TÉCNICO LISBOA

Faster CPU emulation and dynamic kernel compilation in Multi2Sim

Rodrigo Zambujinho Lourenço

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Pedro Filipe Zeferino Aidos Tomás
Prof. Nuno Filipe Valentim Roma

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Pedro Filipe Zeferino Aidos Tomás
Member of the Committee: Prof. Aleksandar Ilic

October 2019

Abstract

As more and more computers are used, power efficiency is a growing concern, for both mobile devices and large-scale server deployments. Batteries must last enough to be of practical use, and large-scale computing consumes large quantities of electricity and thus generates large amounts of heat. As such, it is important to improve power efficiency, both to reduce consumption and to increase computational capability within a power budget. As current scaling benefits in power efficiency are increasingly difficult to leverage, focusing on alternative lines of improvement can be fruitful. However, prototyping hardware is costly and morose. Unlike hardware, software is very malleable, and therefore we can exploit this property to more easily explore the design space before moving onto costlier prototyping.

To this end, we extend Multi2Sim, a simulator of central processing units (sCPUs) and graphics processing units (sGPUs), to address some of its shortcomings. Multi2Sim emulates, among others, the AMD Southern Islands microarchitecture, a relatively recent GPU architecture with an open-source register-transfer language (RTL) implementation, in addition to providing a runtime implementation for OpenCL, a standard for heterogeneous computing. Unfortunately, Southern Islands in Multi2Sim stopped in time when AMD deprecated its proprietary drivers, upon which Multi2Sim depended, and no new software could be run in it. In this work, we imbue Multi2Sim with the ability to compile new software, and modified its emulation capabilities to significantly speed-up execution ancillary to GPU emulation.

Keywords

Simulation, tracing, GPGPU, CPU, compilers, heterogeneous architecture.

Resumo

Eficiência energética é uma preocupação cada vez maior, tanto em dispositivos móveis como em computação de larga escala. Dispositivos móveis estão dependentes do uso de uma bateria, que tem de durar tempo suficiente para ser de uso prático, e as grandes instalações de servidores consomem grandes quantidades de electricidade, produzindo grandes quantidades de calor, que tem de ser dissipado. Dado que os métodos actuais de dimensionamento estão cada vez mais difíceis de aproveitar, é sensato explorar vias de melhoramento alternativas. No entanto, criar protótipos de *hardware* é caro e moroso. Ao contrário do *hardware*, o *software* é intrinsecamente maleável, e podemos aproveitar-nos disso para explorar desenhos alternativos antes de passar a prototipagem mais cara.

Deste modo, estendemos o Multi2Sim, um simulador de unidades de processamento central (CPUs) e unidades de processamento gráfico (GPUs), para abordar algumas das suas deficiências. O Multi2Sim emula, entre outras arquitecturas, Southern Islands, uma arquitectura relativamente recente de GPUs, bem como fornece uma implementação de um *runtime* de OpenCL, um norma para computação heterogénea. Dado que uma implementação em RTL (linguagem de transferência de registos) existe para a arquitectura Southern Islands, e que o Multi2Sim é o único simulador existente compatível com essa implementação, é um excelente candidato para prototipagem em derivados dessa arquitectura. Neste trabalho, damos ao Multi2Sim a capacidade de compilar novo *software*, coisa que deixou de ser possível quando a AMD descontinuou e indisponibilizou os seus *drivers* proprietários, dos quais o Multi2Sim dependia, e acelerámos a execução de partes ancilares à emulação do GPU.

Palavras Chave

Simulação, rastreio, GPGPU, CPU, compiladores, arquitecturas heterogéneas.

Contents

List of Figures

List of Tables

Acronyms

ABI application binary interface

ALU arithmetic and logic unit

API application programming interface

APP Accelerated Parallel Processing

CAL Compute Abstraction Layer

CPU central processing unit

CU compute unit

DMA direct memory access

DVFS dynamic voltage and frequency scaling

ELF Executable and Linkable Format

FPGA field-programmable gate array

GCN Graphics Core Next

GPGPU general-purpose graphics processing unit

GPU graphics processing unit

HSA Heterogeneous System Architecture

IR intermediate representation

ISA instruction set architecture

LDS local data share

LRU least-recently used

MIR machine intermediate representation

RAM random-access memory

RTL register-transfer language

SDK software development kit

SGPR scalar general-purpose register

SI Southern Islands

SIMD single instruction multiple data

SSA static single assignment

VGPR vector general-purpose register

VLIW very long instruction word

1

Introduction

With the widespread use of battery-powered devices, such as mobile phones, laptops, smart watches, and even cars, energy efficiency is more important than ever, since it means that for the same work a battery charge can last longer, which in turn means that the batteries themselves can have a longer lifespan because they need to be charged less often. At the opposite end of the computing spectrum, large-scale computing environments, such as datacenters and supercomputers, benefit from improvements in energy efficiency because they would cost less in electricity, not only because the machines would consume less but also because they would need less cooling, which is a significant cost in today's datacenters [1].

Energy efficiency can be improved on two fronts: software and hardware. Improving software, in fact, means better taking advantage of the hardware resources that are available or adopting more efficient algorithms. We can do this either by judiciously choosing where to run code (i.e. scheduling) or by reworking code to take advantage of the architecture. As an example of scheduling, tasks that are not latency-sensitive can be run on the little cores in big-little architectures, leaving the big cores turned-off, resulting in power savings [2, 3]. Reworking the code to exploit the architecture is also advantageous since, when performance matters, the faster the processor can complete a task, the sooner it can be powered off. Examples of architecture-specific optimizations include using new instructions or taking advantage of the memory architecture [4, 5].

This is due to the fact that there have been no good, general-purpose, solutions for fine-grained power management, so power saving comes in the form of a race-to-finish, to power off the die as soon as feasible. For example, dynamic voltage and frequency scaling (DVFS) works well at the whole-core scale, especially in tandem with scheduling, but not at the functional unit scale, because it significantly increases instruction latency [6]. While chaning the voltage and frequency transition, the processor does nothing, with the consequence that it wastes power during the transition.

Improving efficiency in hardware has traditionally been a by-product of die scaling because at each die shrink, transistors require less voltage to switch and thus consume less energy, giving hardware designers more resources to produce more powerful hardware. Therefore hardware has become both faster and more efficient, simply due to die shrinkage. This continuous improvement in transistor density at a lower energy cost and price has been named Dennard scaling [7] and has been valid for the past few decades. Nevertheless, this scaling has not been as fruitful lately. This is because clock rates cannot scale as they used to, since, as the chips become smaller, it is harder and harder to cool them down due to lack of surface area. Most programs do not make use of the whole hardware and, in fact, they cannot use the full chip at once due to thermal limitations, leading to the problem known as dark silicon: most of the chip is not used most of the time [8], hence wasting silicon.

On the hardware side, there are a few options to improve efficiency. One way is to specialise chips to the application. Using a custom chip specialised to a certain application has advantages in both perfor-

mance and energy efficiency [9–11], with the obvious disadvantage that a specialised circuit will either perform badly on another task or will not perform at all. An alternative is to improve on the architecture, but this is hard since hardware is difficult to design and costly to prototype. As an example, improvements in the memory architecture would be welcome, because memory speeds have improved much slower than central processing unit (CPU) speeds, hence applications typically spend a lot of time waiting for memory unless they efficiently exploit the cache hierarchy. This is a problem known as the memory wall [12]. Finally, we can improve the technology that enables us to build circuits. As most improvements have been incremental and been bound by the physics of current designs, this would likely require a breakthrough in semiconductor devices.

1.1 Objectives

Because hardware is hard to develop and costly to prototype, software simulators have been developed to estimate circuit performance, power, and assorted metrics, in addition to testing functionality. Software simulators have additional advantages, for instance, we can model both hardware and software at any level of detail we desire, from the transistors in logic gates to nodes in a distributed system. It is useful to model at the microarchitecture level and then, if it merits further investigation, move on to circuit design and associated considerations.

In this work we focus on simulation at the microarchitecture level, and in particular, on an early member of the Graphics Core Next (GCN) family, Southern Islands (SI). This architecture is currently in use in AMD's general-purpose graphics processing units (sGPGPUs), so it is, in a sense, "industry-tested". That makes it a good candidate for research, since, not only it is relatively new, it also has enough usage to draw conclusions from. What also sets it apart is that an (unofficial) open-source register-transfer language (RTL) implementation exists for its compute units, MIAOW [13], and it even has an field-programmable gate array (FPGA) implementation, dubbed Neko. This enables researchers in this area to better focus their work, as they need not implement the whole core from scratch.

The only simulator that targets the Southern Islands instruction set architecture (ISA) is Multi2Sim. Unfortunately, Multi2Sim currently has a few problems. Though it implements an OpenCL [14] runtime, used to run kernels in a simulated SI GPGPU, users cannot run their own kernels since the provided runtime cannot compile them, thus limits users to run already existing applications provided in binary form. With this work, we aim to extend the Multi2Sim runtime to support new applications, by leveraging existing compiler technology.

1.2 Contributions

Our main contributions are as follows.

- We extended Multi2Sim's OpenCL runtime implementation to allow users to compile kernels in their applications, using the normal OpenCL application programming interface (API). This enables many more applications to be run under Multi2Sim.
- Multi2Sim now executes the application's host code over 100x faster, in the case where the application code runs natively on the machine, and the operating system supports the ptrace API.
- Several bugs were fixed in the GPGPU simulator, support for new x86 instructions was added, and more Linux x86 system calls are supported, which means that more x86 binaries can be emulated, in addition to being run under ptrace.

1.3 Document Organization

This document is organised in six chapters. The rest of the document is organized as follows:

- Chapter 2 describes technology we base our work upon. We introduce OpenCL, the runtime execution and memory model, Southern Islands, the underlying hardware architecture we are targetting, and LLVM, the compiler toolkit we use to target the hardware.
- We describe Multi2Sim's architecture and functionality in detail in chapter 3: how is the project organized, how it loads and runs programs, what does it take to emulate an instruction, and how it simulates the microarchitecture.
- Our work is laid out in chapter 4. We begin by describing AMD's proprietary format, followed by how to embed and use LLVM to compile application kernels. Then we describe how to employ the ptrace API to accelerate the CPU portion of the application.
- Chapter 5 evaluates our work on several benchmarks, including ones not previously available to be run under Multi2Sim.
- Chapter 6 offers a retrospective on our work, what conclusions we draw, what were our contributions, and directions for future work.

Throughout the document we write some numeric constants with the base specified in subscript. If no base is specified, the number is in base 10. For example, $2A_{16}$, 42_{10}, and 42 all represent the same number.

2

Background

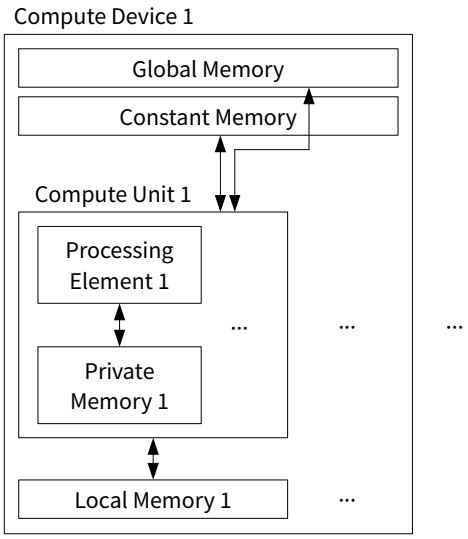

Figure 2.1: The OpenCL abstract memory model.

As noted in the introduction, Multi2Sim is the only simulator that targets the Southern Islands graphics processing units (sGPUs) and, therefore, it will be our work's main subject. Other simulators exist, such as GPGPUSim [15], gem5 [16], and others, but they lack a full GPGPU core to validate in hardware any design that is implemented in them. In this chapter we cover a popular programming API that targets many accelerator devices, OpenCL. OpenCL is the interface that allows applications to interact with the emulated Southern Islands GPU in Multi2Sim. We will then detail Multi2Sim's architecture and simulation facilities, and we will conclude with a short description of LLVM, a compiler toolchain of which we will make use later, when addressing some of Multi2Sim's shortcomings.

2.1 OpenCL

OpenCL is a standard for data and task parallel programming on heterogeneous devices [14]. Unlike CUDA, OpenCL is an open standard, and is designed to work with GPUs and other devices, such as domain-specific accelerators and CPUs.

The standard specifies an API, a language, a platform, and a low-level hardware abstraction. An OpenCL platform comprises two parts, the host platform and at least one device. Applications are run on the host and are subject to its requirements. Through the OpenCL API, applications can leverage the devices to accelerate their code.

As seen in fig. 2.1, devices are specified to contain at least one compute unit, with each compute unit having at least one processing element. Computations are specified using *kernels*, that is, functions that instantiated and run per each point in an index-space. To these instances we call *work-items*, and to the index-space that is specified by the application we call *NDRange*.

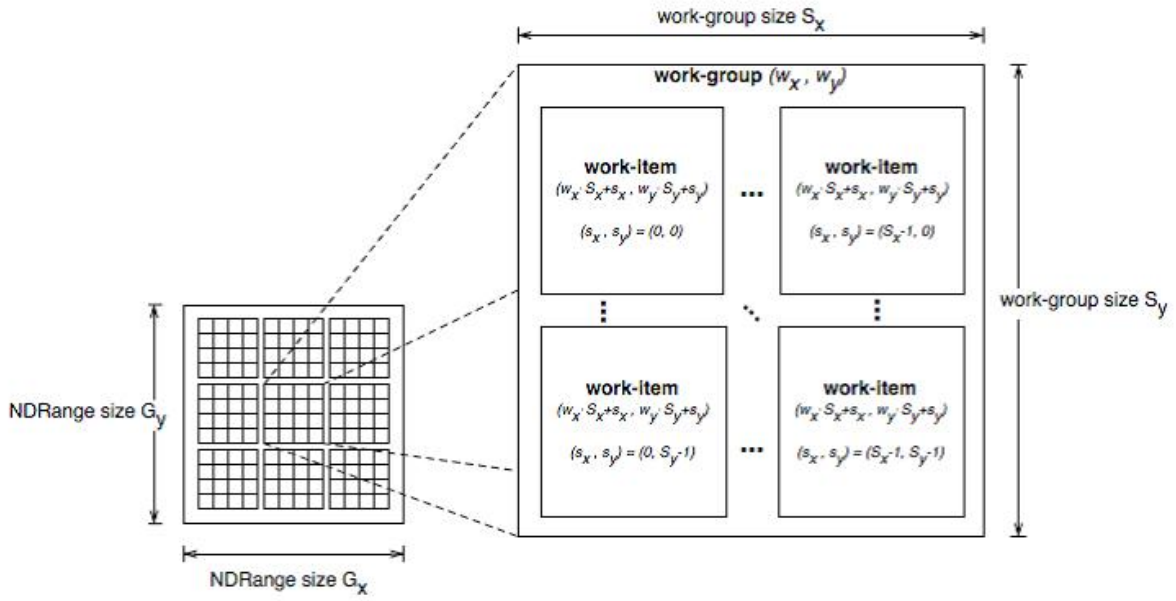

Figure 2.2: A 2D visualization of an NDRange, courtesy of the OpenCL 1.1 specification [14]. Here, w denotes the work-group ID, and s denotes (local) work-item ID.

The space is a one-, two-, or three-dimensional "box", that is, a line, a rectangle, or a rectangular cuboid, respectively. An NDRange is defined per dimension with an offset F and an extent D. For the ith dimension, a work-item must have its respective coordinate in $[F_i, F_i + D_i[$. As the point globally identifies the work-item, it is also called the *global ID*. Work-items are most useful for fine-grained data parallelism. Work-items execute in the processing elements, so each work-item has its own private memory, not visible to other work-items.

As a means of supporting more coarse-grained data parallelism, work-items are grouped into *work-groups*. Work-groups are likewise uniquely identified by a *work-group ID* as part of a space akin to an NDRange, but with a fixed zero offset. The work-group extents should evenly divide the NDRange extents. In this way, work-items are evenly divided amongst work-groups. Within a work-group, its work-items are uniquely identified by a *local ID*, making it possible to refer to a particular work-item either by their global ID or by a combination of its local ID and work-group ID. Work-groups run at the compute unit level, with the consequence that their work-items may communicate through local memory. This is shown in fig. 2.2.

Other than local and private memory, two other kinds of memory are defined: global and constant. Global memory is shared across all compute units and can be read from and written to by both the host and the kernels running on the device. Constant memory is writeable by the host but not by the kernels, and is used to set-up read-only data. Private and local memory cannot be accessed by the host.

Several OpenCL platforms may be available on a given host. The application must choose one of

them through the OpenCL API. Once a platform is chosen, its devices can be queried to check if they meet the application's requirements. A context spanning admissible devices can then be created. Contexts in OpenCL are used by the runtime to manage the devices. On a context, one can compile and link kernels, manage memory objects, and run commands via command queues. Command queues are OpenCL's approach to task-level parallelism. (For example, a command is something like launching a kernel or issuing a buffer copy from the host to the device.) Operations (commands) in a command queue is issued in order to the devices[1] and run independently across queues. Barriers are available if some form of synchronism is required. When a command is inserted into a command queue, an event object is returned. These objects can be used to query status information and to run callbacks on termination, as well as to refer to issued commands.

OpenCL kernels are written in OpenCL C, a language based on the C99 standard [17], with a few extensions and restrictions. As an example, in OpenCL 1.1 the `double` data type is not required to be supported, being instead available by means of an official `cl_khr_fp64` extension on some platforms. Available extensions can be queried using the standard OpenCL API and are enabled via a `#pragma` directive in the kernel source code.

2.2 Southern Islands

Southern Islands is a subfamily of the GCN architecture, launched by AMD in 2012. It departs from the previous AMD TeraScale architecture in that is not a very long instruction word (VLIW) architecture, having instead a simpler (RISC) single instruction multiple data (SIMD) instruction set. The general structure of a GCN GPGPU is shown in fig. 2.3. As can be seen, the GPGPU contains several compute units, all of them connected to a dispatcher and the L2 cache.

A compute unit (CU) can be seen as a small SIMD core, with its own fetch/decode/issue/execute/write-back pipeline, register files and arithmetic and logic units (sALUs). It has 4 SIMDs, each 16 lanes wide, for a total of 64 SIMD lanes available per compute unit. Work-groups are scheduled in sets of 64 work-items at a time, called a wavefront. A wavefront executes in a single SIMD unit, 16 work-items at a time. It follows that at least 4 wavefronts must be assigned to a compute unit to fully exploit its parallelism. Subject to resource requirements, such as available registers and local data share (LDS) space, the dispatcher maps up to 40 wavefronts, not necessarily from the same work-group, per compute unit, with a limit of 10 per SIMD unit. The scalar unit, shared amongst all wavefronts, has access to the scalar and vector condition codes, and is used to direct control flow and mask vector execution.

A wavefront can allocate up to 112 scalar and 256 vector registers, stored in the scalar and vector register files, respectively. The scalar register file is evenly split across the SIMD units, with 512 32-bit words being available per SIMD unit, for a total of 8 KiB of storage. The vector register file is also split,

[1] If specified, they can execute out-of-order, but the default is to execute in-order.

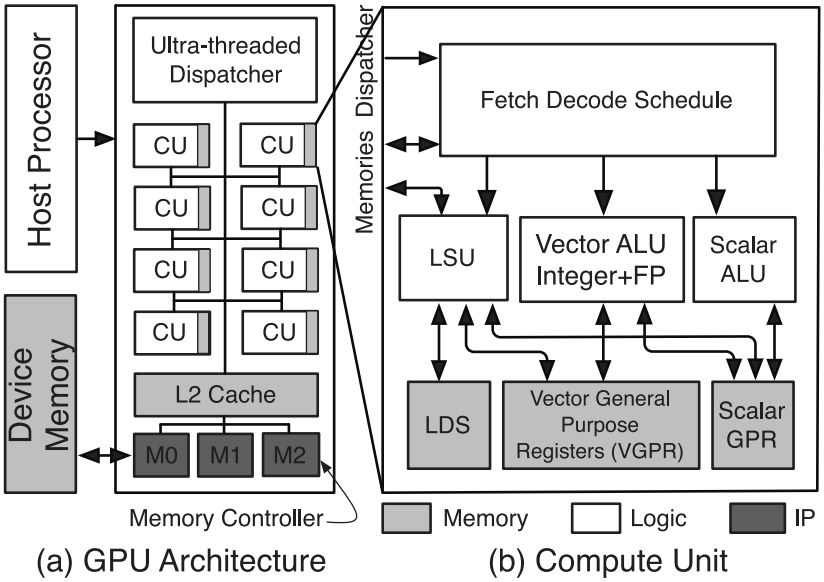

Figure 2.3: Broad overview of the GCN architecture for compute, from [13].

comprising a set of 64 32-bit registers. A total of 256 KiB of storage is provided, or 512 1024-bit registers per SIMD unit. 64-bit values are supported by merging two adjacent registers. This is needed because the vector masks and condition codes must be 64 bits wide, since a wavefront is 64 elements wide.

A 32 KiB instruction cache is shared per group of up to 4 compute units. It is 4-way associative, with a 64 B cache line, and least-recently used (LRU) replacement policy, being capable of fetching 32 B of instructions per cycle. Each group also shares a scalar 16 KiB L1 read-only data cache. It is read-only since the scalar unit is mainly used for control-flow and masking operations, whose results need not be written to memory. The L1 vector data cache is per compute unit, with 16 KiB storage and LRU replacement. It is coherent within a work-group, with some instructions available in the ISA to ensure global coherency. The last stop in the cache hierarchy is the L2 cache, which is shared among all compute units. It is 16-way associative, with 64 B cache lines and LRU replacement. It is partitioned into slices, each dedicated to a memory channel, with all slices and caches being connected via a crossbar.

Additional memory exists for communication between wavefronts, called the LDS. It allows fast synchronization, atomics, and swizzling, that is, vector accesses with re-arrangement. Each pair of SIMD units can send 16 lanes per cycle, with bank conflicts handled by the hardware. If the LDS can coalesce lanes from the SIMD units, it can process 32 elements per cycle, or two wavefronts every four cycles. The LDS can suppy a value as operand to the ALUs, with the value being broadcast to all lanes.

Instructions have two sizes, 4 bytes and 8 bytes, with some being followed by a literal constant. Table 2.1 lists all instruction formats in the Southern Islands architecture. For most compute kernels, there is typically no need for the EXP, MIMG, and VINTRP encodings, since these are most useful for pixel and vertex shaders in graphical applications.

Operation	Encoding	Description
Scalar	SOP2	Scalar operation with two inputs.
	SOPK	Scalar operation with one inline constant input.
	SOP1	Scalar operation with one input.
	SOPC	Scalar comparison operation with two inputs.
	SOPP	Scalar special operation with one inline constant input.
	SMRD	Scalar memory read from L1 memory.
Vector	VOP2	Vector operation with two inputs.
	VOP1	Vector operation with one input.
	VOPC	Vector comparison operation with two inputs.
	VOP3a	Vector operation with three inputs.
	VOP3b	Vector operation with three inputs and two outputs.
	VINTRP	Interpolation operation for pixel shaders.
	DS	Local/Global Data Share operations.
	MUBUF	Untyped vector memory buffer operations.
	MTBUF	Typed vector memory buffer operations.
	MIMG	Image memory buffer operations.
	EXP	Pixel and vertex shader export operations.

Table 2.1: Southern Islands instruction encodings.

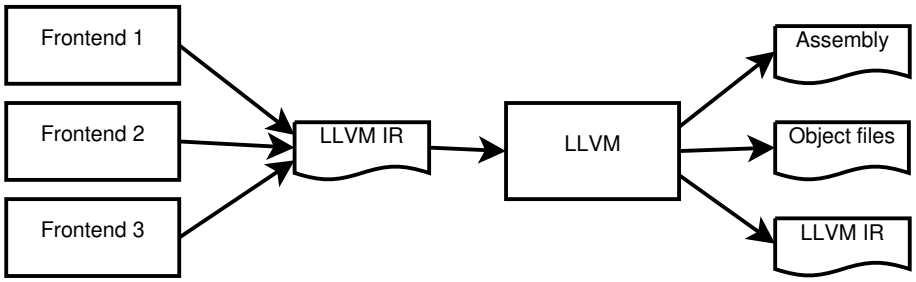

Figure 2.4: Overview of using LLVM.

2.3 LLVM

LLVM [18, 19] is an open-source compiler infrastructure, comprising several libraries and tools. It has a very permissive software license, meaning it has few restrictions in how it can be used. This is one of the reasons for its widespread adoption, since you are not forced to release your software's source code. In contrast with other compiler toolchains, such as GCC, it has been designed as a set of libraries instead of as a monolithic compiler, being easier to integrate into other applications. LLVM can generate code for AMD GPUs, and in particular, for the Southern Islands architecture. It also implements many of the OpenCL C's built-ins, under the umbrella of the libclc project [20].

To interface with LLVM, a language called LLVM intermediate representation (IR) is used. This is the sole interface to LLVM and one of its output formats, the others being machine code in text and binary format, as can be seen in fig. 2.4. This means LLVM does not actually understand most programming languages; instead, a *front-end* is used to compile from any given language down to LLVM IR. Part of

the LLVM project is a front-end for languages derived from C, such as C++, Objective-C, and OpenCL C, called Clang. Since it supports OpenCL C, we will make use of it and, by extension, LLVM, to make Multi2Sim a more complete simulation platform.

LLVM IR is low-level enough to be straightforward to translate to machine code for most architectures. All values are explicitly typed and can only be assigned to once, that is, they can only be defined and never overwritten. This latter property, called static single assignment (SSA), is useful because it simplifies some analyses and transformations on the code. Values whose definition depends on control-flow are selected using Φ-instructions, which define the value depending on where the control-flow came from.

LLVM IR is also completely self-contained. This means it can contain all relevant data, such as debug metadata. It has well-defined text and binary representations, enabling easier composability and debugging. IR code is organized around *modules*, which are loosely equivalent of a *translation unit* in C/C++, that is, the source after full pre-processing, suitable for immediate compilation. Modules are composed of functions and global variables. Functions have an entry point and zero or more exit points, and are graphs of *basic blocks*, which is a set of instructions that begins with a Φ-node and ends with a branch or return, with no branch targets in between, so the code always executes from the beginning to the end linearly.

Another advantage of LLVM being a set of tools and libraries is that front-ends are highly flexible in what they can do. For example, when you use LLVM you can also register your *passes* to run on the IR, be it for analysis or transformation. This is useful because it means that one can do language-specific analysis and optimization, but still use the whole of LLVM and all its *back-ends*.

Operations on the IR are called passes, and are run by a *pass manager*. The pass manager is responsible for scheduling the passes, that is, to decide in which order and how many times to run them. Each pass declares what passes it depends on and which passes it invalidates. A pass is restricted in what state it can track because the plan is to fully parallelize the execution of the passes. One can arrange passes to run on modules, functions, or with a finer granularity.

In fig. 2.6 we see the general organization of LLVM's back-end. LLVM can target several architectures, such as x86, PowerPC, ARM, and, more importantly, AMDGCN. LLVM's code generator is independent of the actual target: all the target specifics are handled by specializing generic interfaces. The back-end is responsible for transforming the IR into machine code, in the form of assembly files or object files. The LLVM compiler is not responsible for generating final executables or libraries, but there is a linker, LLD, in the LLVM project, which uses some of the LLVM libraries to do its task.

Code generation for a target begins by instruction selection. Instruction selection is the process of converting generic and machine-independent code to machine-specific code. In concrete terms, it converts from LLVM IR into instructions that have meaning only for the particular target. This means

```
__kernel void
add(float __global *acc,
    float __global *val,
    int size)
{
        size_t ix = get_global_id(0);
        if (ix < size)
                acc[ix] += val[ix];
}
```

(a)

```
define amdgpu_kernel void
@add(float* %acc, float* %val, i32 %size)
{
entry:
  %0 = call i32 @workgroup.id.x()
  %1 = zext %0 to i64
  %2 = call i32 @local.size.x()
  %3 = zext %2 to i64
  %4 = mul nuw i64 %3, %1
  %5 = call i32 @workitem.id.x()
  %6 = zext %5 to i64
  %7 = add i64 %4, %6
  %8 = call i8* @implicitarg.ptr()
  %9 = getelementptr i8*, %8, 4
  %10 = bitcast %9 to i32*
  %11 = load i32, %10
  %12 = zext %11 to i64
  %13 = add i64 %7, %12
  %conv = sext %size to i64
  %cmp = icmp ult i64 %13, %conv
  %arrayidx2 =
      getelementptr float*, %acc, %13
  br %cmp, label %if.then, label %if.end

if.then:
  %arrayidx =
      getelementptr float*, %val, %13
  %14 = load float, %arrayidx
  %15 = load float, %arrayidx2
  %add = fadd float %14, %15
  store %add, %arrayidx2
  br label %if.end

if.end:
  ret void
}
```

(b)

Figure 2.5: A kernel that adds in OpenCL C and LLVM IR.

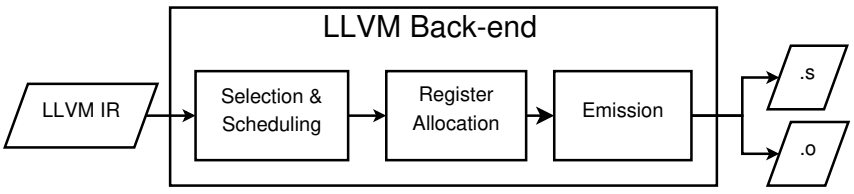

Figure 2.6: Overview of the LLVM back-end.

walking the instruction graph and generating corresponding machine instructions. The registers used for these instructions do not yet need to be physical, other than to enforce target constraints[2] and calling conventions.

As of this writing, LLVM does instruction selection using one of three methods: FastISel, SelectionDAG, and GlobalISel. The latter is not yet complete and is intended to replace the other two. FastISel is a simple and fast instruction selector, it generates poor code and does not support illegal types or more complicated selections, that is, it tries to select an instruction at a time. An instruction is said to be *legal* if it can be performed as-is by the target machine. A type is illegal if it is not supported by the target machine, e.g. a 4×31-bit vector in x86. If FastISel fails, or if one needs to generate better code, SelectionDAG is used. GlobalISel is intended to replace the other two because it can select and schedule instructions across basic blocks (something FastISel does not care about and that SelectionDAG cannot do), enabling optimizations that might be difficult to do otherwise, is more modular, so targets can control code generation better, and one can verify and test it easier.

SelectionDAG works by first converting the instruction graph into a graph of *SelectionDAG nodes*, where instructions can define multiple values (e.g. the x86 `div` instruction defines both the remainder and quotient). The translation is naive at first, and is then improved by multiple passes, legalizing types and operations, with clean-ups and optimizations interspersed. The way the passes work is by matching patterns in the graph, so as to reorder and convert instructions. These patterns can be architecture-specific, since we are already past the IR. To exemplify, it often happens when programming to need both a quotient and remainder. In x86, both are calculated with the `div` instruction, which returns both quotient and remainder in different registers. A simple compiler would generate two divisions to obtain both values, but with some cleverness it is possible to observe in the instruction graph that the second division is redundant, and thus remove the second division. Another example is seeing that we are adding a value after multiplying two others — we can generate a fused multiply-add instruction, if the architecture supports it.

Instructions and graph patterns are defined in a language called TableGen, a domain-specific language designed to write descriptions for a large number of records while reducing duplication as much as possible. Definitions are organized into records and classes, where both records and classes can inherit from classes. Entries in records are inherited and can be redefined, and there is support for looping, parameterization (akin to templates in C++), branching, and arithmetic. Another feature that is very handy for defining instructions is support for bit fields and ranges, including support for unknown bits. TableGen by itself is not very useful, since it only expands the definitions, but there are a number of back-ends that can generate C++ code. For example, a lot of the instruction selection is done by generating a table-driven graph matcher, leaving more complicated selection to hand-written C++

[2]For example, in x86, the 32-bit `div` instruction requires that the dividend is in the `eax` register, and writes to both `eax` and `edx` registers.

when needed.

After selection comes *linearization*, also called *scheduling*. This step converts the instruction graph to a sequence of instructions, broken up in basic blocks, which is a step closer to how processors execute code. The basic blocks are defined by the control flow while the order of the instructions within a basic block is defined by the dependencies between them and by a notion of cost. Instructions on most real-world designs take a variable number of cycles to execute, and some instructions cannot execute while others are executing because, say, there might not be enough functional units. Some processors can issue the instruction, inserting stalls as needed, but some require the compiler to insert *no-ops* (or independent instructions) to stall for time. As a real-world example, the compiler has to insert two cycles of latency between a `s_setreg` and a `s_getreg` on the same register in the Southern Islands ISA. All processors, but especially ones without out-of-order execution, benefit from instruction scheduling because the compiler can arrange the instructions such that the latencies are filled with useful work.

The code is now in a format called machine intermediate representation (MIR), which is a graph of basic blocks in an abstract representation of actual machine instructions. Though calling conventions and architecture restrictions may have put some values into physical registers, it may still have virtual registers, so next in the pipeline is register allocation. Register allocation converts the code from a virtually infinite register set down to a fixed register set. The way it works is as follows: (1) analyze the live ranges of used values, that is, for each value keep track of when it is defined and when it is last used, (2) build an interference graph of used values. Values are nodes in the graph and are connected if they are live at the same time, (3) color the graph with registers such that no two connected nodes share the same color. Sometimes there are not enough registers to color the graph. When that happens, registers have to be *spilled*, that is, stored into memory, so they are loaded afterwards when register pressure decreases. The code to spill registers into memory can be again scheduled, and register allocation can also be run again.

When allocating registers for AMDGPU, it is necessary to know if scalar or vector registers must be used. The OpenCL kernels are programmed as if each work-item executes in isolation from the other work-items; this is not always the case when executing as scalar registers are shared between work-items in a wavefront. Thus, to properly execute the code, values that diverge between work-items must be put into vector registers. To do this, the instructions are analysed to calculate what values diverge, an analysis aptly named divergence analysis. It begins with some known-divergent values, such as the result of calls to the OpenCL built-in `get_local_id`. This flag is then propagated transitively throughout its uses. Sometimes, it is possible to reverse divergence, as certain operations leave the registers uniform. For example, `xor`ing a value with itself always results in a zero, hence it does not diverge. Register allocation can then proceed once the scalarity of values is decided.

After register allocation, we are almost ready to generate machine code in an executable or assem-

blable format. Executable formats, such as ELF or Mach-O, are structured into *sections* of code and data, where the data sections store things such as global variables, constants, and tables. Our graph of basic blocks must be linearized to generate a stream of instructions so it can be put into its section(s). A topological order of the basic blocks is sufficient but cannot be directly applied, since the graph may contain cycles, for example, when the code contains a loop. Cycles are identified using an algorithm to detect strongly connected components, which is a maximal set of nodes in a graph in which from any node you can get to any other node. These components are isolated and treated, for ordering purposes, as if they were a single basic block. Within a cycle, a topological order can be applied.

Things that need an address, such as jump targets and globals, are uniquely labeled. The beginning of a basic block can be labeled because it usually is either the beginning of a function (a call target), or the target of a jump. It usually ends with either a return or a jump. The compiler can perform some optimizations here, such as detecting jumps to jumps and doing a jump directly to the end target, or detect that the jump is to the following instruction and eliminate the jump. When the code, data, and labels are in place, the compiler can now emit the final representation, be it in text format for an assembler, or directly into binary format.

2.4 Summary

As the need for more specific hardware is rising, the need to adequately take advantage of it has also risen. In this chapter we covered OpenCL, a standard for heterogeneous computation that was created to fill this void. OpenCL is used in Multi2Sim to target one of its emulated devices, a Southern Islands GPU, of which we give a very short overview. In the next chapter, we will describe Multi2Sim and how exactly it emulates applications. This chapter ends with a description of LLVM, a project on compiler infrastructure which is both powerful and easy to embed, and how it works. We will make use of it in chapter 4, when we embed it into Multi2Sim to make it more useful for general research.

3

Multi2Sim

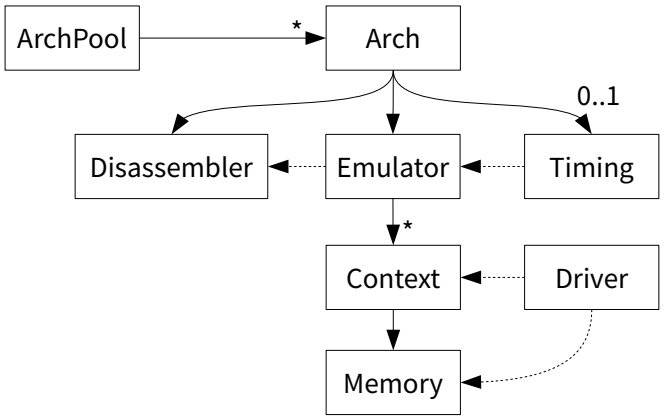

Figure 3.1: Multi2Sim's simulation architecture.

Multi2Sim [21, 22] is a tool for CPU, GPU, memory and network research. It supports several CPU and GPU architectures, comprising functional simulators (or emulators), timing simulators (or simulators), drivers, runtimes, and a network and memory stack. It provides guest applications with a Linux environment, as well as a selection of runtimes for the applications to link with, allowing them to communicate with the emulated GPU devices.

3.1 Overall architecture

Though Multi2Sim has several components, for architecture purposes it can be thought of as three: the simulators, the drivers, and the runtimes. Figure 3.1 shows the simulation architecture. On the simulation side, the top-most notion is that of the architecture pool, or `ArchPool`. It is a singleton where all architectures register their disassemblers, emulators, and simulators. Every supported architecture must have a disassembler and an emulator, with all other components being optional. Drivers are not directly registered, and will be explained in further detail in section 3.4. Multi2Sim's main loop revolves around of its methods, `Run`. This method steps once through all registered architectures that have a simulation going on, gathering and aggregating execution statistics that are displayed at the end of the simulator's execution.

The concept that is most central to Multi2Sim is that of the `Emulator`. The emulators are what execute the applications, orchestrating the different parts of Multi2Sim's support for that particular architecture. An emulator loads the application and its dependencies, as described in the next section, steps throught the different application contexts on its `Run` method. A good metaphor for the emulator is the kernel scheduler. The emulator also keeps track of statistics, such as running time and instruction count, to return to the architecture pool.

A `Context` maps one-to-one with an application thread. It contains all the relevant state of a process, such as its registers, its memory space, its open files, etc. Multi2Sim implements many of the system

calls of Linux 3.20, being exposed to the application as if it were the kernel. Contexts have a notion of being suspended and waking up. This is managed by the emulator, but can be caused by e.g. issuing a blocking system call. How these are handled is covered in section 3.3. The most-often used method is the `Execute` method, which fetches an instruction from memory, and disassembles and executes it.

As each context keeps track of its memory, memory must also be modelled. In Multi2Sim, `Memory` is implemented using a map from a page number to a fixed-size array, i.e. a page. The page size is configurable and defaults to 4 KiB, a common page size. Each page can be individually protected, that is, it has its own permission bits for read/write/executed. Though applications, by virtue of being executed in an unprivileged context, do not directly manipulate the page tables, Multi2Sim must implement something similar in order to implement system calls such as `mmap`. Implementing such a design also allows modelling of page-fault statistics, a useful metric in predicting performance. Another consequence of this design is that it is easy to implement shared memory, required to simulate inter-process communication.

The `Disassembler`, in tandem with the contexts, must understand the target architecture, albeit in different measures. The disassembler understands the executable formats supported by the architecture, being used to setup the contexts when an application is loaded. It must also understand and decode the architecture's ISA, unpacking the instructions into an easy-to-use format for the contexts to execute. The disassembler must also be able to display the instructions in human-readable format, which is useful for debugging. It lists all instructions in a macro file that is then included in the `Context` to generate a method per instruction and a dispatch table from an instruction opcode to a method.

To further understand how an application is behaving performance-wise, it is necessary to simulate the application in great detail. That is the purpose of the simulators, implemented in the `Timing` classes. The simulators do not actually compute any values or perform any work—they leave such work to the respective emulator; instead, simulators work at the functional unit, simulating the time it takes the functional units to do their work. For example, they model pipeline stages, stalls, cache latencies, and register pressure. We will not cover this subject in this document as it is not the focus of this work. The reader is directed to the Multi2Sim guide [23].

`Drivers` in Multi2Sim are quite different from the other classes, in that they are not associated with any particular architecture. Drivers are used whenever an operation on a virtual device is invoked. They expose themselves to applications via the file system. When calling a driver, the invoking contexts passes-in its memory descriptor. This must be so as the drivers need to read and write from a process' memory, for example, for a direct memory access (DMA) transfer between a device and its host, both emulated. Not all architectures have a driver—more details about drivers are covered by section 3.4.

Finally, we have the runtimes. The runtimes are not implemented in the Multi2Sim executable, being instead separate libraries. These libraries implement a programming API, such as OpenCL or CUDA,

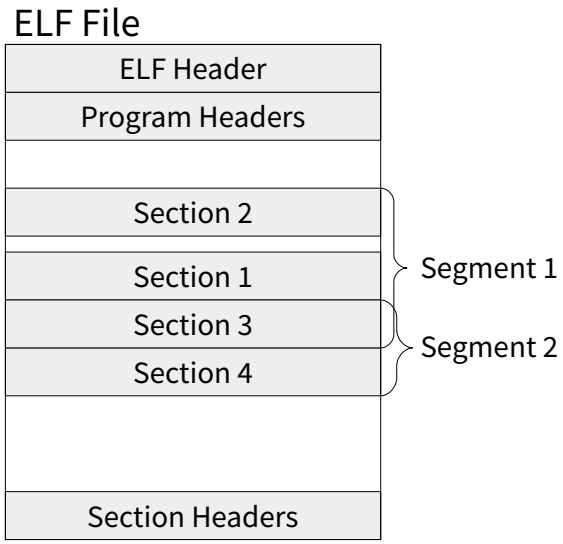

Figure 3.2: The structure of an ELF file.

that is meant to be linked to the guest applications. Underneath, the libraries open and communicate with the Multi2Sim devices exposed by the drivers. Thus, applications can still run natively while linking to the Multi2Sim runtimes, albeit with no access to the emulated devices. Runtimes are explored in more detail in section 3.4.

3.2 Loading Linux programs

Guest applications are supported in the Executable and Linkable Format (ELF) file format. In order to run a program, though, it must be first loaded. We start by describing the general procedure for loading and executing programs in Linux, and then we describe in more detail how this works in Multi2Sim.

As shown in fig. 3.2, an ELF file is composed of an ELF header, a table of program headers, and a table of section headers. The ELF header describes what kind of ELF file it is: what ELF version it is, what operating system is it meant for, what application binary interface (ABI) it uses, what kind of file it is, etc. It also describes where in the file the program and section headers reside. The "unit" of data is the section. A section header describes what kind of data it contains. For example, a section may contain machine code meant to be executed, or it may contain strings used in program, or symbols present in the binary.

Some commonly used sections and their respective contents include:

- `.text`—program code.
- `.rodata`—constant data used in the program. Typically strings or other constant data that the compiler does not place inline in the code.

Encoding	Bit pattern	Description
SOP1	101111101	Scalar operation with one input.
SOPC	101111110	Scalar comparison operation with two inputs.
SOPP	101111111	Scalar special operation with one inline constant input.
VOP1	0111111	Vector operation with one input.
VOPC	0111110	Vector comparison operation with two inputs.
VOP3	110100	Vector operation with three inputs.
VINTRP	110010	Interpolation operation for pixel shaders.
LDS	110110	Local/Global Data Share operations.
MUBUF	111000	Untyped vector memory buffer operations.
MTBUF	111010	Typed vector memory buffer operations.
MIMG	111100	Image memory buffer operations.
EXP	111110	Pixel and vertex shader export operations.
SMRD	11000	Scalar memory read from L1 memory.
SOPK	1011	Scalar operation with one inline constant input.
SOP2	10	Scalar operation with two inputs.
VOP2	0	Vector operation with two inputs.

Table 3.1: The discriminating bit-patterns for all Southern Islands instruction encodings.

struction entails decoding it, which is the disassembler's job. Decoding an instruction can be simple or complicated, depending on the architecture. For example, the Southern Islands ISA is simple to decode, while the x86 ISA is very complicated[1]. In this section, we begin by showing how to emulate a Southern Islands GPGPU, as it is this work's motivation, and then complete the picture with x86's emulation. This should provide a comprehensive overview of Multi2Sim's emulation capabilities.

The context begins by fetching 4 bytes from its memory. The disassembler is then called to decode the instruction. Instructions in the Southern Islands ISA are 4 or 8 bytes, possibly followed by a 4 byte literal constant. However, all encodings have a fixed bit pattern in the high bits of the first word, as shown in table 3.1. Therefore, it is never necessary to initially fetch more than a word to decode an instruction, since it is sufficient to look at the bit patterns roughly in the order shown in the table to discriminate between the different encodings.

Once an instruction is decoded, the type of instruction is checked. If it is a scalar instruction, a designated work-item is used to execute it while others do nothing in that emulator cycle. This correctly emulates the effect of using a scalar instruction, as the work-items do not diverge. Vector instructions are executed once per work-item. When executing instructions, some metrics are gathered, such as dynamic instruction count by type. A detail to consider when executing vector instruction is the execute mask, `EXEC`. This is a 64-bit register whose bits decide whether a particular work-item will execute or not.

Some instructions merit special atention. For example, the `V_READFIRSTLANE_B32` instruction, which reads one value from a vector register into a scalar register, executes in only one of the SIMD

[1] In fact, many x86 instructions are emulated using themselves: values are placed into registers and inline assembly is used to implement the instruction. This does mean that the x86 emulator will only work on x86 processors.

lanes, specified by the index of the first non-zero bit in the EXEC mask, and executes regardless of the execute mask. For other, more common, instructions, execution is dispatched to all work-items in the wavefront. Work-items execute one-by-one and follow roughly the same pattern: read input registers, compute the result, write output registers.

The register files are modelled using an array of integers per wavefront, for the scalar registers, and an array of integers per work-item, for vector registers. While this is not faithful to the hardware, it is simple to implement and, most importantly, it has the correct semantics for regular registers. Alas, this is not enough for control registers, because a write on a control register is only visible for reading after two cycles, therefore more information must be kept per register to allow for correct emulation. "Fixing" this behavior could break applications that depend on this behavior. Control registers provide access to values such as physical addresses of registers, counters, and exception status. These are not of much use to compute kernels, being more useful for driver or runtime programmers. As such, they have not been implemented because they have never been needed.

As specified in the OpenCL standard, execution is specified with NDRanges, split into work-groups, and further subdivided into work-items. Multi2Sim's emulation roughly follows this organization, with the addition of wavefronts, as it is how the hardware executes. In the Southern Islands emulator, there is no explicit notion of context like there is for other architectures. Instead, responsibility is split across different modules. The emulator owns the global memory for the GPU, being shared between all NDRanges. The kernel code is copied into each NDRange's private memory, not being present in the global memory. The local data share is allocated in the work-groups, being shared amongst all wavefronts. The wavefronts hold the scalar registers and instruction pointer, with the vector registers being held per work-item. This organization makes sense given the task at hand, except for the whereabouts of code in memory. Applications which depend on being able to read data from the code segment will not work, as the pointers will not make sense given they really are from different memories.

Unlike the Southern Islands emulator, the x86 emulator must deal with its environment. Files, locks, threading, networking, users, permissions, etc. must be implemented. This is because x86 is a general-purpose ISA and thus implements all kinds of applications, unlike the more niche Southern Islands ISA. The instruction emulation loop is similar, though, with the additional detail of handling system calls, as shown in fig. 3.3. It suffices to talk about how to implement system calls, since all other features derive from or are manipulated with these.

System calls in Multi2Sim are implemented using a dispatch table. All system call handler functions are put into a table, indexed by the system call number. Performing a system call is then a matter of bounds-checking and calling the appropriate function through the table. In x86, system calls are invoked using the `int $x80` instruction. Registers are used to pass arguments, with the system call number and result being placed in the `%eax` register. Each system call handler reads the registers from the context to

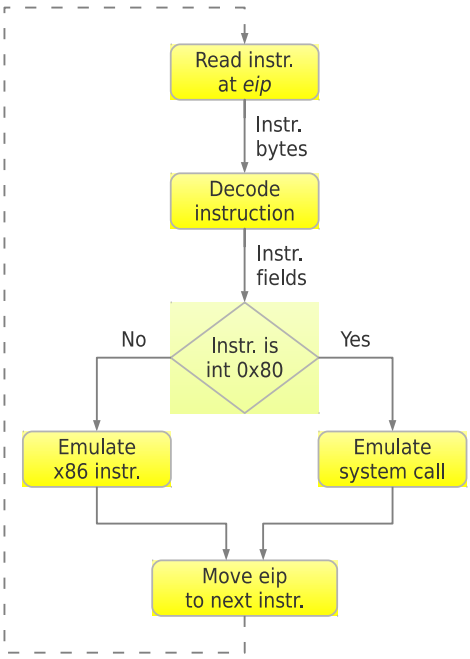

Figure 3.3: The Multi2Sim emulation loop. Though it depicts x86, the idea is applicable to all architectures. Taken from the Multi2Sim 4.2 guide [23].

access its arguments, but returns the result. The dispatcher then writes the result into its proper place.

Open files are kept in a file table. This table is shared between contexts, since Multi2Sim has little support for multiple running processes, having implemented just enough to run the forking part of the `system` C library call, leaving the execution for within the simulator. Thus, Multi2Sim really only supports threads, which (usually) share file tables and virtual address space.

The file tables are quite simple, being just a dynamic array of file descriptors. File descriptors contain a host file descriptor and a guest file descriptor. The reason for this data structure's existence if three-fold. First, it must support drivers, which are exposed through the virtual file system and thus require a virtual file descriptor. Second, it must keep track of the flags per open file, in order to implement proper semantics. Third, some files must be virtualized, as is the case of those under the virtual `/proc/self` file system. These files are used for introspection and reflection of processes. For example, a list of open descriptors (and where they point to) is accessible under `/proc/self/fd`, and a list of memory mappings is in `/proc/self/maps`. In Multi2Sim, only the memory mappings are virtualized, with other files being unavailable under simulation.

A final consideration we must have is how to handle input/output. Since guest applications do not perform I/O by themselves, it follows that the simulator must perform them on their behalf. This poses a problem: most I/O calls block, hence halting progress on threads that should not be blocked. To work around this, when issuing blocking calls, Multi2Sim suspends the context and spawns a thread to

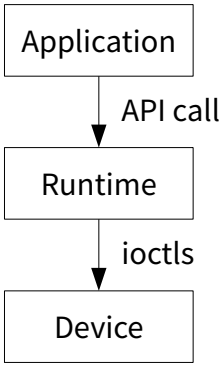

Figure 3.4: Communicating with the driver.

perform the I/O, delivering an event to the context when it finishes. It is also through this event system that signals are delivered, invoking the respective handlers, that timer events are delivered, and that threads are unblocked when waiting on I/O or low-level synchronization primitives. Events are checked by the emulator before and after stepping through the contexts.

3.4 Offloading computation to another device

Emulating just the host portion of a guest application is useful, but the most proeminent feature is to also emulate the accelerated portion of the application. Multi2Sim meets this requirement using two components, drivers and runtimes. When programming an application which uses accelerator devices, programmers target the runtimes, which implement a dedicated API. They then link with the provided runtime binaries and run the application under the simulator. In theory, since the runtimes are an implementation of an API, it is not required that the simulator is present for the application to work correctly. Having a fallback implementation when not running under Multi2Sim, such as delegating to a system implementation, is possible. Unfornunately, it is not implemented.

On the simulator side, the emulated device is exposed to the architectures through a driver. A driver exposes itself via the virtual file system under the `/dev` directory. Drivers register themselves in a driver pool, which is checked whenever a file is opened. Operations on these file descriptors are then forwarded to the respective driver. Incidentally, these virtual devices do not support the usual I/O operations. Instead, they communicate through the `ioctl` system call. The `ioctl` interface is very generic, taking a file descriptor, a device-dependent request, and an unspecified number of arguments, dependent on the request. These are used to control any kind of devices, such as terminals, tapes, GPUs, disks, etc. The general procedure to talk to a device is shown in fig. 3.4.

Because requests are device dependent, the device is free to do as it chooses. As such, to make things easy, device requests for emulated devices begin in the low positive numbers. This makes it trivial to build a dispatch table to jump on the received request to the respective handler. Drivers are given access

to the context wherein they are called, so they have access to registers and memory. Currently, no drivers make use of the registers, taking a single pointer argument per request, with all arguments packed behind it.

As a more concrete example, let us look at the implementation of the `clCreateBuffer` call. The function creates a buffer object; roughly, it allocates memory somewhere, usually on the device. If the buffer must live in the device, an `ioctl` is invoked with request `SIMemAlloc` and an argument pointing to an array, sized 1, that contains the allocation size in bytes. The return is a pointer to device memory, which is stored in the buffer object.

3.5 Summary

This chapter covered Multi2Sim, a functional and timing simulator for serveral architectures. We walked through the most important concepts while emulating an application for the Southern Islands GPUs. Each architecture makes use of a main emulator module, that manages all processes running in it. Emulators deal in contexts, which represent the notion of a process. Contexts contain all details pertaining a process, such as its memory and resources. It is in these contexts that instructions are executed, with the assistance of a disassembler.

Also discussed is how a program is loaded and executed in memory. The kernel, must load the binary into memory. This binary contains several data and metadata segments, of which some are mapped into memory. If the program is not standalone, an interpreter (dynamic linker) must additionally be loaded and executed to complete the program image in memory. After the stack and heap have been set-up, the program can then be executed.

It is then a matter of emulating the program instruction by instruction. Most instructions do not need any special attention, as they merely operate on data. Some, though, need special treatment as they need extra work on the simulator. Examples of these are system calls or barrier instructions. We saw how Multi2Sim emulates a Southern Islands application and how it deals with applications interacting with an operating system. Finally, we looked at how different architectures communitate, via the dichotomy between driver, running inside Multi2Sim, and runtime, linked with by the application.

4

Improving Multi2Sim

Guest applications, through the runtimes, launch kernels that are executed in simulated devices. Since Multi2Sim simulates binary code, kernels must be in a binary format to be executed. However, Multi2Sim provides no facilities to compile one's own kernels, depending instead on pre-compiled binaries. It did provide, in version 4.2, a separate OpenCL compiler called m2c. It partially supports OpenCL C and has its own machine code generation. It was removed in 5.0, presumably due to the lack of manpower to keep it functioning. As a consequence, there is no way for researchers to run new programs under simulation, stiffling research.

In this chapter, we will address this deficiency, by embedding the LLVM compiler in the OpenCL runtime. This creates a performance problem. As LLVM is a relatively large project, it has large binaries which must be emulated. Though it is now possible to run new applications, it is a morose process, and thus not a good user experience. We will also address this new issue by leveraging the operating system's tracing and debugging capabilities.

4.1 Compiling OpenCL kernels at runtime

When using CUDA, it is the NVIDIA compiler's job to build an executable that contains the CUDA kernel, so the runtime always uses the pre-compiled kernel. In OpenCL it is not so simple to load application kernels. To run a kernel in OpenCL, one must either create a program from source using `clCreateProgramWithSource` and dynamically compile the kernels, or one can load an already compiled program using `clCreateProgramWithBinary`. Programs in source form are specified in the OpenCL C language but nothing is specified about the format of program binaries other than it may contain either one or both of (a) device-specific executables and (b) an implementation-specific IR which will be converted to the device-specific executables.

This means that if an application wants to save the program binaries for use later, it must build the kernels and use the `clGetProgramInfo` API to extract the program binaries. At the time, the chosen OpenCL implementation was AMD's, as part of their AMD Accelerated Parallel Processing (APP) software development kit (SDK) 2.5. The runtime and the driver are proprietary software and hence not easy to examine and/or interface with. The OpenCL implementation shares many details with AMD's Compute Abstraction Layer (CAL) platform, including its binary format. AMD did publish in 2012 a document [24] where it partially describes the format but it is missing information necessary to produce and interpret the binary. In this section, we detail the format and show how to generate it so that it can be used in Multi2Sim.

Figure 4.1 describes the process and result of compiling OpenCL code using the AMD proprietary driver. It turns out that on Linux the program binary is an ELF file containing both the intermediate code in the form of LLVM IR, in the `.llvmir` section, and device-specific code, in the `.text` section. The resulting ELF is essentially a fat binary, that is, one that contains the program for several different

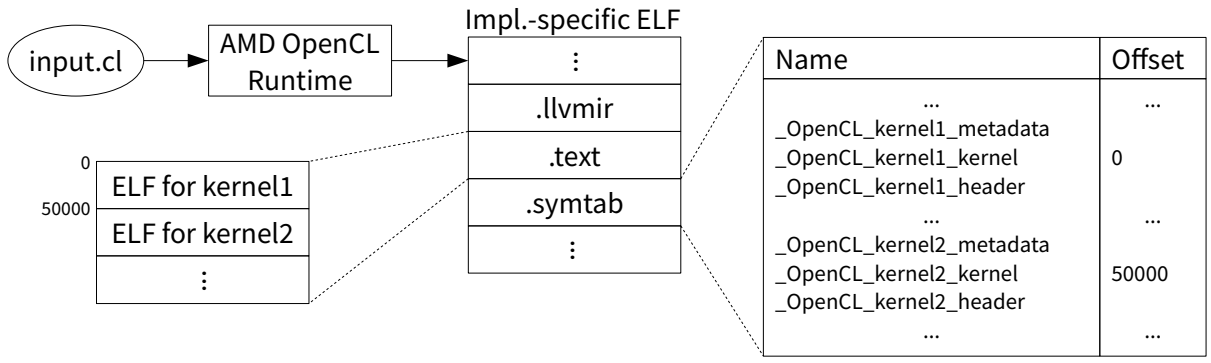

Figure 4.1: The structure of the implementation-specific executable generated by AMD's proprietary driver.

```
;ARGSTART:__OpenCL_binarySearch_kernel
;version:3:1:104
;device:tahiti
;uniqueid:1024
;memory:uavprivate:0
;memory:hwregion:0
;memory:hwlocal:0
;pointer:outputArray:u32:1:1:0:uav:10:16:RW:0:0
;pointer:sortedArray:u32:1:1:16:uav:11:4:RO:0:0
;constarg:1:sortedArray
;value:findMe:u32:1:1:32
;value:globalLowerBound:u32:1:1:48
;value:globalUpperBound:u32:1:1:64
;value:subdivSize:u32:1:1:80
;function:1:1027
;privateid:8
;reflection:0:uint4*
;reflection:1:uint*
;reflection:2:uint
;reflection:3:uint
;reflection:4:uint
;reflection:5:uint
;ARGEND:__OpenCL_binarySearch_kernel
```

Figure 4.2: Example CAL metadata for the BinarySearch kernel in AMD APP SDK 2.5.

architectures. In the `.text` section we can see that it contains several ELF files, each corresponding to one and only one kernel in the input code.

The `.symtab` section, the symbol table, contains three symbols for each kernel, one for metadata, one for a header, and one for the device-specific program binary. All such symbols are prefixed with `__OpenCL_`. The size and offset into the `.text` section of a kernel is given by the symbol table. The metadata and header are held in the `.rodata` section. The data for the header symbol are not used in Multi2Sim and are not described in AMD's document.

The metadata contains in a textual format kernel metadata, most importantly: (a) its parameters and its parameters' types, (b) what device it is targeting, (c) what driver version compiled the code, (d) what memory resources does this kernel use, and (e) where are the arguments mapped. These metadata are not described in any public AMD documentation, but are present in Multi2Sim's source code, which

means they either have direct or indirect access to internal AMD documentation, or they spent quite some time reverse-engineering the format. An example is given in fig. 4.2. A short reference of the most important tags follows. Tags not listed have either an unclear/unknown purpose or are self-explanatory.

- `version` : *major* : *minor* : *patch* — The version of the driver that compiled this kernel. It is also present in the outer ELF's note section.
- `device` : *device* — The device for which this binary object is compiled.
- `memory` : *type* : *size* — Memory requirements. *hwlocal* reserves that number of bytes in the LDS for a particular work-group. (The LDS is a fast memory shared between wavefronts in a work-group.) The other types, *hwregion* and *uavprivate,* are not documented or used in Multi2Sim. The size is usually 0 because the LDS size is either dynamically allocated or is specified in the CAL notes. We will describe these notes later in this section.
- `value` : *name* : *base-type* : *count* : *cbuf* : *offset* — Declares a parameter *name* of base type *base-type*. The *count* field denotes scalarity. For example, a parameter of type `uint` has a count of 1, whereas one of type `uint4` as a count of 4. The *cbuf* and *offset* fields indicate what constant buffer and with what offset the argument is passed in. Typically *cbuf* is 1 and offsets have been observed to always be a multiple of 16 bytes.
- `pointer` : *name* : *base-type* : *count* : *cbuf* : *offset* : *scope* : *buffer-num* : *alignment* : *access-type* : *unknown* : *unknown* — Declares a pointer parameter named *name* with base type *base-type*. Unlike with value parameters, pointer parameters seem to have a *count* of 1 whether they load or store vectors. Pointers are passed at offset *offset* in constant buffer *cbuf*. However, the memory they point to is held by buffer *buffer-num*, with *access-type* being one of `RO`, for read-only accesses, or `RW`, for read-write accesses. There are several *scope*s for a buffer, but the most relevant are `hwlocal` (goes to the LDS) and `uav` (goes to random-access memory (RAM)). Oddly, the exact access size is not specified; instead, its *alignment* is. The last two fields serve an unknown purpose.
- `constarg` : *integer* : *name* — Although a pointer already specifies whether the memory it points to is constant or not, this is present. It serves no discernible purpose, as it can be inferred from the corresponding pointer argument.
- `reflection` : *index* : *type* — Declares a parameter and at which *index* it occurs in the kernel signature. It also specifies its OpenCL *type*.

Each binary contains multiple segments: one for the encoding dictionary, marked by the special segment type PT_LOPROC $+ 2$, and n pairs of note and loadable segments, each corresponding to a specific device described in the encoding dictionary. The encoding dictionary thus extends the program

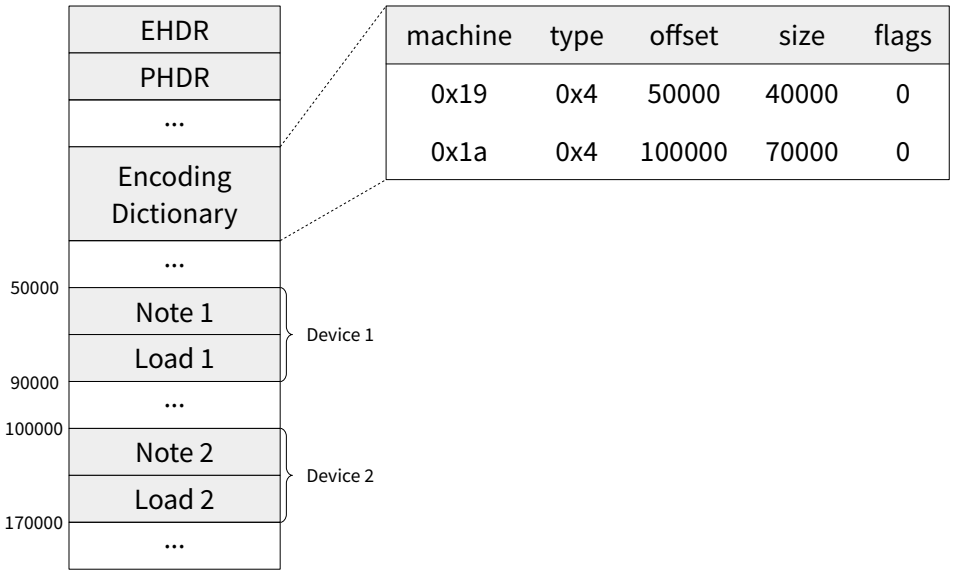

Figure 4.3: The structure of an encoding dictionary within a kernel executable.

header table by specifying to what device type each segment pertains. PT_LOPROC is a constant defined by the ELF specification that, when used as a program header type, denotes the lowest bound for processor-specific semantics.

Each entry in the encoding dictionary specifies a device type, an offset into the file, and a size. It represents a "meta-segment" of code and metadata for a specific device. There are usually two entries in the encoding dictionary. The first one, of machine type 19_{16}, seems to be unused and contains no useful information for emulation. The second entry is the actual device data, and has had several machine types for the same kernel across different driver versions.

The note segment is where the metadata lives. Each note begins with a header that specifies its type, how big is the note, and a mandatory 8-byte signature that must contain the C string `AMD CAL`. The data for a note come just after the header. This structure is properly documented in AMD's document about the CAL platform. However, the exact values for the notes are not in the document or in any public documentation. They are, though, in Multi2Sim's source code. The documentation indicates that not all notes must be present in the binary, but does not say which are mandatory and which are optional.

A list of addresses and their names can be found in Multi2Sim's source code. A short description of the more interesting ones:

- `Inputs`, `Outputs`, `SamplerMap`—sparse lists of texture inputs, color buffer outputs, and samplers. These are used to specify shaders for graphics applications but can also be used for general image processing.
- `ConstantBuffers`—a list of buffers to be mapped into memory. The size of a buffer is given in number of `vec4f` constants, that is, four single-precision floats (i.e. 4×32 bits), and hence

constant buffers have sizes that are multiples of 16 bytes.

- `ProgramInfo`—device addresses to configure. Each entry is a pair (address, value). Some addresses are just metadata about the target device while others contain metadata about the kernel. A few interesting addresses:
  - `[80001000`$_{16}$`:80001040`$_{16}$`]`—user elements. A user element is a mapping from a resource to a block of scalar general-purpose registers (sSGPRs). SGPRs are set before the kernel begins execution.
  - `80000082`$_{16}$—LDS size in bytes.
  - `00002e13`$_{16}$—`COMPUTE_PGM_RSRC2`. This is a 32-bit bit-field describing flags and sizes used by the kernel, such as whether it uses scratch memory, what dimensions are enabled, how many LDS blocks are used, etc. The size of an LDS block varies from device to device but in SI it is 256 bytes.

Having described the executable format, we now describe the calling convention. The calling convention is how one maps from the kernel arguments, from the perspective of the user, to how the machine code receives them. Binary code sharing the same calling convention is interoperable. Vector general-purpose register (VGPR) 0 always contains the local thread ID for the first dimension of the NDRange. If further dimensions are enabled, VGPRs 1 and 2 contain their local thread ids. No further VGPRs are used. The first SGPRs are configured using the addresses in the `ProgramInfo` note, starting from SGPR 2. This is because the SGPR pair `[0:1]` contains a 64-bit address pointing to the kernel arguments, with each argument being aligned to a 16-byte boundary. This is because arguments are passed in a constant buffer, where all entries are 16 bytes. These SGPRs are collectively called user SGPRs and all work-groups share the same initial values. The number of user SGPRs must match the value indicated in Compute PGM RSRC2. Following the user SGPRs are the system SGPRs, that contain in order the work-group ID for each dimension, if enabled. The first dimension is always present.

It so happens that LLVM is not fully compatible with this calling convention. The main problem is that LLVM expects the arguments to be aligned to their natural alignment (e.g. 4 for int, 8 for double) and offset by 36 bytes, to make room for work-group metadata. In addition, work-group metadata is also not passed to the kernel through different registers, as it is present at offset 24 in the kernel arguments buffer. We are then presented with two choices, to either modify LLVM to support the CAL calling convention or to modify Multi2Sim to support LLVM's calling convention. We chose to modify Multi2Sim because the old calling convention is not used anywhere else and in supporting LLVM code we are a step closer to being able to support binaries compiled with other toolchains, such as ROCm.

When compiling with LLVM, we can get two kinds of output. We can use LLVM only as a middle-end optimizer and use a custom machine code emitter to generate binary code, which was the ap-

proach taken in m2c, Multi2Sim's previous compiler. This has additional complications, since the whole back-end is our responsibility, so machine-dependent optimizations, register allocation, ABI, and executable format considerations must be implemented. The alternative is to use LLVM's AMDGPU back-end, which takes care of most of the job, but the executable format is not quite right to run on Multi2Sim, since it is not compatible with AMD CAL.

Unlike with Multi2Sim's m2c, we have taken the second approach. We use LLVM to generate the target code. There is a caveat: LLVM does not generate executable files directly; instead, it generates *object files*, which usually contain target code but may lack code from other sources to perform its function. They must then be passed to a linker, whose job is to resolve these dependencies and can generate an executable or a library. This is the way applications are typically compiled, with each translation unit[1] compiled individually and linked into a library or executable. Although it is possible, OpenCL programs are typically not spread across several files that are compiled independently, being instead in a single file or several files that are concatenated together. This means that our resulting object file is, in principle, standalone, so what we do is copy its `.text` section's contents (the machine code) and package it in the CAL ELF format. If the kernel's source code is not standalone, that is, if it calls functions in other kernel programs and will thus require linking Any other case is not currently implemented. LLVM is also useful here, since it generates the `COMPUTE_PGM_RSRC2` constant that we need for the program notes.

To generate the CAL metadata, it is enough to look at the kernels' signatures. First, we compile the code to LLVM IR. The first four fields are immediately appended, as they are constant for our use-case. We then go through the function signature, generating a tag for each parameter. Mapping LLVM types to CAL types is straightforward, since they share the same type names. Most types are supported, with the exception of structures, which are not implemented. We do not replicate exactly the CAL format as emitted by the driver because not every tag is useful. For example, we have no need for the `reflection` or `constarg` tags, as we target the Multi2Sim simulator. It is possible that this would break compatibility with the proprietary driver, but that is not a concern since it has been phased out.

Using the LLVM compiler was straightforward: all that needed patching was the OpenCL runtime's `clBuildProgram` function. Beforehand, it would check if a binary (loaded with `clCreateProgramWithBinary` or through the `M2S_OPENCL_BINARY` environment variable) was available and fail otherwise, even if kernel sources were present. Thus, the function was modified to compile the kernels. We used Clang to compile the kernel sources from OpenCL C down to LLVM IR, resulting in a module. As noted before, a module is essentially a representation of a translation unit. We then link this translation unit with the OpenCL built-ins, implemented by libclc. If the program does not depend on other programs or modules, the resulting module will be self-contained. (This is not enforced or verified in any

[1] The exact definition of a translation unit can be found in the C and C++ standards, but for our purpose it can be considered to be a source file after pre-processing.

way.)

Now that we have plain IR code, we can proceed to the next step: generating native code. As shown before, each kernel is separate from the others. It follows that we will need to generate several different binary streams, noting that Southern Islands does not support function calls. (This is not strictly true, as functions can be implemented using jumps and a stack. Southern Islands has both jumps and can implement a stack using regular memory. There is, however, no explicit support for function calls in the architecture.) As such, it is important that there is no "cross-talk" between the binary streams. The simplest way to achieve this is to make a copy of the module for each kernel. A specially devised pass is then run on each module, hiding all functions and kernels, except one "main" kernel, in a process called internalization, as in: making things internal to the module. The LLVM optimizer can then remove unused functions and inline used ones. These modules, containing a sole kernel, are then compiled to machine code. The instruction stream is extracted and placed in a new ELF file. This binary is structured as shown in fig. 4.3. The encoding dictionary is given a single entry, with the appropriate value for the architecture.

Once all binaries have been created, and all metadata strings have been generated, they are packaged into a binary as described in fig. 4.1, which can then be exported to the file system for later loading, or sent to the emulated device for simulation.

Using a new compiler has another set of challenges. LLVM generates code that is quite different from the code generated by the driver. This had the consequence that we had to implement many more instructions and fix several others. In total, 54 AMDGPU instructions were added/fixed. Unfortunately, the Southern Islands reference manual is quite vague in its descriptions of what instructions do, which led to some trial-and-error when attempting to implement instructions. Whilst running the x86 simulation, 12 new x86 instructions had to be implemented as well. This was quite easier because the x86 manual is very thorough, and most instructions are implemented using themselves. Some inconsistencies in file handling were fixed in `open` system call family, and some new system calls, used by LLVM and newer versions of the C library, were implemented.

4.2 Accelerating CPU execution

Adding LLVM to the runtimes exposed an unfortunate fact of simulation: it is slow. The Clang front-end and the LLVM optimizer are large portions of code which must be run. In general, the more instructions execute, the longer the running time. A modern, out-of-order, processor might breeze through hundreds of millions of instructions in a few seconds, but an emulator has no such fortune. In addition, while working on the simulator, it is useful to run the benchmarks in verification mode to check whether the CPU and GPU implementations agree. While this is not strictly needed, it is very useful while working on the simulator since changes impacting execution results are seen sooner. Motivating

running times will be given in chapter 5.

The idea to accelerate execution is simple: if we are running e.g. on an x86 processor, we should directly use it to run the x86 code in the guest application. This has several advantages: (a) no chance to misemulate instructions—they execute as the processor executes them, (b) it is much faster to run native code than it is to emulate its behaviour, (c) we can better focus on doing GPU research if we do not have to care about the CPU.

To mitigate the slowdown induced by the compiler, one might pre-compile the kernels and load them whenever needed. While this will indeed avoid running the compiler every execution, it is not a panacea. For example, if the kernel code is being worked on while the device is also evolving, this approach nets no benefit while requiring the user to perform one extra step. Additionally, as we will see, running the full-blown compiler every time will likely be faster than emulating with a pre-compiled kernel, with the advantage only growing as more time is spent in the application code when compared to compiling the kernels.

There are several ways to achieve this. We can run our application as if it were a virtual machine; however, creating a hypervisor is quite complex and we do not need to virtualize the whole machine. It is also possible to use dynamic binary translation, where we examine guest code for privileged instructions and patch it with a call to our code to service it. This is also very complex, since we need to both understand the ISA and examine and hotpatch a running executable, opening up the possibility of concurrency bugs. We also have to pay attention to self-modifying code since that may introduce privileged instructions that we must patch.

An alternative is ptrace. Ptrace is a mechanism by which a process can observe and control another process. Processes can be attached to, if the required conditions are met, or they can request that they are to be traced by their parent. It is a complicated and error-prone system to use since it abuses the traditional mechanism for child status notifications, that is, the `waitpid` (and related) system calls. Attaching to a running process, while possible, was not implemented for two reasons. First, detaching cannot possibly work because reversing the file descriptor mapping is impossible to revert, and second, we found no use-case for it.

To use ptrace, the simulator forks a child process. The child process invokes `ptrace` with the `PTRACE_TRACEME` request. Because it is in a ptrace-stop, we can set useful options, such as `PTRACE_EXITKILL`, to ensure all child processes are killed when the simulator terminates. Then the child process uses the `execve` system call to execute the target application. In the simulator, we must maintain the ptrace state per context, that is, per process, identified by its thread ID. Unlike in the usual Multi2Sim emulation, process IDs here correspond to actual processes, isolated from each other and from the simulator by the usual process isolation mechanisms.

The simulator now enters a loop where it listens for any child events using the `waitpid` system call.

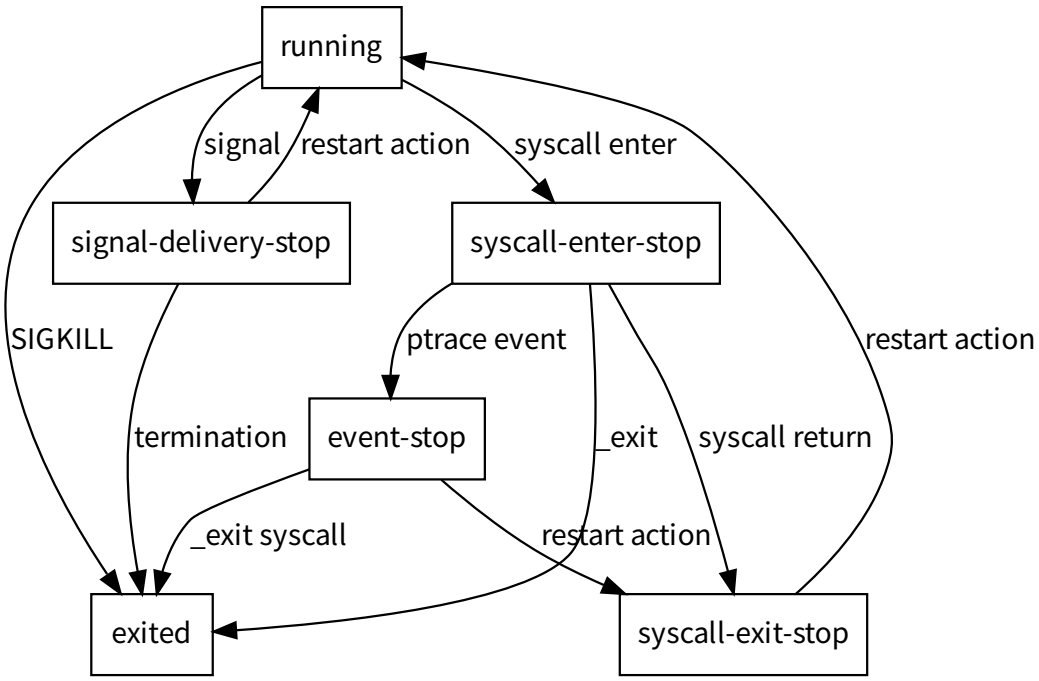

Figure 4.4: The possible states a process is in once it is stopped by ptrace.

It is not necessary, but it is useful to wait in non-blocking mode, so that if no action is required, then other ongoing simulations, such as GPGPU kernels, may continue, increasing parallelism. Once such an event arrives, we must look at what kind of event it is. A child may have exited (using the `_exit` system call), or have been terminated by a signal (using the `kill` system call), or have changed state. It is in these latter events that we are interested in, since there is not much to do in the former two. Figure 4.4 shows the possible states a process is in, regarding ptrace.

Handling a signal event is trickier, since when looking at the signal number, it is not clear whether it is due to ptrace or another process using the `kill` system call. The source of this confusion is that the kernel sends ptrace events using the `SIGTRAP` signal. Fortunately, there is an option for ptrace (`PTRACE_O_TRACESYSGOOD`) to send a signal with a special number, with value `SIGTRAP|0x80`. Now we must consider three different cases:

- If the process stopped due to a signal, we must remember what signal was sent, so we can resume the process with the appropriate signal. This is needed because we can suppress child signals if we so desire. We must deliver the signal as usual so that the installed signal handlers run on the child process.

virtual fd	real fd	driver
0	0	
1	1	
2	2	
3	4	
4		(si::Driver *)0x12345678
5		
6	6	

Table 4.1: An example of a virtual file table. The first three descriptors correspond to the usual standard input, output, and error, and point to the corresponding real file descriptors. Virtual descriptor 5 has been closed, so its entry is vacant. Virtual descriptor 4 corresponds to the Southern Islands driver.

- If it stopped due to a system call, we must look up in the associated context to know whether we are entering or leaving a system call, or whether we are in an event-stop. If the process is just starting a system call, we can inspect the registers to know what call it is and what its arguments are. We can cancel the call by replacing the system call number with an invalid one. Not only can we inspect the arguments, we can also modify them. In fact, since the process is in a ptrace-stop, we can read from and write to arbitrary memory locations. If we want to edit the call's result, we must remember what to do once we reach the corresponding syscall-exit-stop. To do this, we use a closure on the process context. Note that this handling depends on the architecture and operating system, so it is an separate, dedicated file.

- If the process received an event, such as when using the `clone` system call, or one of its variants, such as `fork`. Events can only be generated if their respective ptrace options are set. For example, using the `PTRACE_O_TRACEVFORK` flag enables the `PTRACE_EVENT_VFORK` event to occur when invoking the `vfork` system call. These process forking events must be enabled if new threads and processes must be traced. Events carry one word of information, the message, which can be queried using the `PTRACE_GETEVENTMSG`. In our use-case, the fork events are useful because we can set-up the file table for the process just before it starts, using the new process ID contained in the message. Events occur after the system call enters and before it exits.

The reason to virtualize file tables is that Multi2Sim and its drivers expose themselves to the application using the file system. As such, it must be possible to open/close/etc. the simulator-specific files, and have the associated code run. To do this, a custom file table is maintained per process. Since the file descriptors we expose to the application are not real, that is, not from the kernel, we call them virtual file descriptors. We maintain a table from the virtual file descriptors to either a real descriptor, or to a device driver, as shown on table 4.1.

To maintain this facade, we intercept all system calls that handle file descriptors, such as `read`, `openat`, `lstat`, etc. If the involved file descriptor has an associated real file descriptor, it is replaced

just before the system call is invoked, delegating the work to the kernel. This is in contrast to the old Multi2Sim model, where the simulator invoked the calls on behalf of the application. Otherwise, if the descriptor points to a driver, we must invoke the corresponding actions on the driver.

A closing thought on handling the descriptors. Allocating a new descriptor is simple: walk the table from beginning to end until a vacant position is found. If all spots are filled, resize the table. A linear scan suffices to search the table since processes usually use few file descriptors. The kernel has an upper limit to how many descriptors a process may open but we do not implement such a mechanism, since it is not immediately useful. If opening and closing files is very frequent, an ordered free list can be used to mitigate the cost of searching for vacant positions. When closing a descriptor, it is enough to clear its position.

Another detail that must be handled in system calls is pointers, and in particular, C strings. Given that pointers from the simulator and the application do not share the same address space, a mechanism must exist to allow this kind of access. This can be done using the ptrace API. The ptrace API supports two requests, `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`, which read from and write to, respectively, the supplied address. The disadvantage is that it is only possible to operate on a single word at a time. This is slow, since system calls have cost associated with them, and transfering single words at a time is inefficient.

A more efficient approach is to use the `process_vm_readv` and `process_vm_writev` system calls. These require that the process is in ptrace-stop mode, but can read whole chunks of memory at a time. They require a process ID and two arrays, one for the local addresses, one for the remote addresses. Each element in the arrays is a structure containing an address and a size, describing a sized buffer, called a `iovec`. The addresses are in the respective process' address space. The arrays need not be the same size, and are guaranteed by the kernel to be filled fully and in order, that is, the $n+1$th `iovec` is only read/written after the nth `iovec` has been fully used. The disadvantage is that these system calls are specific to Linux.

C strings pose an additional problem, in that they are not of a known size. Though things such as paths have an upper size limit, it is not practical or correct to always read up to the size limit. A read crossing a page boundary into an unmapped page could trigger a segmentation violation, and if not, will definitely abort processing the current and further `iovecs`. It is thus necessary to avoid spanning pages when reading from and writing to the application. The algorithm to read strings is, then, to issue a process read call for a single `iovec` beginning at the string's address, sized just enough to reach the end of its page. If the NUL terminator is found in that memory, halt, otherwise read the next pages one-by-one until the terminator is found. If the string is not terminated before reaching unmapped memory, it is an error in the application code and no behavior is guaranteed. Thus, the algorithm is correct.

A final note on ptrace, regarding system call ABI. On i386[2] Linux, there is only one system call ABI. Arguments are pushed in order to the `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp` registers. The system call number is put into the `%eax` register, which also performs the duty of holding the result of the call. The kernel is then invoked using the `int $0x80` instruction. Negative numbers in the result usually mean that an error occurred. Multi2Sim only supports i386 Linux applications.

On amd64[3], there are three system call ABIs. Code running in 32-bit mode can only use the i386 ABI, but code running in 64-bit mode can use either the x32 or the amd64 ABI. In the amd64 ABI, arguments are pushed in order to the `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9` registers. The system call number is put into the `%rax` register, and the kernel is called using the `syscall` instruction. Paralleling the i386 ABI, the call also returns a result in the `%rax` register. It is also possible to use the x32 ABI, but it is seldom used. It is most useful for programs that want to use the full x86-64 instruction set, but want to use 32-bit pointers. The x32 `ABI` and amd64 `ABI` are identical, with the difference being that the system call numbers are different.

The problem with all of these different ABIs is that, when a thread enters `syscall-stop`, there is no certain way to distinguish between them. It is possible to take the instruction pointer, `%rip`, and disassemble the previous instruction to know whether we are using `int $0x80` or `syscall`. However, unless all other threads in the process are stopped before we enter the `syscall-stop`, there is a race condition. It is possible that, by the time we inspect the system call instructions, another thread has overwritten the instruction stream, leading to bogus examination. Due to complexity, we do not check whether an invoked process, which would also be under the purview of the simulator, has any particular ABI, with the consequence that it will fail if amd64 applications are executed. This is a known problem of the Linux ptrace API, and there have been some attempts to fix it over the years. Only recently, in kernel 5.3, has a new ptrace request to obtain information about the system call been added. Alas, it was too recent to implement in time for this document. The request adds the ability to known not only whether it is a `syscall-enter-stop` or `syscall-exit-stop`, but also get the arguments in order without explicitly knowing the ABI, as well as to know what the ABI actually was.

4.3 Summary

In this chapter we described in detail the main problem with Multi2Sim, as well as our proposed solution. Multi2Sim has effectively stopped in time, since it cannot run any new code. Our work embeds LLVM into the OpenCL runtime, enabling new applications to run under Multi2Sim and allowing further research. However, this exposed a performance problem, as LLVM is a large body of code that must also run under simulation. Hence, we propose to fix the performance problem for GPU researchers by

[2] Also called x86 and IA-32. It means plain x86, with no 64-bit extensions.
[3] Also called x86-64 and x64, denoting the 64-bit extensions to the IA-32 architecture.

running the native portion of the application at full speed, taking advantage of ptrace, a debugging and tracing API.

We will present our results in the next chapter.

5

Experimental Results

This chapter presents our experimental results. In the previous chapter, we have shown some shortcommings of Multi2Sim. Multi2Sim did not allow for new software to run, because there was no tool to compile it. As such, we embedded a compiler in its OpenCL runtime, enabling new applications to be run. As the compiler is a large body of code, it exacerbated the fact that emulating code is slower than directly run it, so we used the operating system's support to execute the application faster.

In this chapter, we evaluate our changes to Multi2Sim. We will cover our experimental setup in section 5.1, where we detail our used benchmarks and our evaluation method, and in section 5.2 we show our results, as well as their analysis.

5.1 Setup

To show our improvements, we must demonstrate two things. First, we must show that we can, indeed, run programs that previously could not be run. On Multi2Sim's GitHub page[1], we can see several benchmarks, most of them binary-only, and some with source. The Hetero-Mark benchmark targets the Heterogeneous System Architecture (HSA) runtime, so we will not use it. The CUDA SDK 6.5 benchmark targets CUDA, which we also do not make use of. Of the remaining benchmarks, a single one is targetted at OpenCL, the AMD APP SDK 2.5 benchmarks, which are our baseline when comparing what code can be run. Version 2.5 of the SDK is from around 2012, so it could be that a newer version would contain new benchmarks. Unfortunately, AMD has pulled their APP SDK software (and samples) from their website. It is, however, available from other sources. The newer AMD APP SDK 3.0 samples are available in a git repository[2]. This newer version contains benchmarks that were not present in version 2.5.

To evaluate our changes, we will use the AMD APP SDK 2.5 OpenCL benchmarks. This suite comprises 36 benchmarks, using several features of the OpenCL standard. We will evaluate our work using 22 of the 36 benchmarks, because 14 of them use features our simulator does not implement, such as image buffers or scratch memory. Additionally, we will be using one benchmark, Template, from version 3.0 of the SDK, as it is the only one that both executes correctly in Multi2Sim and does not use unimplemented features, barring the others that are common to both versions. This benchmark was not previously available to run, as there is no pre-compiled version as there is for the other benchmarks.

To add another point of comparison, we will be using pocl [25]. Pocl, standing for Portable Computing Language, is an OpenCL implementation with several targets, most notably, the CPU. Pocl is also based on Clang and LLVM. The idea is to have a first approximation of the impact on execution time that compiling a kernel has, by comparing the runtime of a benchmark with pocl with the same benchmark, with pocl, but pre-compiling the kernels. The ratio of the time it takes with a pre-compiled kernel and

[1] Accessible at `https://github.com/Multi2Sim`.
[2] Accessible at `https://git.sr.ht/~rzl/amd-app-sdk-3.0-src`.

the time it takes if it must also compile the kernel will give us a baseline idea of how much of an toll compiling kernels takes in the benchmark's runtime.

In practice, the ratios will not hold because ptrace and simulation introduce additional, hard to predict costs. To have an idea of how much overhead ptrace has, a quick test was performed. Running the `ls` command on a directory with 650 entries showed a ten-fold difference in runtime when compared to the same program under `strace`, a system call tracing program that uses ptrace. `ls` is quite heavy on system calls, as all it does is query files and folders for details, having done about 2000 system calls. Hence, we can reasonably predict that our ptrace-accelerated Multi2Sim will run the same x86 code at most $10\times$ slower than the pocl version, because both LLVM and the benchmarks are not as heavy a system call user, being mostly computation, and because strace does quite a bit of work per system call.

All benchmarks were run on an Intel Core i5-6400, clocked at 2.7 GHz. Benchmarks are run with the `--verify` flag, which checks whether the GPU implementation and a reference CPU implementation agree in results. Pocl benchmarks are run single-threaded.

5.2 Results

Table 5.1 shows the runtimes of the benchmarks under pocl, averaged over 10 times to lessen the impact of noise. As we can see, pre-compiling kernels results in an average speed-up of about $2.07\times$, which means that compiling the kernel is taking about 52 % of the total benchmark time. This impact is somewhat magnified because the benchmarks complete quickly. A slower kernel, or running with more data, would offset the time it takes to compile the kernels.

We now compare the runtimes when using ptrace versus when emulating x86 code. Also presented is a new benchmark, not previously available to run on Multi2Sim, Template. This benchmark is from version 3.0 of the AMD APP SDK. All times are in seconds, with one standard deviation presented. As seen in table 5.2, there are large speed-ups in directly using the CPU when compared to emulating instruction by instruction. This happens for several reasons. In no particular order:

- The memory is also emulated. Setting up memory pages, doing bounds and permission checks, etc. takes time;
- Contexts do not execute in parallel, even if they could. Though Multi2Sim uses threads to allow for some degree of parallelism, namely when invoking blocking system calls, a single context steps at a time. The ptrace "architecture" keeps all contexts running in parallel. Only when they perform some action that triggers interception do they stop, waiting for the emulator to handle the event;
- As a consequence of the above two points, one instruction in the binary expands to several instructions when emulated. This is speculation, as it was not measured, but from the simulation results

Benchmark	pocl	pocl (pre)	Speed-up
BinarySearch	0.134 ± 0.002	0.058 ± 0.002	2.31
BinomialOption	0.150 ± 0.002	0.074 ± 0.002	2.01
BitonicSort	0.135 ± 0.002	0.058 ± 0.001	2.31
BlackScholes	0.179 ± 0.003	0.103 ± 0.002	1.75
DCT	0.135 ± 0.002	0.058 ± 0.002	2.31
DwtHaar1D	0.138 ± 0.003	0.060 ± 0.002	2.28
FastWalshTransform	0.140 ± 0.004	0.059 ± 0.002	2.37
FFT	0.139 ± 0.002	0.064 ± 0.004	2.18
FloydWarshall	0.167 ± 0.002	0.091 ± 0.002	1.83
Histogram	0.160 ± 0.002	0.084 ± 0.002	1.90
MatrixMultiplication	0.144 ± 0.003	0.068 ± 0.004	2.11
MatrixTranspose	0.134 ± 0.002	0.058 ± 0.002	2.30
MemoryOptimizations	0.403 ± 0.006	0.335 ± 0.013	1.20
MersenneTwister	0.160 ± 0.002	0.081 ± 0.002	1.98
PrefixSum	0.134 ± 0.002	0.059 ± 0.002	2.27
QuasiRandomSequence	0.157 ± 0.002	0.080 ± 0.002	1.95
RadixSort	0.138 ± 0.003	0.061 ± 0.003	2.25
RecursiveGaussian	0.219 ± 0.003	0.143 ± 0.003	1.53
Reduction	0.134 ± 0.002	0.059 ± 0.002	2.28
ScanLargeArrays	0.137 ± 0.002	0.063 ± 0.004	2.18
SimpleConvolution	0.134 ± 0.002	0.059 ± 0.002	2.26
SobelFilter	0.150 ± 0.002	0.073 ± 0.002	2.06
		Average	2.07

Table 5.1: Running times under pocl (plain) vs pocl (with pre-compilation). Times are in seconds, with one standard deviation presented. The speed-up provided by pre-compiling the kernel is provided.

it seems plausible that that one instruction can end up taking hundreds of instructions to emulate. This is not the case with ptrace, as instructions execute at full speed.

Some benchmarks deserve an additional commentary because their running time seems disproportionate. In the case of the FloydWarshal and MemoryOptimization benchmarks, the GPU code takes much longer to execute, and thus leads to smaller speed-ups.

The results in the table are annotated with some symbols. † means that the CPU and GPU implementations disagree. During the course of this work, many new instructions have been added and some have been fixed. Alas, the simulator still does not correctly implement the Southern Islands ISA. The reason is two-fold. Multi2Sim was implemented with the instructions generated by the proprietary driver in mind. LLVM generates very different code from what was previously Multi2Sim's target, and not everything is yet fleshed out. For example, LLVM generates data tables for certain instructions and places them in the middle of the instruction stream. This is incompatible with how Multi2Sim operates, as it has a separate memory for instructions to the one for data. This breaks instruction pointer-relative addressing when looking up values in these tables, breaking user code. The other reason is that the ISA manual is quite vague in what the instructions actually do, making the task of correct emulation much more difficult.

‡ is much simpler. It means that the benchmark crashes due to a segmentation violation. In other words, it accesses memory that is either unmapped, or it has no permission to access. This can be due to several things, but in these cases, it is due to a stray memory access in the kernel code, likely caused by wrong emulation of some instructions. Debugging GPU assembly code is hard, due to the many simultaneous values in registers, with the added complication of poor ISA description.

Finally, § is a peculiar one. The OpenCL runtime currently does not support exporting the kernels, so these are pre-compiled using an external tool. The only difference between this tool and the runtime, is that the runtime is run in i686 code whilst the tool is run in amd64 code. The source code is otherwise identical. This means that LLVM is generating different code when run in 32-bit mode when compared to running in 64-bit mode, leading to the pre-compiled version producing correct results, while the dynamically compiled version produces wrong results. This is yet to be investigated, but it makes sense given that 32-bit code is less and less used in desktops, and therefore less tested.

So as to refute the argument that pre-compilation is enough, we thus present in table 5.3 the running times of our ptrace-based approach when compared to x86 emulation with a pre-compiled kernel. As can be seen, using ptrace still consistently beats pre-compilation by a large margin. The Template benchmark is not presented because it does not implement loading and saving pre-compiled kernels. Using the same kernel, the only way for pre-compilation to beat a ptrace-based approach is if the kernel compilation is so heavy when compared to the running time of the kernel, that emulating the processor code hundreds of times faster will not bridge the gap. Additionally, nothing prevents one from

Benchmark	ptrace	emulation	Speed-up
BinarySearch	0.622 ± 0.001	672.670 ± 0.214	1081.76
BinomialOption	2.281 ± 0.003†	768.180 ± 3.720†	336.75
BitonicSort	1.693 ± 0.003	681.76 ± 14.37	402.79
BlackScholes	2.492 ± 0.006	858.630 ± 2.480	344.56
DCT	0.660 ± 0.002§	683.160 ± 2.470†	1035.11
DwtHaar1D	0.637 ± 0.002	677.650 ± 2.370	1064.63
FastWalshTransform	0.636 ± 0.002	686.25 ± 11.75	1078.98
FFT	0.919 ± 0.002	1098.54 ± 14.28	1194.97
FloydWarshall	21.903 ± 0.076	765.86 ± 1.43	34.97
Histogram	2.313 ± 0.001	2483.61 ± 8.73	1073.77
MatrixMultiplication	0.847 ± 0.002	1007.52 ± 3.97	1190.04
MatrixTranspose	0.626 ± 0.002	680.680 ± 1.440	1087.89
MemoryOptimizations	17.798 ± 0.021	919.500 ± 2.040	51.66
MersenneTwister	0.895 ± 0.002‡	1174.74 ± 1.06 ‡	1312.65
PrefixSum	0.633 ± 0.001	676.270 ± 1.480	1068.05
QuasiRandomSequence	0.812 ± 0.003†	782.16 ± 3.04 †	962.68
RadixSort	0.889 ± 0.002	911.05 ± 3.71	1024.71
RecursiveGaussian	1.538 ± 0.003‡	706.76 ± 1.59 ‡	459.66
Reduction	0.627 ± 0.001	673.220 ± 0.188	1074.24
ScanLargeArrays	0.676 ± 0.002§	600.69 ± 2.05 §	889.19
SimpleConvolution	0.663 ± 0.001	682.12 ± 1.09	1028.68
SobelFilter	1.033 ± 0.002†	712.96 ± 2.31 †	690.18
Template	0.631 ± 0.001	667.66 ± 2.64	1058.61
		Average	664.38

Table 5.2: Running times when using ptrace and when using x86 emulation. The speed-up of ptrace over x86 emulation is provided. † means the benchmark fails verification. § means that pre-compiling yields a different result than when compiling under ptrace. ‡ means that the process crashes.

Benchmark	ptrace	emulation (pre)	Speed-up
BinarySearch	0.622 ± 0.001	25.322 ± 0.174	40.72
BinomialOption	2.281 ± 0.003†	66.589 ± 0.577†	29.19
BitonicSort	1.693 ± 0.003	27.103 ± 0.099	16.01
BlackScholes	2.492 ± 0.006	84.868 ± 0.169	34.06
DCT	0.660 ± 0.002§	25.749 ± 0.032	39.01
DwtHaar1D	0.637 ± 0.002	28.282 ± 0.045	44.43
FastWalshTransform	0.636 ± 0.002	26.930 ± 0.014	42.34
FFT	0.919 ± 0.002	25.625 ± 0.292	27.87
FloydWarshall	21.903 ± 0.076	124.643 ± 0.202	5.69
Histogram	2.313 ± 0.001	50.349 ± 0.754	21.77
MatrixMultiplication	0.847 ± 0.002	38.671 ± 0.216	45.68
MatrixTranspose	0.626 ± 0.002	25.437 ± 0.010	40.65
MemoryOptimizations	17.798 ± 0.021	198.826 ± 0.535	11.17
MersenneTwister	0.895 ± 0.002‡	30.175 ± 0.069‡	33.72
PrefixSum	0.633 ± 0.001	26.102 ± 0.055	41.22
QuasiRandomSequence	0.812 ± 0.003†	57.548 ± 0.106†	70.83
RadixSort	0.889 ± 0.002	26.532 ± 0.058	29.84
RecursiveGaussian	1.538 ± 0.003‡	26.329 ± 0.123‡	17.12
Reduction	0.627 ± 0.001	25.424 ± 0.030	40.57
ScanLargeArrays	0.676 ± 0.002§	27.444 ± 0.097	40.62
SimpleConvolution	0.663 ± 0.001	25.670 ± 0.113	38.71
SobelFilter	1.033 ± 0.002†	40.282 ± 0.106†	38.99
		Average	30.42

Table 5.3: Running times when using ptrace execution and when using x86 emulation (with a pre-compiled kernel). The speed-up of ptrace over x86 emulation is provided. † means the benchmark fails verification. § means that pre-compiling yields a different result than when compiling under ptrace. ‡ means that the process crashes.

pre-compiling the kernels under ptrace.

5.3 Summary

In this chapter we showed the results of our work. We began by describing what benchmarks we would use. We then made predictions about the runtime of three different systems: pocl, Multi2Sim with ptrace, and Multi2Sim with x86 emulation. As predicted, using ptrace is significantly faster than emulating x86, because a lot of the simulation runtime is spent running everything but the kernel. We have also shown the impact in runtime of pre-compiling the kernels which, due to how heavy LLVM is, is very advantageous if the kernel rarely changes or if LLVM has to be emulated.

We also discussed some deficiencies we have in our current implementation. More on this in the next chapter, where we draw conclusions from our work and lay a roadmap for future improvements.

6

Conclusion

In this document we presented Multi2Sim, a simulator of CPUs and GPUs. We have shown its architecture, how it operates, and addressed two of its shortcomings. First, we added the ability to execute new applications since Multi2Sim had no tool to compile code. We leveraged Clang and LLVM to compile OpenCL kernels, both offline as a standalone compiler and online in the OpenCL runtime, so that new applications can be run. During the course of this work, many new instructions were implemented, resulting in a more solid emulation experience.

Embedding a compiler into the runtime exacerbated an inherent performance characteristic: simulators are slower than the corresponding CPU. In that manner, we accelerated the CPU portion of the simulator if the application does not require special CPU simulation. Using ptrace, a UNIX API to observe and control system calls in another process, we exposed the Multi2Sim devices whilst running the application at near-native speed, and thus sped-up the edit-test-debug cycle when exploring GPU architecture. With this work, we hope to put Multi2Sim again into a usable state for GPU systems research.

6.1 Future work

Multi2Sim still has a number of shortcommings that need to be addressed. The highest priority is to follow better engineering practices, such as adding tests. Multi2Sim has few tests, making it hard to ensure that changes to the simulator do not break something. Adding a battery of regression tests would be a very welcome addition for future researchers. As seen in the results, many benchmarks complain about wrong results, almost certainly due to an error in the implementation.

Lower on the priority list, but also quite important, is making Multi2Sim more portable, so researchers can use it elsewhere. Currently it is only possible to run it on x86 Linux. However, there is no intrinsic reason it has to be so. At least, it should also be possible to run on other architectures, e.g. ARM. This would entail cleaning the x86 simulator to remove the dependency on x86 assembly.

Another interesting addition would be to extend the OpenCL runtime to be able to run kernels in several, potentially different, devices at once. For example, the Kepler architecture is not accessible via OpenCL but it is possible to expose it. Additionally, running several different instances of the same architecture but with different configurations would also be an interesting use-case.

Finally, removing compatibility with AMD CAL is also an option, as it is a deprecated format and will not be used in the future. The currently supported toolchain is ROCm, and it is HSA-compliant. Supporting this format would be good for interoperability.

Bibliography

[1] E. Pakbaznia and M. Pedram, "Minimizing data center cooling and server power costs," in *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2009, pp. 145–150.

[2] M. Kim, K. Kim, J. R. Geraci, and S. Hong, "Utilization-aware load balancing for the energy efficient operation of the big. little processor," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–4.

[3] Y. G. Kim, M. Kim, and S. W. Chung, "Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1878–1889, 2017.

[4] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982. [Online]. Available: http://doi.acm.org/10.1145/356887.356892

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, Oct 1999, pp. 285–297.

[6] T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling," in *Proceedings of the 2000 international symposium on Low power electronics and design*. ACM, 2000, pp. 9–14.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[8] M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1131–1136.

[9] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 186–193.

[10] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts," *IEEE Micro*, vol. 34, no. 1, pp. 42–53, 2014.

[11] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[12] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[13] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol *et al.*, "Enabling GPGPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GPGPU," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, p. 21, 2015.

[14] Khronos OpenCL Working Group, *The OpenCL Specification*, Khronos, Jun. 2011.

[15] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 163–174.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[17] ISO/IEC, *ISO International Standard ISO/IEC 9899:1999 – Programming Language C*, ISO, Dec. 1999.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[19] The LLVM Project, "The LLVM compiler infrastructure project," https://llvm.org/, accessed: 2017-12-19.

[20] ——, "libclc," https://libclc.llvm.org/, accessed: 2017-12-19.

[21] R. Ubal, J. Sahuquillo, S. Petit, and P. L'opez, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[22] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[23] T. M. authors, *The Multi2Sim Simulation Framework*, http://www.multi2sim.org/downloads/m2s-guide-4.2.pdf, Jul. 2013, accessed: 2019-10-24.

[24] Advanced Micro Devices, *AMD Compute Abstraction Layer*, https://developer.amd.com/wordpress/media/2012/10/AMD_CAL_Programming_Guide_v2.0.pdf, Dec. 2010, accessed: 2019-06-28.

[25] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y