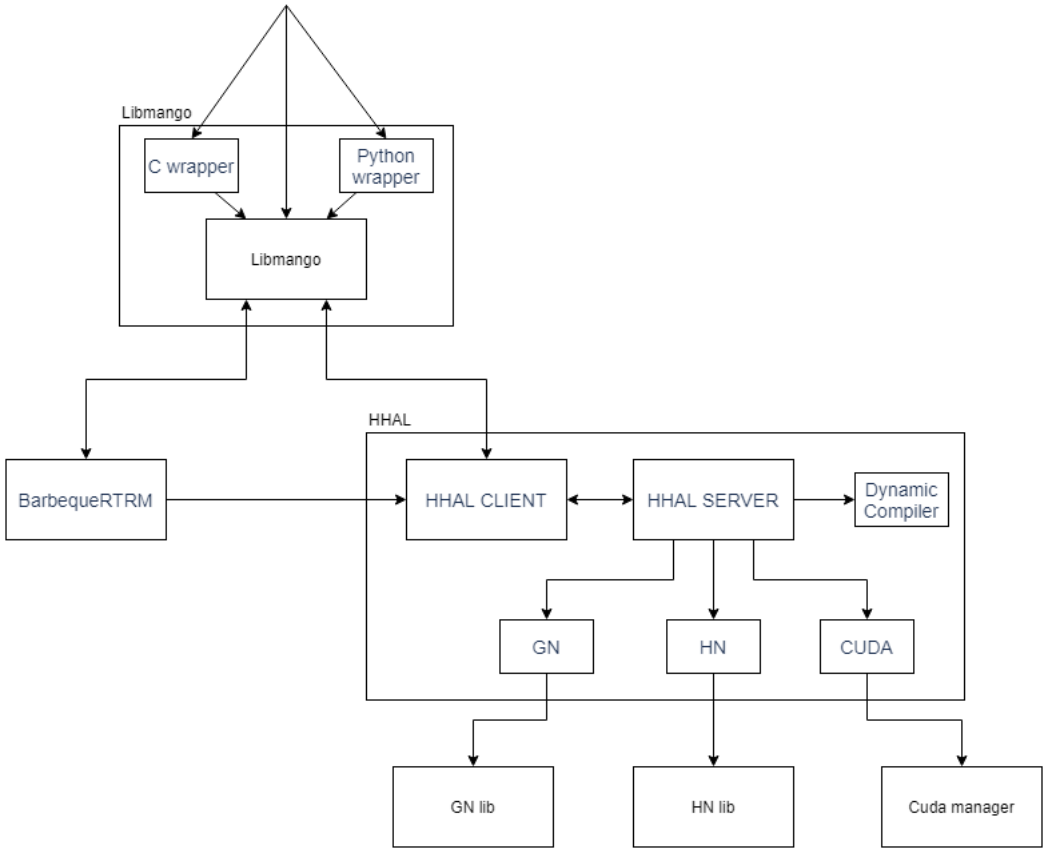
The MANGO project aims at allowing developers to easily develop applications that target different types of accelerator architectures. In particular, three types of accelerators are considered: symmetric multiprocessors, which are characterized by good capabilities in terms of OS support and execution 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 or support any kind of software runtime. Applications, on the other hand, may be developed either by domain experts with limited knowledge of parallel computing and programming models, or by more experienced programmers.

Thus, the following requirements arise:

- Supporting the use of industry-standard programming models for heterogeneous systems, such as OpenCL, while guaranteeing functional portability across different programmable accelerators, as well as host-side compatibility for all accelerators and automated resource management;
- Supporting a simple fork-join model, on which application developers not willing to use OpenCL can map their applications;

- Supporting future extensions of the MANGO software stack to support skeleton-based programming.

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 at least functional compatibility. Depending on the individual capabilities of each accelerator, the low-level runtime system should also introduce optimizations or additional features; this would indeed cause compatibility issues, but it would also allow developers to implement specialized versions of their applications for any given accelerator [63].



4.2 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.

4.2.1 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.

4.2.2 Memory Buffer

Figure 4.1: *Input and Output Buffers*

In computer science, a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another [65].

Kernels read from Input Buffers and write to Output Buffers for inter-kernel and host-kernel data transferring.

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.

4.2.3 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.

4.2.4 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. The Task Graph provides the resource manager with the information needed to generate the best feasible resource allocation for the requested Quality of Service (QoS) [63].

4.3 Libmango

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 BarbecueRTRM 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.

4.3.1 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 its created by the user at the

beginning of their interaction, with an application name and a recipe file needed by BarbecueRTRM for the resource allocation (further explained in the BarbecueRTRM section 4.4.3). 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 BarbecueRTRM for the resource allocation. After a successful resource allocation, the application is now ready to run.

4.3.2 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 - a kernel implementation targeting an Nvidia GPGPU 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 string stored in memory or a source file, to be dynamically compiled if required. The resource manager will rely on the available options in 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 essentially 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 and 32 bits, as well as long and float 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.

4.3.3 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.

4.3.4 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.

4.3.5 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 compatibility with C code 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 [63].

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 4.2, Python was the second most popular programming language in the RedMonk's Programming Language Rankings [66], only falling behind JavaScript.

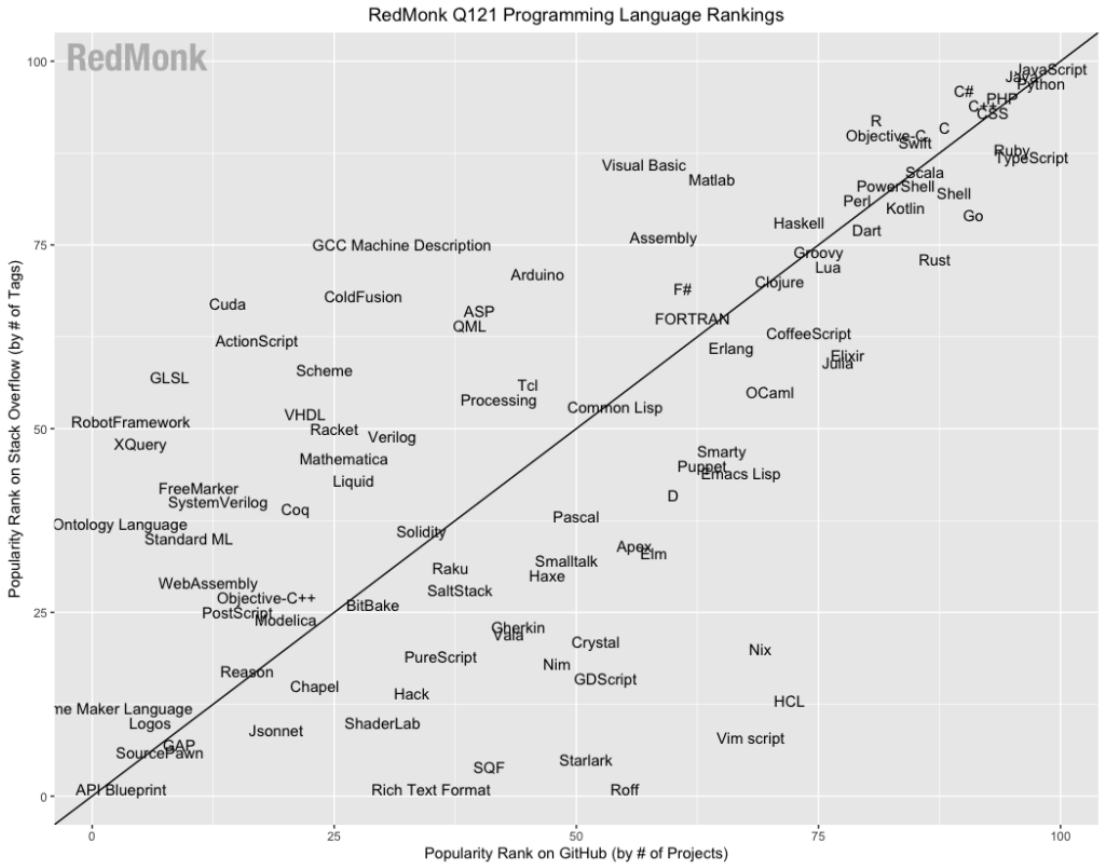


Figure 4.2: *RedMonk's January 2021 Programming Language Rankings*

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 [67]. For the API user, Cython usage is completely transparent and not a requirement, so their code can be purely written in Python.

Listing 4.1: *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
```

4.3.6 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 4.2: *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 4.3: *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 4.4: *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** 4.2 kernel computation, we will use it to check the obtained results.

Listing 4.5: *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 BarbecueRTRM 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 4.6: *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 4.7: *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 ouput 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 4.8: *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 respective 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 4.9: *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 4.10: *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 4.11: *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 4.12: *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.

4.4 BarbecueRTRM

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 [63].

The framework used for resource management in the MANGO system is the Barbeque Run-Time Resource Manager (BarbecueRTRM). As described in its official documentation [68], 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 [69] [70].

4.4.1 Resource Manager Architecture

In figure 4.3, we can see an overview of the BarbecueRTRM 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 BarbecueRTRM Daemon running in the host system.

The Programming Model Synchronization Layer in the RTLib facilitates MANGO integration with the BarbecueRTRM by working as an abstraction layer that supports the MANGO programming model for a BarbecueRTRM managed application. The Libmango-BarbecueRTRM 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

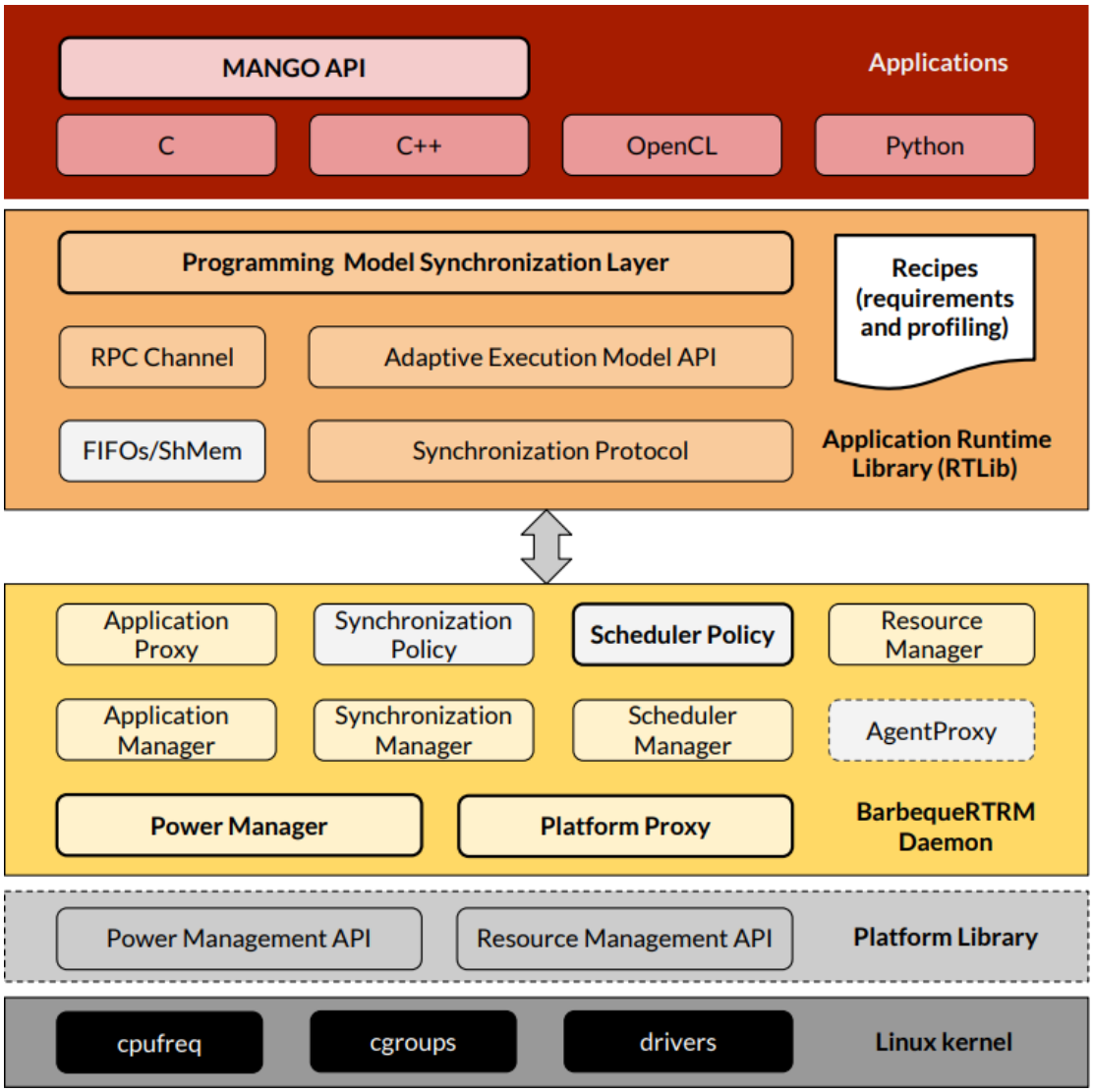


Figure 4.3: *Overview of the BarbequeRTRM*

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) [71] is utilized for controlling and obtaining runtime data of NVIDIA GPUs [63].

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

- scheduler policy

4.4.2 Integration

As previously stated, The Libmango-BarbecueRTRM communication is done through an Application Controller, which is created when a Libmango Context is initialized.

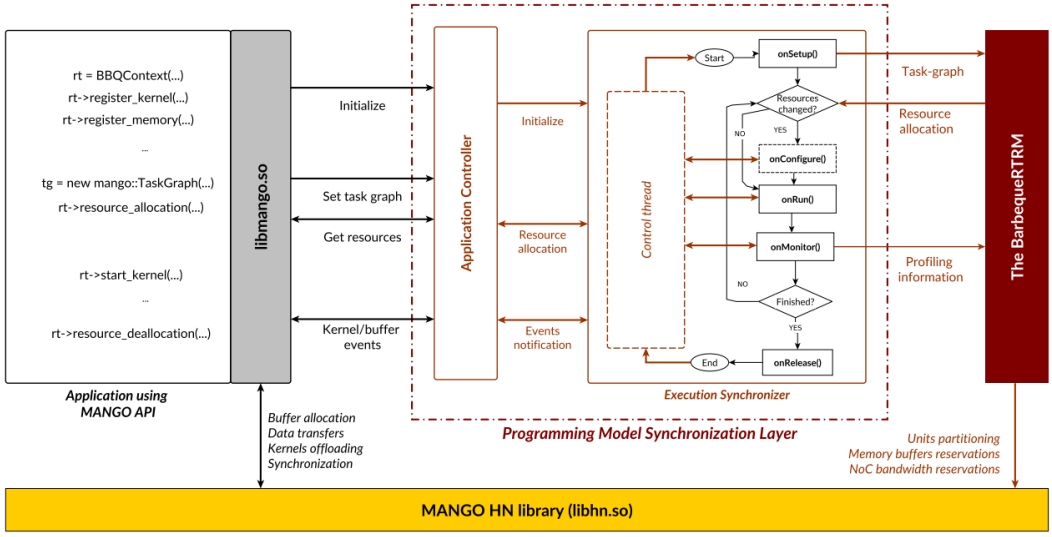


Figure 4.4: *Flow between BarbequeRTRM and the MANGO system*

4.4.3 Recipe file

- recipes

4.5 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 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. In this way, both BarbecueRTRM 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

managers that implement the API functionalities for their specific architecture.

4.5.1 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 4.13: *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 4.14: *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.

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

Listing 4.15: *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;
    }
}
```

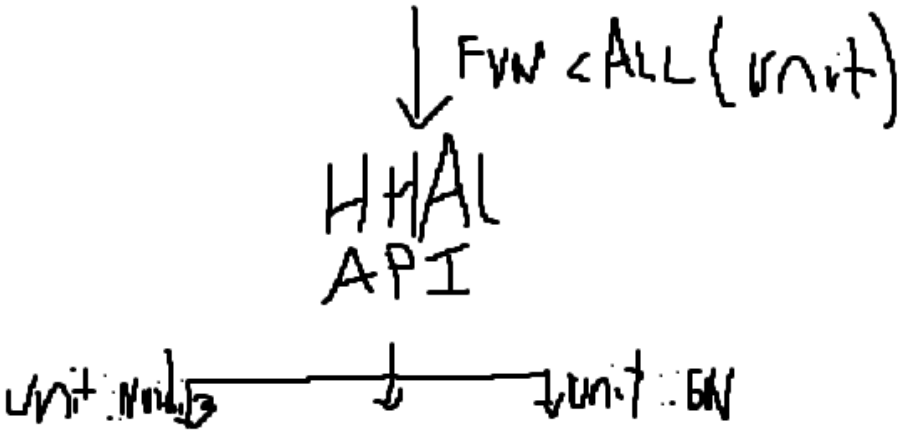


Figure 4.5: *HHAL API dispatching flow*

4.5.2 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 4.16: *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 4.17: *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 4.18: *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 4.19: *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 the Dynamic Compiler 4.5.3 subsection.

Listing 4.20: *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. 4.3.2

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 bits as well as long and float 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.

4.5.3 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. Ideally, once a solution is in place, kernels would be pre-compiled for optimal performance.

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 4.21: *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. 4.5.2

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 4.22: *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 4.6.1 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 4.23: *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 Caching subsection 4.5.3.

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 4.24: *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 4.5.3, defaulting to three days.

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 4.25: *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.

<RESULT OF ENTRYPOINT GENERATION EXAMPLE HERE>

4.5.4 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 4.6, referenced in code as `CudaApi`, which is the library implemented for launching kernels in Nvidia GPUs.

Listing 4.26: *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 4.14,

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 4.27: *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 4.28: *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 4.29: *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 when on kernel 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 4.30: *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 4.31: *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.

4.6 Nvidia Architecture Node

- ptx format

4.6.1 CUDA Compiler

4.7 GN

4.8 HN

CHAPTER 5

Experimental Results

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

6.2 Future Work

6.2.1 Task Graph Automatic Execution

6.2.2 GPU Kernel Parallelism Optimization

6.2.3 Host HHAL Manager

6.2.4 HNlib integration

Bibliography

[1] B. Pervan and J. Knezović, "A survey on parallel architectures and programming models," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pp. 999–1005, 2020.

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

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

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

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] "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.

[19] "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.

[20] "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.

[21] 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.

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

[23] "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.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

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

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

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

[38] 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.

[39] "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.

[40] 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.

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

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

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

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

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

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

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

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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," 02 2019.

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

[65] "Wikipedia: Data Buffer." https://en.wikipedia.org/wiki/Data_buffer, Accessed: Apr 2021.

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

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

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

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

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

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

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