Adding support for heterogeneous parallel architectures to Julia

Georg Rollinger

Bayreuth Reports on Parallel and Distributed Systems

No. 8, Dezember 2015

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Adding support for heterogeneous parallel architectures to Julia

Erweiterung der Programmiersprache Julia um Unterstützung für heterogene parallele Architekturen

Georg Rollinger

December 18, 2015

A Thesis
submitted in partial fulfillment of the requirements for the degree of

Master of Science (M. Sc.)

University of Bayreuth

Committee:
Prof. Dr. Thomas Rauber
Dr. Matthias Korch

Contents

Abstract

English

Traditionally, using a GPGPU to accelerate arbitrary calculations has always been difficult. The common GPGPU platforms introduce new languages exclusively for programming the devices, runtime libraries and new tools that have to be integrated into the build process. That means that using them from any language is cumbersome, repetitive and error-prone. Obviously, there is a need for better integration of GPGPU platforms with programming languages.

We provide that using the Julia language and the HSA computing platform. In a first step, we simplify using the HSA runtime library from within Julia. Improving on that, we introduce modifications to Julia that obviate the need for external tools and free the user from interacting directly with the runtime.

Deutsch

Die Benutzung einer GPGPU zur Beschleunigung von Berechnungen ist von jeher schwierig gewesen. Die meistverbreiteten GPGPU Plattformen benötigen eigene Sprachen nur für die Programmierung der GPU, Laufzeitbibliotheken und neue Programme, die in den Buildprozess integriert werden müssen. Das führt dazu, dass ihre Benutzung umständlich, repetitiv und fehleranfällig ist. Es ist offensichtlich, dass wir eine bessere Integration von GPGPU Plattformen in Anwendungsprogrammiersprachen brauchen.

Unsere Arbeit setzt diese Integration für die Programmiersprache Julia in Kombination mit der HSA GPGPU Plattform um. Zu Anfang vereinfachen wir die Benutzung der HSA Laufzeitbibliothek von Julia aus. Darauf aufbauend nehmen wir Veränderungen an Julia selbst vor, die HSA's externe Sprache und Compiler unnötig machen. Darüber hinaus automatisieren wir die Interaktion mit der Laufzeitbibliothek, so dass der Programmierer davon befreit wird.

1. Introduction

Traditionally, using a Graphics Processing Unit (GPU) to accelerate arbitrary calculations has been difficult. At first, when there was no explicit support from device vendors, the programmer had to trick the GPU into running a calculation via graphics Application Programming Interfaces (APIs) like DirectX or OpenGL, not intended for that purpose.

Since then, many General Purpose GPU (GPGPU) compute platforms like NVIDIA's Compute Unified Device Architecture (CUDA) [1], Open Compute Language (OpenCL) [2] or the Heterogeneous Systems Architecture (HSA) [3] have been made available. While these *do* make accelerators easier to program for, the experience they offer by themselves is still far from seamless. All the platforms mentioned above introduce new languages, specifically for the parts of a program intended for the GPU. They come with new tools like compilers, assemblers or debuggers that deal only with these parts.

Additionally, they require using a platform runtime library to load, compile and run GPU kernels. That means large chunks of application code have to be written just to prepare input data, interface with the runtime and then interpret the output data retrieved from a device. None of that code actually helps with the calculation, it is simply boiler-plate.

Beyond that, this is also rather error-prone. Because calling GPU kernels is not done as a simple function call but via generic runtime APIs, validation of kernels and their arguments cannot happen at compile time. Instead, these errors only surface when the program is already running.

As these deficiencies show, there is a need for better integration of GPGPU platforms with programming languages. We attempt this using the Julia language and the HSA GPU computing platform. In a first step, we simplify using the HSA runtime library from within Julia. Improving on that, we introduce modifications to Julia that obviate the need for external tools and free the user from interacting directly with the runtime.

The remainder of this chapter gives brief introductions to the major constituents of our work, Julia and HSA, outlines the objectives we want to achieve and gives a quick overview of some similar projects for other languages as well as for Julia.

Chapter 2 revisits parts of our software's components and goes into further detail to build up a foundation for later chapters. Chapter 3 goes on to describe our contribution to the software in depth. The last chapter, chapter 4, covers the performance and limitations of our approach. It concludes with an evaluation of the objectives reached and pointers for future development. Finally, appendices A and B contain instructions for building the project and an example of its use, respectively.

1.1. Julia

Figure 1.1.: The Julia language project's logo

The domain of scientific computing has traditionally belonged to venerable languages such as C and FORTRAN. While these provide the best performance, short of handwritten assembly code, they do leave something to be desired where ease of use and developer productivity are concerned. These drawbacks have spurred interest in using higher-level, modern languages for numerical computing, for example Python (with NumPy [4]) or MATLAB [5]. The newer languages have only made a small impact, however, in large part because they give up too much performance compared to the incumbents.

A new language, still in development, aims to bring together the best of both worlds. The Julia programming language [6] is high-level, expressly designed for numerical computing, and has performance comparable to C as its key objective. Julia's syntax is similar to MATLAB's in many respects, like its 1-based array indexing, to make users familiar with that language immediately comfortable. It is a scripting language, i.e. one that can execute a program directly from its source without a distinct compilation step. As such, it can execute scripts entered directly into the Read-Evaluate-Print Loop (REPL) console as well as script files.

Scripting languages usually have difficulties in achieving good performance because of the need to parse and interpret the source code before execution. Julia gets around that by leveraging LLVM's Just-In-Time (JIT) compiler to compile programs to machine code just before executing them. Subsequent executions of the same program fragment don't incur the same compilation overhead. That allows Julia to attain native performance while keeping the flexibility of a scripting language.

Julia is dynamically typed which often implies lower performance because machine code, generated for a method, needs to be able to deal with all possible argument types at runtime. That would prevent optimizations that can only be applied if the argument type is known beforehand. Julia avoids this problem by specialization of functions on their arguments' types. That means each function is compiled once for each set of argument types, allowing the generated code to take advantage of that knowledge. Then, when calling the function, the version of it compiled for the correct sequence of argument types is executed.

Among the most notable features of Julia are macros. These, inspired by LISP's feature of the same name, are source code transformations that are applied to an expression before it is compiled and executed. Macros are implemented as functions that are called by prefixing their name with an @ sign. They get passed the Abstract Syntax Trees (ASTs) of their argument expressions and have the opportunity to modify or extend them. The resulting new AST is what is compiled by Julia. A simple example is shown in listing 1.1 where the `@assert` macro is used to check the validity of an assumption. It works by extending the argument AST with code that checks the assertion and throws an error if necessary. Macros are extremely powerful, they enable us to provide a very concise syntax for calling GPU code, without modifying the language itself.

Listing 1.1.: Example of a Julia macro.

```
macro assert(ex)
    # use expression interpolation to
    # embed ex into an if statement
    return quote
        if !($ex)
            throw(AssertionError($(string(ex))))
        end
    end
end

# call the macro
@assert true == false
```

Besides these novel features, its comprehensive standard library, thriving package ecosystem and focus on scientific computing make Julia an ideal target language for our project.

1.2. Heterogeneous Computing and HSA

The advantages of specialized Single Instruction Multiple Data (SIMD) processors like GPUs for running suitable, i.e. data-parallel, calculations compared with classical Central Processing Units (CPUs) both in speed and in energy efficiency have long been recognized. That, combined with the hard limits to single-thread performance such as the limited clock frequency and the slowdown in Moore's law [7], have led to the development of GPGPU platforms. These are the result of transforming classic GPUs to enable them to carry out general purpose, not just graphics related, calculations.

Now that GPGPUs are becoming commonplace and moving into supercomputers, servers and desktop PCs alike, the need for accompanying tools for programming them is growing.

OpenCL has emerged as one of the most popular platforms, targeting computing hardware from a wide range of vendors. It allows distributing a co-processor program (a kernel) in a form independent of the target hardware that is compiled to machine instructions only at runtime. While, initially, that meant distributing source code written in OpenCL C, with version 1.2 OpenCL introduced a native Intermediate Representation (IR) called OpenCL Standard Portable Intermediate Representation (SPIR) that is close to assembly code but with some higher-level features. Version 2.0, published in November 2013, introduces support for memory sharing between the host CPU and accelerators. Because co-processors traditionally don't have direct access to main memory but instead control memory of their own, that was not necessary before. With the introduction of hardware platforms such as AMD's Accelerated Processing Units (APUs), where CPU and GPU are on the same chip and access the same memory, it increasingly is.

Figure 1.2.: HSA Foundation Logo[1]

Simultaneously, the HSA Foundation, created by AMD and many other device manufacturers, has been working on another platform for systems with CPU — GPU memory coherency, namely HSA. The final specification for version 1.0 of HSA [8] was finished earlier this year, 2015. It shares some concepts with recent OpenCL versions, for example a hardware independent IR, but also introduces new ideas of its own.

One example is **user mode queuing** of computational tasks for the GPU. This improves performance by no longer requiring system calls and their associated, expensive context switches to kernel mode, to enqueue new work. Instead, once a device queue is created, the user mode application can write to it and the device will automatically pick up the new jobs without kernel intervention.

Out of necessity, HSA moves away from traditional memory models that guarantee sequential consistency for data-race-free programs in favor of a newly introduced memory model with **sequential consistency for heterogeneous-race-free** [9] programs. The biggest practical difference for the user is that memory synchronization operations are no longer global by default. Instead, they take a new parameter that determines the scope of memory they act on such as system-wide, processor-wide or only within a group of threads. These scoped synchronization operations can improve application performance because costly global synchronization is often not required.

[1] ™ HSA Foundation, used with permission

AMD is also working on an LLVM target for HSA [10] meaning a compiler built on LLVM can, in principle, generate code for HSA devices. Since Julia is built on LLVM this suggests the possibility of integrating one with the other.

Because of HSA's youth, there are, so far, no Julia packages wrapping the HSA runtime or otherwise integrating it into the language. In combination with its features and the availability of an LLVM backend, this makes HSA an ideal subject for this work.

1.3. Objectives

The objective for this work is to integrate Julia and HSA. Integration in this case means enabling seamless use of HSA from within the language and eliminating the previously mentioned drawbacks of using a GPU compute platform.

In short, our implementation should fulfill the following requirements.

- No external language for GPU programming necessary
- No modifications to the host language
- No additional tools required during build
- Reduce the necessary code for interacting with the runtime
- Minimal loss of performance compared to directly using the GPGPU platform

1.4. Related Works

All compute platform runtime libraries we looked at are either written in C or offer a C interface. Therefore, since virtually all languages support calling a C interface, they have always been usable, in some form or another, from any of these languages.

Deeper integration, in contrast, i.e. the ability to write in one language for both CPU and GPU, is not commonly available. There are, however, projects that achieve it, a few of which we will cover here.

Some modify the host language or its compiler to achieve this goal. C++ AMP [11], for example, re-purposes the `restrict` keyword from C to mark C++ functions for compilation for the GPU. It requires support by the compiler but since its specification is openly available there are already implementations targeting DirectX as well as OpenCL and HSA [12].

Aparapi [13], on the other hand, requires neither modifications to the host language nor to the compiler. It translates Java code to OpenCL C or the HSA Intermediate Language (HSAIL).

Another interesting example is the Dandelion project [14] that leverages the Language INtegrated Query (LINQ) feature of .Net languages like C# or F#. It allows writing data transformation expressions that can then be offloaded to a GPU using CUDA.

There are Julia packages available for OpenCL and CUDA. Both primarily wrap the C API of their respective runtime libraries. Neither of them directly integrates their platform with Julia.

However, in both cases there are experimental versions available that do attempt tighter integration. The OpenCL.jl package repository contains functionality [15] that allows translating Julia functions to OpenCL C.

A recently published, modified version of CUDA.jl [16] and Julia [17] enables compilation of Julia functions to CUDA PTX and running them on compatible hardware. The authors take a very similar approach to the work presented here. Our implementation integrates several modifications to Julia stemming out of that effort.

2. Technical Introduction

In this chapter, we cover the most important parts of the software components our project is built on in more detail. That encompasses Julia's foundation **LLVM**, our chosen GPU platform **HSA** and the bit of Julia itself we need to lay hands on, the **code generator**.

2.1. LLVM

Figure 2.1.: The LLVM project logo

LLVM is the foundation for Julia and in large part responsible for making this work possible. It is an open-source project implementing all the necessary components for a compiler. It uses a very modular architecture which enables a high level of code reuse and flexibility. The most important high-level components of LLVM are the following.

- Language Front End
- Intermediate Representation (IR)
- Optimization and Transformation Passes
- Machine Back End

Of these, the IR is the central part gluing all the others together but also isolating them from each other. This allows swapping out the parts with different implementations and having them still work together because they speak to each other using the IR

2.1.1. Language Front End

The first part of the LLVM pipeline that a program passes through is the frontend. This part of the system is responsible for parsing input files in a particular programming language and generating LLVM IR from that.

In Julia, this is split between a parser implemented in LISP and the code generation logic in C++. The reason for the split is that with Julia, parsing and code generation happen at different times. Julia code is parsed as soon as it is read in, but the IR for it can only be generated when it is about to be executed for the first time. The parser builds an AST, where some parts of the language — that are only syntactic sugar — have already been converted (lowered) to more basic operations. This simplified AST is then passed to the code generator and stored. Later, when a function is called, the code generator uses its AST to create corresponding LLVM IR instructions.

2.1.2. Intermediate Representation

The IR is the most important part of LLVM. It is a pseudo assembly language that uses Single Static Assignment (SSA). The IR is not a traditional assembly language because it still contains higher-level concepts, such as function calls, and not just operations that map directly onto machine instructions.

The IR is structured in a hierarchy of containers beginning with a *Module* at the top that contains *GlobalValue*s such as *Function*s. *Function*s in turn have a signature and a body composed of *BasicBlock*s. Each of these holds a sequence of *Instruction*s. These structures represent the program code in memory and allow it to be manipulated.

2.1.3. Optimization and Transformation Passes

Once the IR has been generated, it is time to apply LLVM passes to it that transform or optimize the code. Passes are essentially functions that walk the IR hierarchy and look at each part in turn, possibly making changes to it. Classical examples for optimization passes are loop unrolling and function inlining.

LLVM is structured as a pipeline of passes that is initially built up and then runs on the input IR. The passes are called on the IR, one after the other, in the sequence they were inserted into the pipeline. Each pass sees only the transformed output of its predecessor. LLVM has base classes for different kinds of passes, depending on at what granularity they need to act on the IR. Module passes, for example are called for each *Module* in the IR, function passes for each *Function* and so on.

2.1.4. Machine Back End

While the pipeline might consist only of IR to IR passes, if that is the desired final output, usually it does not. Instead, the final pass is special in that it takes IR as its input but does not modify the IR.

This pass is the backend or (machine) *Target* pass. Its output is machine code for its particular target platform. LLVM already has many machine target implementations ranging from actual hardware Instruction Set Architectures (ISAs) like x86 or SPARC to virtual instruction sets like NVIDIA Parallel Thread Execution (PTX) or HSA Intermediate Language (HSAIL) and new ones are constantly being developed.

Usually, the target pass writes output in a particular object format like MachO or ELF. The object code can be written directly to a file or into an in-memory buffer which allows it to be used immediately (see section 3.2).

2.1.5. LLVM in Julia

The LLVM pipeline can be invoked by a traditional compiler to produce binaries from source code which is what for example clang does. However, it can also be used as a JIT compiler to generate machine code from IR in memory which is then immediately executed. For this purpose, LLVM contains several JIT frameworks with the most recent ones being MCJIT and its upcoming successor OrcJIT. Julia uses these frameworks to implement its JIT compilation.

2.2. HSA

Because it is our intention to integrate HSA with Julia, we will now give a short introduction into the central objects and functionality in its runtime API. The full developer documentation is available on the HSA Foundation's website at [3, 8, 18, 19].

Similar to OpenCL, HSA provides abstractions and interfaces to distribute calculations between the components of heterogeneous systems and coordinate them. The focus for HSA lies on programming APUs but the plan is to support discrete GPUs or Digital Signal Processors (DSPs) in the future.

2.2.1. Runtime

The first part of the HSA runtime API an application has to interact with, is the platform itself. This includes methods to initialize and shut down the runtime. Other methods query global information about the platform, like its endianness, the pointer bit width and extensions it supports.

Agents

After initialization, the next step is to discover processors available for computation. In HSA, each component of the system that participates in some way in the calculation is called an Agent. Components participate by creating new work, and submitting it for execution. If an Agent can also process work itself, meaning it can run compute kernels, it is called a Kernel Agent.

The runtime provides information about the available agents like their capabilities, hardware characteristics and ISA

Kernels and Executables

In HSA, a program for an agent is called a Kernel. A kernel contains instructions for a single thread of execution with the understanding that when it is finally executed, it will be run in parallel on many tens or hundreds of threads in lockstep. This view is a good fit for the Single Instruction Multiple Threads (SIMT) nature of GPUs.

To illustrate this concept, consider an algorithm for matrix multiplication. The Kernel for this algorithm might consist of the calculation of one single cell in the result matrix from input data. To produce the complete result matrix, the kernel is run once for each output cell. Each run has its own id that is retrieved by the kernel and influences its behavior. The set of indices is defined by a Range which, for HSA, can be of one to three dimensions depending on the needs of the algorithm.

Before a kernel can execute, it needs to be available as an Executable for the destination agent. An executable is another runtime object that encapsulates pieces of machine code for a particular agent's ISA. Just like with a host program, multiple pieces of code might be loaded into an executable. Before it can execute, an executable containing all necessary code objects, has to be frozen. At that point, the contents are linked together and symbol references between code objects are resolved. After that, the executable can be queried for a pointer to the agent code that can then be used to launch the kernel.

Signals

A computation is often not simple enough to be done in a single kernel launch. That means there is a need to coordinate the sequential execution of different kernels that each perform a single step in the algorithm. Even for problems that *can* be handled by executing a single kernel, the host application needs to know, when the kernel has finished running and it is safe to use the result. This is what HSA signals provide. They implement a traditional semaphore, an integer valued shared variable that can be set by one process and waited on by another.

Each launch of a kernel has an associated signal that is decremented once that launch completes. A kernel launch can also be told to wait for a number of prerequisite signals before starting execution.

Queues

The last type of HSA primitives we will mention are queues. These are allocated for a specific kernel agent and can be used to submit new work for it to do. A queue, internally, is just an array used as a ring-buffer and two associated signals. The array stores Architected Queueing Language (AQL) packets with a fixed size of 64 bytes. These are inserted into the queue by one or more agents in the system and consumed by the kernel agent associated with the queue.

The main purpose of AQL queues is to allow work queuing by a user-space process to an accelerator without needing to involve the operating system kernel. Once allocated, the queues can be written to directly by the application and read by the HSA hardware. This works by storing the next write-index in an HSA signal that is part of the queue and is signaled every time a new packet has been written. The hardware then waits on that signal and automatically wakes when new work is available. Once a packet has been removed from the queue, the hardware uses the second associated signal to increment the current read-index which tells the application, that the associated ring-buffer entry has been processed and can be reused.

AQL supports three types of packets. The first and most important is the kernel dispatch packet which contains the information necessary for a kernel agent to run a kernel. Among other things, the kernel arguments, the kernel binary and the index range the kernel is to be run on are part of the packet.

The second packet type is the barrier packet that simply postpones the processing of packets behind it in the queue, until after a set of signals have been triggered. The third packet type is the agent dispatch packet that executes a built-in function on an agent that supports it. This is primarily useful for allowing running kernels to call into the host application for something it cannot do itself, for example dynamic memory allocation.

Memory

One important issue with computation using accelerators is memory management. Discrete GPUs, for example, have their own memory that is separate from main system memory. That accelerator memory is accessed using an address space of its own. This means that any input data has to be copied from main memory to accelerator memory before being accessible to a kernel running there. Any output data computed by a kernel needs to be transferred back to main memory. Even with Integrated Graphics Processors (IGPs), where host memory and accelerator memory regions *both* are in main memory,

the copies are generally still necessary. That iw the case, because the regions are not accessible to both of the processor's components.

HSA, in contrast, requires an implementation to supply a unified memory address space accessible from all agents in the system. That allows an application to share data with kernels running on accelerators simply by passing them a pointer. Combined with AQL user-mode queueing, this eliminates a large part of the overhead previously associated with heterogeneous computation.

2.2.2. Kernel Compilation

To be executed on a particular agent, an HSA Kernel needs to be available as a binary compiled for its ISA. There are two ways for an application to obtain this kernel binary, online or offline finalization.

HSAIL and BRIG

In both cases, the necessary input to the finalization step is the source code of the kernel program. The source code language used by HSA is an assembly language called HSAIL. It is rather low-level and cumbersome to write by hand. Instead of as a development language, HSAIL is intended as the output format for high-level compilers. To that end, the *HSA Programmer's Reference Manual* defines both the human-readable HSAIL text format and the more compact HSAIL binary format (BRIG).

So far, there are few high-level compilers that can target HSAIL/BRIG. There is, however, an implementation of the HSAIL Target for LLVM [10]. This, in principle, allows any compiler that can target LLVM IR to compile down to HSAIL. That is the approach taken by the *HSA OpenCL Offline Compiler* script provided by the HSA Foundation which automates compiling an OpenCL kernel to HSAIL. We use the same mechanism to get the HSAIL for a Julia function.

Finalization

To go from an HSAIL kernel to the final binary an accelerator can execute, it needs to be finalized. This can either happen at compile- or install-time (offline) or at run-time (online). The more flexible approach, because it does not depend on knowing the target ISA beforehand, is run-time finalization.

To enable this, the *HSA Runtime Programmer's Reference Manual* defines a runtime extension for finalization that allows an application to finalize a program from input BRIG modules. A program, in this case is another HSA runtime object similar to the executable with the difference that where an executable holds machine code, the program contains source code. After a program object is created and the necessary HSAIL binary

format (BRIG) modules are added to it, it is then finalized, yielding a code object fit for consumption by an executable.

Since finalization relies on an extension library, an HSA application cannot assume it to be supported. The HSA API allows querying the runtime for the presence of a particular extension. After support is confirmed, another API call retrieves a table of function pointers that allow calling the methods that make up the finalizer extension. An example of its use is shown in appendix B.

2.3. Julia Code Generator

This section outlines the inner workings of the Julia code generator, concentrating on those parts that we modified in the course of this work. The code generator is the part of the Julia backend, that takes the AST for a function, when it is called, and generates LLVM IR for it. The IR is then handed off to the JIT compiler.

The code generation phase comprises the following major steps:

1. Function Creation
2. Variable Allocation
3. Function Body
4. Machine Code Generation

Function Creation

When the code generator is invoked, it first unpacks the AST if it is stored in compressed form and then collects general information on variables used inside the function.

Then, the code generator has to create LLVM objects for the function. When using MCJIT as the JIT compiler, that means a new empty LLVM *Function* as well as a new *Module* to hold it. A dedicated *Module* for each *Function* is necessary, because MCJIT compilation works at *Module* granularity.

Since Julia supports function specialization on argument types, the code generation is invoked once for each newly encountered set of argument types. The new *Function* can then be created with knowledge of the argument types being passed. In particular, its signature can reflect the expected number and type of the arguments. Because, a *Function*'s signature is immutable after creation, this is the last chance to change it without replacing the whole *Function*.

In case specialization is not possible, the generated *Function* uses a generic Julia signature that takes two arguments. The first is a pointer to an array of arguments of the generic Julia object type `jl_value_t*`, the second is the length of the array. Possible reasons

for not being able to specialize a function's signature are the capture of outside variables or a variable number of arguments.

After these objects are created, the necessary debug information is added and then we move on to the function body.

Variable Allocation

Julia is a garbage-collected language. In order for the Garbage Collector (GC) to work, any function that copies a reference to a heap-allocated (and therefore garbage-collected) object needs to register the new reference with the GC. That is achieved by allocating a GC-frame on the stack and adding it to a global list of frames. All GC'd references are then stored in slots in this frame and can be found there by the Garbage Collector.

Allocation of the GC-frame is the responsibility of the function preamble. During generation of the preamble, the number of necessary slots is determined. Any value that can be held in its entirety in a register or on the stack does not need to be garbage-collected but those that do have to be put onto the heap, a process called Boxing, get a slot in the GC-frame. As a performance optimization, if the function uses no references to boxed values, no GC-frame is allocated and the preamble is empty.

Function Body

Now, the actual function body is generated, each bit of Julia code is transformed into its corresponding LLVM IR representation. Most of that is not important to us in detail.

One exception is the `llvmcall` intrinsic. The `llvmcall` can be used to embed any valid LLVM IR snippet into the body of a function. This is useful in cases where we want to access functionality that LLVM knows about but that does not have a corresponding Julia function. For example to call LLVM intrinsics. Section 3.2.4 shows how we use this intrinsic to enable writing HSA kernels in Julia.

Machine Code Generation

Finally, after the LLVM *Function* is fully generated, its containing module is normally passed to MCJIT for compilation to machine code. Since, at that point, the function is fully translated to IR, we can now apply custom LLVM passes and circumvent MCJIT, instead applying our own compilation logic.

3. Implementation

This chapter describes the core of our work, our implementation of tighter integration between Julia and HSA. Figure 3.1 shows, conceptually, the structure of our solution. The foundation we build on is provided by Julia and HSA. Our contribution is split into two parts, the **HSA.jl** package and the **HSAIL code generator**.

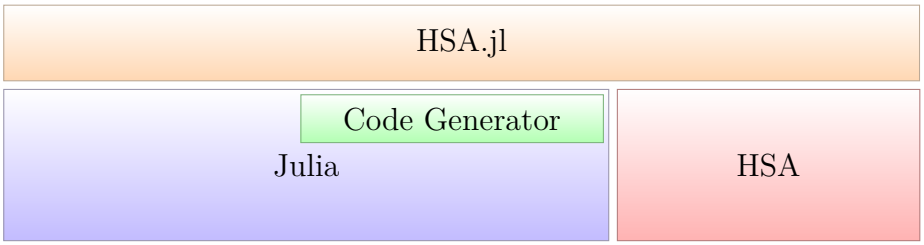

Figure 3.1.: Project Architecture

The HSA.jl package is intended to be useful on its own. Therefore, the majority of its functionality does not rely on the presence of the code generator. That includes the auto-generated HSA runtime bindings as well as our custom additions to them that make HSA more convenient to use from Julia. We cover this part of the system first, in section 3.1.

Next, we describe the modifications to core Julia that make up the HSAIL code generator. It implements the necessary infrastructure to convert a Julia function to BRIG code.

The final section of this chapter ties it all together by detailing the additional functionality that is enabled in HSA.jl when the HSAIL code generator is present. The union of those two parts is what enables the deepest integration and the most radical simplification of using HSA, the `@hsa` macro. It allows running a Julia kernel function using HSA in literally a single line of code.

3.1. HSA.jl

The majority of the HSA.jl package is concerned with making the HSA runtime API conveniently accessible from Julia. That part, then, is usable on any HSA system and does not require our modified Julia version. In this section, we go over the major features of HSA.jl and its improvements over writing your own interface code. For a complete usage example, see appendix B.

3.1.1. Generated Runtime Wrappers

Julia supports interfacing with C and FORTRAN code [21] via the `ccall` intrinsic. While this makes using the HSA runtime API possible, it is still very verbose. For example, the definition of any non-trivial data type used by the C interface needs to be replicated in Julia. A C function being called from Julia also needs to be identified using its full signature as in listing 3.1.

Listing 3.1.: Example of a `ccall` to an API function.

```
ccall(
    (:hsa_system_get_info,libhsa), # function and library
    hsa_status_t,                  # return type
    (hsa_system_info_t,Ptr{Void}), # argument types
    attribute,value)               # actual arguments
```

The necessary boiler-plate code can be auto-generated to some extent. The *Clang.jl* [22] package makes it possible to use the clang C++ compiler's API to parse a C header. From the parsed interface definition, it can then generate `ccall` wrapper functions as well as constant and data-type definitions in Julia.

That solves the problem of having to copy the interface by hand. However, the generated code does not simplify creating the correct arguments for called functions. Pointer arguments, for instance, need special handling to work correctly. In the case of input pointer arguments, the caller needs to ensure that the argument object can be converted to the desired pointer type. For output pointer arguments, memory in the form of a `Ref{T}` or `Array{T}` has to be explicitly allocated and the values extracted after the call completes.

We use hooks provided by the *Clang.jl* generator to inject custom generation logic. For example, we need to parse three header files that contain the HSA definitions for the HSA runtime interface and for the two extensions, images and finalization. Because the extension headers both include the main header, the generator processes its contents three times, by default. That leads to many duplicate definitions and makes our package slow to load. To prevent that, we look at each new symbol being processed and only generate code for it the first time we encounter it. Some symbols, like structure names and `#define`s are filtered out completely because they are merely artifacts of the way the generator works and should not be part of our code.

The generated Julia code, initially, is an AST in the form of an array of Julia expression objects. After the generator is done processing the headers and before the code is written to a file, we go over it, make modifications and generate custom wrapper methods.

3.1.2. Custom Wrappers

To arrive at a simpler and more idiomatic Julia interface, HSA.jl improves the generated code and complements it with additional handwritten types and methods. This section highlights the most important of these improvements.

Wrapper Types

We replace references to certain HSA interface types with new types using the Julia naming convention, for example `hsa_queue_t` with `Queue` or `hsa_packet_header_t` with `PacketHeader`. In some cases, this is purely cosmetic, but in others it is not.

The replacements for AQL packet types, for example, define very concise constructors that make some fields available as optional parameters with sensible default values. That allows creating a fully configured `KernelDispatchPacket` in a single call (listing 3.2)

Listing 3.2.: Example of an AQL packet constructor.

```
p = KernelDispatchPacket(
        kernel_object,
        (grid_size_x, grid_size_y),
        completion_signal = s)
```

The new types also simplify using special packet fields like the `header` or the `setup` field for `KernelDispatchPacket`. Both are packed fields, meaning they contain several sub-fields that are compressed into a two byte value. Julia does not support union types or fields smaller than one byte. That means that to access the sub-fields we need to do bit-shifts and masking. In the replacement types, these sub-fields have been promoted to full width fields which makes working with them easier. The logic for converting these Julia types to the correct in-memory representation is implemented in conversion methods. A package user never has to deal with this directly.

The `Queue` type on the other hand adds convenience functions that allow writing packets to or reading them from the queue using array-index syntax. These also take care of mapping the monotonically increasing write index to an offset into the ring-buffer. As shown in appendix B, this is usually the responsibility of the programmer.

Property Getters

The data types used in the HSA API are mostly just opaque handles. Reading their properties works by passing the handles to a corresponding getter function. One parameter for the getter determines which property is read and what is returned via an output

parameter of pointer type. Some properties, that are strings, have a second property that returns the length of the string to be returned. This pattern, which is common for a C API, makes using the getters from Julia code difficult.

To work around that, HSA.jl generates one getter function for each property. These getters pass the value of the property as their return value and handle conversion to the corresponding Julia type. A string-property/length-property pair, for example, is collapsed into one function that returns a native Julia string object (listing 3.3).

Listing 3.3.: Example of an auto-generated property getter:
Retrieving the name of a device ISA (comments added manually)

```
function isa_info_name(isa)
    # get the name's length and allocate memory for it
    len = isa_info_name_length(isa)
    value = Array(UInt8,len)

    # write the name into our buffer
    err = ccall(
        (:hsa_isa_get_info,libhsa),
        hsa_status_t,
        (hsa_isa_t,hsa_isa_info_t,UInt32,Ptr{Void}),
        isa,HSA_ISA_INFO_NAME,Base.zero(UInt32),value)
    test_status(err)

    # convert the character array to a string
    value = strip(ascii(value),'\0')
    # implicitly return value
end
```

Iterate Callbacks

Another pattern used in several places in the API is an iteration callback. For example, the enumeration of all known agents available to the application works by passing a pointer to a callback function to `hsa_iterate_agents(...)`. That callback is then invoked by the runtime for each agent in turn until it returns `HSA_STATUS_INFO_BREAK`. Getting and using a pointer to a Julia function for use with a C API requires some special handling. For that reason, HSA.jl provides wrappers for these iteration functions that take regular Julia functions as a callback. There is also functionality for the most common use cases like getting all agents (`all_agents()`) so no callback is necessary.

Signal Accessors

The runtime API has a lot of functions for setting or getting the value of signals atomically. Most are available in several variants that only vary in the memory order they use. There are, for example, two "store" implementations: `hsa_signal_store_relaxed` and `hsa_signal_store_release`. The Julia API maps these into one function that takes memory order as an optional argument. The default memory order is always the most conservative of the available options (AcqRel, Release or Aquire, in that order).

We also add new methods to the well known `unsafe_store!` and `unsafe_load` functions in the Julia base module. These are the customary way of interacting with values unsafely, i.e. through pointers.

Julia Code Conventions

HSA.jl tries to make using HSA feel more natural for someone used to Julia's idioms and conventions. We want to mention two specific instances of that here, error handling and naming conventions.

Library interface methods in C, for performance reasons or just because Structured Exception Handling (SEH) is not available, commonly signal errors through function return values. In Julia on the other hand, errors are usually surfaced via exceptions. To match expectations, all generated wrapper functions capture the `hsa_status_t` return value for API calls and raise an `HSAException` containing the error code as well as the corresponding friendly error string.

HSA.jl also tries to match Julia naming conventions. It adds aliases for type names, constants and enumeration values that use camel-casing instead of the caps_with_underscore C symbol names. Also, since all HSA.jl code is put into the `HSA` Julia module, the `hsa_` name prefix is redundant and, in most cases, is removed during generation.

Simplified Initialization

The first steps for running an HSA kernel are always very similar:

1. Initialize the HSA runtime
2. Find an agent to run the kernel
3. Create a queue and a completion signal

To simplify this process in cases where no special configuration is required, HSA.jl offers convenience methods that automate initialization (listing 3.4).

In case the default constructed objects are not what is desired, they can be replaced using `HSA.set_defaults(...)`. This is useful mainly, because some parts of HSA.jl

Listing 3.4.: Using the simplified runtime initialization.

```
cfg = HSA.init_managed()

# The first GPU agent in the system
cfg.agent
# A newly created single threaded queue
# of maximum size
cfg.queue

# ... use HSA

HSA.shutdown_managed(cfg)
```

automatically use the objects from the managed configuration to avoid passing them as arguments. The most important example of that is the `@hsa` macro (see section 3.3.2)

Automatic Resource Destruction

Many of the HSA objects have to be explicitly created and later destroyed by calling into the API. In Julia, a garbage-collected language, one does not usually have to deal with explicit object lifetime management. Normally, some time after the last reference to an object is released, the Garbage Collector will handle its destruction.

When interacting with objects created outside of Julia, that is no longer the case. Instead, the necessary destruction logic has to be invoked manually. To recover the ease of use of garbage-collection, Julia offers the `finalizer(object, func)` mechanism to bind a custom destructor to a Julia object. This destructor can either be called explicitly using `finalize(object)` or is called automatically when the GC reclaims the object it is assigned to.

HSA.jl uses this mechanism to tie destruction of HSA resources to the lifetime of their corresponding Julia wrapper objects. A `Signal` instance, for example, automatically calls `hsa_signal_destroy(...)` when it is finalized. The same method is used to associate the initialization and shutting down of the HSA runtime to an instance of the `Runtime` wrapper type.

3.2. HSAIL Support in the Julia Code Generator

This section describes in detail, how we extend the Julia code generator to enable compilation of Julia functions to HSA BRIG. We start with the goal, BRIG and go backwards, step by step, until we reach the Julia code. This way, the reason for each transformation becomes clear by looking at the requirements of its successor.

Figure 3.2 provides an overview of these transformations. The colors reflect, which part of the system in fig. 3.1 handles that particular step. Purple for base Julia, green for the HSAIL code generator and red for HSA.

To illustrate the intermediate forms the code takes on, going through the pipeline, we use a simple Julia kernel for copying a vector, shown in listing 3.6a.

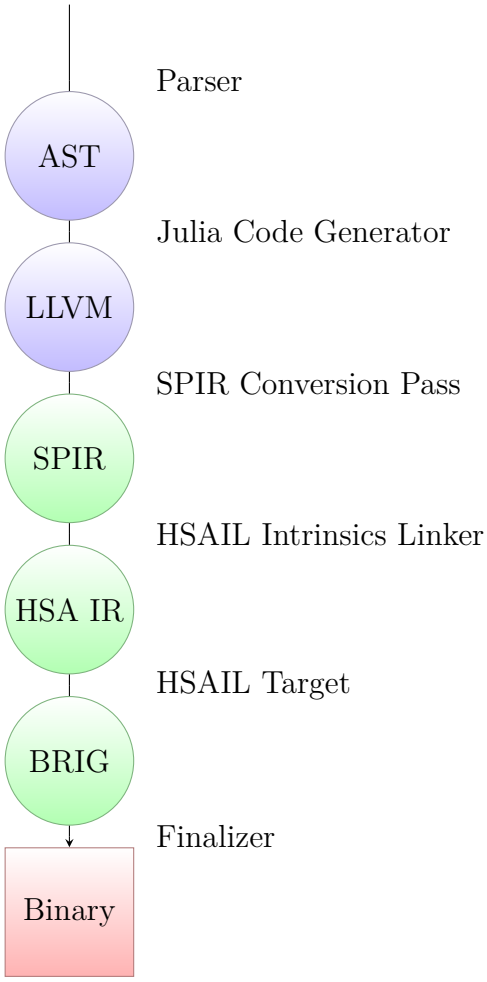

Figure 3.2.: Compiling a Julia kernel to a device binary

3.2.1. The HSAIL Target

The modularity of Julia's foundation, LLVM, allows the connection of new backends to existing frontends or vice versa (see also section 2.1). That, combined with the fact that the HSA Foundation has been developing an LLVM backend for HSAIL, is what makes our implementation possible.

A major difficulty for the HSAIL target stems from the fact, that many HSAIL instructions do not have corresponding IR instructions to map from. Examples for these are the indexing (workitemabsid_u32 etc.) or synchronization instructions (barrier etc.).

The usual approach, and that taken by the HSAIL backend, is to introduce well known *intrinsic* functions like `@__hsail_get_global_id(i32)`. These can be called by the IR program and will then be recognized and lowered to the corresponding HSAIL instruction. Listings 3.6c and 3.6d contain LLVM IR using the intrinsics and the HSAIL assembly generated from it, respectively.

3.2.2. SPIR Intrinsics

The same principle is used by OpenCL's SPIR, which is essentially just an LLVM IR with certain extensions and restrictions. For example, the SPIR function corresponding to the HSAIL intrinsic above is `@_Z13get_global_idj(i32)`.

Instead of generating calls to HSAIL intrinsics, we use the SPIR intrinsic functions. The main reason for this is that SPIR has a specification [23], while the HSAIL intrinsics do not. There is also some interest in generating SPIR from Julia [24] and using that as an IR could simplify future developments in that direction (see section 4.4).

Finally, the OpenCL Offline Compiler (CLOC) script [25] comes with LLVM bit-code files that implement the SPIR intrinsics in terms of HSAIL intrinsics. These bit-code files are then linked into the IR module containing the function being generated. That effectively maps one set of intrinsics to the other.

3.2.3. From LLVM IR to OpenCL SPIR

To convert the plain LLVM IR emitted by Julia to valid OpenCL SPIR, we have to make the following changes to it:

- Add OpenCL Kernel Metadata
- Set the correct Calling Convention
- Add Address Spaces to pointer types

To that end, we implement an LLVM module pass that is run on the LLVM IR emitted by Julia.

Kernel Metadata

The first change to the IR is to add OpenCL metadata. SPIR expects named module-level metadata under the key "opencl.kernels" containing an entry for each kernel function in the *Module*. Each entry contains several pieces of information.

- A pointer to the kernel itself
- Information on each kernel argument
  - Name
  - OpenCL C type, base type and qualifiers
  - Address Space definition

From the perspective of using the SPIR as input to the HSAIL Target, the only required piece of information is the list of kernels, so that the Target can find them. Nevertheless, we generate the full SPIR metadata by, for example, mapping the LLVM argument types back to OpenCL C type strings.

Calling Convention

Any SPIR kernel or non-kernel function must have the corresponding calling convention, `SPIR_KERNEL` or `SPIR_FUNC` respectively. Because the Julia-generated functions use the default calling convention, this is set by us as part of the conversion.

Pointer Address Spaces

For programs that need to access more than one kind of memory, LLVM has support for Address Spaces (ASs). That becomes necessary, when not all resources to be accessed are mapped into the same range of virtual addresses contrary to what is usually the case when running on the host CPU.

GPUs often come with their own memory that is separate from main memory and not under the control of the CPU. In OpenCL SPIR terminology, this is *global* memory, because it can be accessed by all Compute Units in the GPU.

In addition, GPUs normally have smaller, faster cache memory to temporarily store frequently accessed data as CPUs do. Different from the CPU cache, however, the GPU's cache does not operate automatically but is controlled directly by the GPU program. These caches can further be differentiated into two types, *local* (a.k.a. work-group) memory and *constant* memory that differ in how they can be accessed by the GPU program.

Each of these types of memory is not addressable by IR instructions using normal main memory pointers but has its own Address Space. In LLVM, any pointer type has an associated Address Space which defaults to `0`, the *generic* AS.

Memory in HSA. One of the main features of HSA is the fact that the CPU and the GPU share the same memory (see section 2.2). That means that, in HSA, the global AS and main memory are actually the same. Nevertheless, in SPIR, pointers that access main memory have to have the correct AS value for the global Address Space, `1`.

Address Spaces in Julia. Because Julia was developed to run on host CPUs exclusively, it uses the generic Address Space for pointer types throughout the code generation logic. That also leads to assumptions in the code that break down when using pointer types in non-generic Address Spaces. There are, for example, several places in Julia's code generator, where the type of a *Value* is tested for equality with a set of well known LLVM pointer types that, naturally, reside in AS `0`. That means, Julia code generation can fail if *Value*s with non-zero AS are encountered while the code is still being emitted.

Casts that preserve Address Spaces. One possible issue that can arise during generation is that the Address Space information is lost when a bit-cast instruction (*BitCastInst*) is emitted. Bit-casts are the equivalent of C-style type-casts in LLVM IR. They take an input *Value* and a desired output type and return a *Value* of that type.

When casting between pointer types, Julia always uses types in the generic AS as the destination type. For that reason, the generated instruction silently discards the original AS in older versions of LLVM and leads to an assertion in newer releases that no longer allow implicit casts between ASs.

This particular issue is fixed in a Pull Request (PR) [26] that has, so far, not been accepted into Julia master. Even with this fix, there are points all over the code generator where an unexpected AS leads to failure. Julia support for Address Spaces is planned but has yet to materialize.

Adding Address Spaces after code generation. Because of these issues, we do not introduce Address Spaces immediately when a new *Function* is created at the start of code generation. Instead, the LLVM pass later changes the Address Spaces where necessary and then propagates the changed types through the code.

Currently, the only place we *do* introduce non-generic ASs is in the kernel argument types. There, the presumption is that any argument of pointer type must reside in the global AS and its type is changed accordingly.

Many object properties in LLVM IR are immutable. That includes the signature and the argument types of a *Function*. This means that changing the Address Space of a pointer

argument entails re-creating the *Function* with an updated signature and copying over the function body from the old one. Cloning the body also allows us to look at each use of a *Value* whose type has changed and consider recursively propagating the change to users of that *Value*.

We support two types of IR instructions that need to reflect the AS of their inputs in their output type.

BitCastInst
: Used to cast one value to another type. This instruction needs to propagate the AS of the input type, if the output type is also a pointer type.

GetElementPtrInst
: Used to derive the pointer to an element of a composite type from a pointer to that type.

This method fails to catch some instances of missing Address Spaces. For example, a kernel that performs a pointer-int-pointer cast sequence will escape the transformation and not be compiled correctly. However, this enables execution of many kernels for now and bridges the time until Julia adds full support of pointer Address Spaces in the code generator.

3.2.4. Builtin Functions

Converting between Julia and IR, we again encounter a problem we had moving from IR to HSAIL. Specifically, we need something in Julia code that we can map to the SPIR intrinsics. That is the purpose of builtins which are special functions implemented by HSA.jl and only usable by Julia kernels.

To map these functions to SPIR intrinsics, we make use of the LLVM support in Julia [27], the `llvmcall`. In unmodified Julia 0.4, `llvmcall` can only be used to call functions that are already known to the module the call is being emitted into.

The module created for a kernel function does not contain any declarations for SPIR intrinsics, initially. Before they can be used, we therefore need to declare them. This is enabled by a patch to `llvmcall` [28] that came out of the effort to integrate CUDA into Julia. With that applied, we can use `llvmcall` as in listing 3.5, simultaneously declaring and calling an intrinsic function.

Listing 3.5.: Builtin function, implemented using llvmcall.

```
function get_global_id(dim)
    return llvmcall(
    (
    # declaration
    """declare spir_func i64 @_Z13get_global_idj(i32)""",
    # llvm ir function body
    """%res = tail call spir_func i64 @_Z13get_global_idj(%0)
       ret i64 %res"""
    ),
    Int64,            # return type
    Tuple{UInt32},    # argument types
    dim)              # actual arguments
end
```

Using the extended `llvmcall`, HSA.jl provides several built-in functions in the `Builtins` submodule, that can be used by Julia kernel functions.

Identification of Kernel Functions

The SPIR intrinsics only make sense when used inside kernel code. When a function is compiled for the host CPU, there is no concept of a work-item, work-groups or anything related to that like work-item ids and work-group synchronization. Because of that, trying to use the intrinsics in a function that will be compiled for the host will lead to an error.

That means, we need to signal the code generator which compile target a function is intended for. Julia already contains a mechanism for passing information about a function to the code generator. It is used by the implementation of, for example, the `@inline` and `@noinline` macros.

This facility, which is called meta, works by passing the annotation as well as the function definition to a macro. The macro takes the AST of the function and uses `pushmeta(...)` to store the annotation in a certain location inside the AST. When the AST is later evaluated by the code generator, it can extract the value and use it to influence the generation process.

Based on meta, the PR for CUDA support in Julia [17] adds a new macro `@target` that is used to tag a function with its intended compile target. We adopt that same macro in this project and use it to implement our own target macro `@hsa_kernel`. Section 3.3.1 describes in greater detail, what this macro does and how it works.

3.2.5. Development Obstacles

When our project got underway, the HSA specification was still being finalized and the LLVM HSAIL target was under heavy development. Significant effort was spent integrating the HSAIL target's Makefile build with that of Julia and, when support for that was dropped, moving to the CMake build.

Another part of that work was keeping up with changes in all the components, with Julia as well as the HSAIL target moving to newer LLVM builds, while also making progress towards our goal.

If we were to start our project tomorrow, much of that effort would not be necessary. Julia now has support for building LLVM via CMake and the HSAIL target offers *stable* branches that are slated to be merged into upstream LLVM.

This clearly shows that there is a cost to working with software that is still being developed. However, only by accepting this cost can we be part of this development and push the state of research forward.

3.3. Julia Kernel Execution

Building on the code generator support, HSA.jl contains functionality that makes it easy to specify and run Julia functions as HSA kernels. This allows significant reductions in the code necessary to build and launch an HSA kernel. A concrete example of that is shown in appendix B.2.

There are two major parts to this that are enabled when HSA.jl is run on a Julia executable that supports the HSAIL code generator.

- Device Target Support via `@hsa_kernel`
- Managed Kernel Execution via `@hsa`

3.3.1. Device Target Support

Marking a Julia function with the macro `@hsa_kernel` as shown in listing 3.6a, prepares it to be compiled to BRIG.

Internally, the macro tags the function with `@target hsail` so that the code generator knows how to treat it. It also enables the use of built-in functions like `get_global_id(...)` in the function body.

These live in the module `HSA.Builtins`. When the macro is executed, the AST of the function definition is passed to it. It then goes through all the function calls in the AST and sees if they match a built-in function.

The successive phases a Julia kernel goes through on its way to HSAIL

Listing 3.6a.: Julia implementation of the vcopy kernel.

```
@hsa_kernel function vcopy(a,b)
    i = get_global_id(UInt32(0)) + 1
    a[i] = b[i]
    return nothing
end
```

Listing 3.6b.: LLVM IR of the vcopy kernel, using SPIR intrinsics.

```
define spir_kernel void @vcopy(i64 addrspace(1)*, i64
↪ addrspace(1)*)
{
    top:
    %res.i = call spir_func i64 @_Z13get_global_idj(i32 0)
    %2 = getelementptr i64, i64 addrspace(1)* %1, i64 %res.i
    %3 = load i64, i64 addrspace(1)* %2, align 1
    %4 = getelementptr i64, i64 addrspace(1)* %0, i64 %res.i
    store i64 %3, i64 addrspace(1)* %4, align 1
    ret void
}
```

Listing 3.6c.: LLVM IR of the vcopy kernel, using HSAIL intrinsics.

```
define spir_kernel void @vcopy(i64 addrspace(1)*, i64
↪ addrspace(1)*)
{
    top:
    %2 = call spir_func i32 @__hsail_get_global_id(i32 0) #0
    %3 = zext i32 %2 to i64
    %4 = getelementptr i64, i64 addrspace(1)* %1, i64 %3
    %5 = load i64, i64 addrspace(1)* %4, align 1
    %6 = getelementptr i64, i64 addrspace(1)* %0, i64 %3
    store i64 %5, i64 addrspace(1)* %6, align 1
    ret void
}
```

Listing 3.6d.: Generated HSAIL assembly code for the vcopy kernel.

```
module &__llvm_hsail_module:1:0:$full:$large:$near;

prog kernel &vcopy(
kernarg_u64 %__arg_p0,
kernarg_u64 %__arg_p1)
{
    // BB#0:                                // %top
    workitemabsid_u32       $s0, 0;
    cvt_u64_u32     $d0, $s0;
    shl_u64 $d0, $d0, 3;
    ld_kernarg_align(8)_width(all)_u64      $d1, [%__arg_p0];
    add_u64 $d1, $d1, $d0;
    ld_kernarg_align(8)_width(all)_u64      $d2, [%__arg_p1];
    add_u64 $d0, $d2, $d0;
    ld_global_u64   $d0, [$d0];
    st_global_u64   $d0, [$d1];
    ret;
};
```

If so, these calls are modified to reference the correct builtin function in the `Builtins` module. This way, there is no need to add a using statement for that module to every file that wants to define an HSA kernel. At the same time, the built-in functions don't have to be exported which would lead to them being available everywhere, not just inside kernels.

3.3.2. Managed Kernel Execution

Given the simplified initialization introduced in section 3.1.2 and the compilation of Julia functions to BRIG, we implement another macro `@hsa`. This macro takes a simple function call to a kernel function annotated with an expression that describes the grid for the kernel dispatch, as shown in listing 3.7. From that, it generates all the boiler-plate code that is necessary to prepare and launch the kernel.

```
        using HSA

        @hsa (arows, arows) mmul2d(a,b,result,acols)
```

Listing 3.7.: Automatic kernel execution using `@hsa`.

The whole process consists of the following steps.

1. Initialize the HSA runtime objects
2. Prepare the kernel arguments
3. Build the kernel
4. Allocate argument memory
5. Setup the kernel dispatch packet
6. Dispatch the kernel
7. Clean up

The remainder of this section will explain each of them in turn and how they are handled by `@hsa`.

Initialize the HSA runtime objects

The first part of this is the initialization of the HSA runtime itself. Next, a reference to the destination agent has to be acquired and an AQL queue and signal constructed. This is done using the simplified initialization mechanism which allows the user to customize the objects and parameters to be used. It also enables reusing the runtime objects across multiple `@hsa` calls.

Prepare the kernel arguments

The HSAIL code generator only supports a few well known argument types (see also section 4.2). To minimize the need for manual conversion of arguments, `@hsa` automatically recognizes some argument types and converts them to legal kernel arguments.

Currently, that only happens for arguments of array type (subclass of `AbstractArray`). These are converted to their raw pointer equivalent. Additionally, we use the length of the array to register the underlying chunk of memory with the HSA runtime. Registration is necessary for memory that will be accessed from an HSA agent, so that the runtime can guarantee a consistent view of the memory to all parts of the system.

Build the kernel

After the argument conversions, the final argument types for the kernel are known. Using the argument types, we can now retrieve the correct kernel binary.

A Julia function whose arguments do not have explicit type annotations can, in principle, be called with any combination of argument types. That allows us to, for example, write a matrix multiplication kernel in Julia once and reuse it for matrices of different types.

While Julia can be generic, the generated assembly is specific to particular types. That means we have to recompile the kernel for each set of argument types. Since this is an expensive process, we would like to avoid it if we have compiled the same kernel with the same argument types before.

We achieve this, by caching the generated BRIG after compilation. When we are then asked to run the same kernel again, we check to see if we already have generated code for it and, if so, reuse it.

The second phase of compilation is the *Finalization*. Here, HSAIL assembly code stored as BRIG is converted to the final device binary containing machine code. As such, the binary it is not only specific to the kernel and argument types, but also to the ISA of the target device.

To avoid re-finalizations where possible, we use a second level of caching which stores the finalized binary for each BRIG kernel and each ISA. In the most common case of having to run the same kernel with the same argument types on the same agent multiple times, we can thus save the time for compilation on all but the first execution.

Allocate argument memory

After the kernel is finalized, we can query it for some information. Of most immediate interest is how much argument memory it requires.

With that information, we then have to allocate the requested amount of memory. This is not quite as easy as calling `malloc(...)` because we have to consider a few details.

First, memory allocation in HSA has to go through the runtime analogue to `malloc(...)` which is `HSA.memory_allocate(...)`

Second, this function not only needs to know how much memory to allocate, but also where to allocate it. That is, in which memory region. HSA allows an agent to have multiple associated memory regions. Each region can have different access characteristics and, in particular, may or may not be able to hold kernel arguments. So, we have to enumerate the available regions on the target agent, find one that can accept kernel arguments and then actually perform the allocation.

Finally, we have to copy the arguments into the newly allocated chunk of memory.

Setup the kernel dispatch packet

Now that we have all the necessary pieces to dispatch the kernel, we need to put them into the corresponding fields of a kernel dispatch packet. That includes the grid range, kernel object, argument memory, completion signal and memory segment sizes.

Dispatch the kernel

Once the dispatch packet is prepared, we can dispatch the kernel. To do that, we reserve a new slot in the AQL queue. Then, we write the packet into this slot and "ring" the doorbell signal on the queue. That causes the agent to take notice and start executing the kernel. Now we have to wait on the completion signal until we receive notification that the kernel call has finished.

Clean up

After the kernel has executed all that is left to do is to clean up. That mainly consists of handling custom argument cleanup. For now, the only argument type that needs custom cleanup are arrays. These have been registered with the HSA runtime before running the kernel and now need to be de-registered.

The other runtime objects are not destroyed automatically, because we can potentially reuse them on a later kernel dispatch.

4. Results

In this final chapter, we take a look at our work and how it measures up to the goals we set for ourselves in the beginning.

To that end, we talk about the performance implications of using HSA.jl and give some examples for the overhead that should be expected. Then, we go over the biggest limitations of our project and, in light of that, evaluate the success or failure of our work.

We close by discussing some ideas for improving upon the current state of the software and where we would like to take it in the future.

4.1. Performance

The purpose of our project is not to build a new compute platform, but to make using an existing one simpler. That also means that, once a kernel has passed our wrapping layer, it will be processed like a job submitted via any other language and therefore, its performance will be the same. Consequently, there is little point in measuring the performance of a wide variety of the usual test kernels. What *is* interesting to measure, however, is the performance overhead incurred by using our various mechanisms for launching a kernel compared to the gold standard, the performance of a C application.

4.1.1. Methodology

We have access to two AMD based systems with HSA support, one with a Kaveri processor and the other using a Carrizo chip. However, the performance of the Carrizo machine is much worse than the Kaveri based one and it is currently so unreliable that it cannot be used for tests. The following measurements were obtained on a desktop computer with the AMD Kaveri A10-7800 3.5 GHz processor and 7 GiB of RAM running a 64-bit, Version 4.0, Ubuntu Linux kernel.

For measuring the overhead, we chose two different implementations of Julia compatible, that is column-major, matrix multiplication $C = A \cdot B$. The two variants differ only in the indexing used with one, `mmul`, running on a linear grid and the other, `mmul2d`, on a two-dimensional grid. Both are implemented in Julia as well as in OpenCL C. The full kernel source code can be found in appendix C.

Table 4.1 shows the results of running each of the two kernels on random input data of various sizes in a number of ways. The input matrices hold 64 bit floating point values. The times given were obtained by running each test 100 times and calculating the average execution time from the total. The script used for the Julia tests is part of the samples that come with HSA.jl [29], the C test application has its own repository [30].

The columns for the different kernel launch procedures are designated by a combination of the BRIG source (OCL or JL) and launch method (C, JL or ATHSA).

The BRIG source can be one of the following, depending on how we obtain the BRIG from the kernel source code.

OCL By Compiling an OpenCL kernel using CLOC

JL By Compiling a Julia kernel using HSA.jl

The launch method determines how the BRIG kernel is run.

C From within a C application.

Our C application performs the necessary setup and then dispatches the kernel under test 100 times, each time waiting for the kernel to complete before the next is launched. The total execution time is measured using `clock_gettime(...)` and only includes the actual launch, not the preparation before or cleanup after it. To be exact, the following steps are included in the measurement:

- Setting the completion signal.
- Building the dispatch packet.
- Dispatching the kernel by storing the packet header and queue write index and then ringing the doorbell signal.
- Waiting for the kernel to complete.

JL From Julia.

This test performs the exact same steps as the one from C to run a BRIG kernel. The only difference is the time measurement which is done using Julia's `tic()` and `toc()` functions that are based on `libuv`'s high precision timing.

ATHSA From Julia via `@hsa`.

Here, we use the kernel BRIG as in the previous two tests but this time the launch is entirely under the control of `@hsa`. The macro does a couple of things more than what is measured in the manually launched tests, mainly preparing and cleaning up the arguments (see also section 3.3.2). This test shows how much of an impact that has on run time. The corresponding column in table 4.1 shows the difference in execution time compared to the plain */JL launch of the same BRIG kernel

in parentheses after each entry. As with the others, this test is run 100 times to obtain its average run time.

4.1.2. Interpretation

The data in table 4.1 shows a few interesting trends that we want to analyze.

First, the times from the OCL/C and OCL/JL columns are very near the same. Sometimes, Julia is actually faster than C. One reason for that is likely the different timing methods. That should not have a large impact however, because we measure the time of the full 100 iterations to prevent differences in accuracy between the two clocks from biasing the result. That means the differences we see are mostly noise and we can conclude that using the HSA API from Julia is just as fast as from C.

Second, there is a near constant difference between OCL/JL and JL/JL and their corresponding `@hsa` variants OCL/ATHSA and JL/ATHSA. That suggests that, as expected, invoking a kernel via `@hsa` adds a small, constant overhead that becomes increasingly negligible as input size, and with it kernel execution time, increases.

Last, when comparing the OCL/JL and JL/JL run times there is a significant difference for the `mmul` kernel. No such discrepancy was observed when running the `mmul2d` kernel. Even though the implementation for each kernel is as similar as possible between Julia and OpenCL, the two take different paths through the compiler. That means that there are cases where the Julia to BRIG compiler will generate less efficient code than CLOC. With our `mmul` kernel, this causes an increase in execution time of around 5%. In other cases, the generated code for both is equally efficient as is apparently true for `mmul2d`.

4.1.3. Other Factors

Even though the overhead for frequent actions such as kernel launches is very important, there are other factors that play into the overall performance of an application using HSA.jl. Specifically, there are some pieces of work that only have to be done once for each application start. Others are required for each kernel or each device a kernel will be run on. Table 4.2 summarizes the most important of these one-time initialization costs.

Scope	Approx. Time required [ms]	Action
once	4	Code generator initialization
once	550	Runtime object initialization
per kernel	55-70	Julia to BRIG compilation
per kernel and device	20	BRIG kernel finalization

Table 4.2.: Overview of one-time initialization costs.

	INPUT	OCL/C	OCL/JL	OCL/ATHSA		JL/JL		JL/ATHSA	
mmul	200×100	38	37.3	37.9	(+0.6)	39.8	(+ 7%)	40.4	(+0.6)
	400×100	131	130.5	131.1	(+0.6)	137.7	(+ 6%)	138.3	(+0.6)
	800×100	489	489.9	490.6	(+0.7)	516.7	(+ 5%)	517.4	(+0.7)
	1000×100	778	777.4	778.0	(+0.6)	811.8	(+ 4%)	812.9	(+0.9)
	1000×1000	7718	7726.1	7726.1	(+0.0)	8131.8	(+ 5%)	8132.9	(+0.7)
mmul2d	200×100	46	32.8	33.4	(+0.6)	32.9	(+ 0%)	34.1	(+1.2)
	400×100	129	128.1	128.7	(+0.6)	128.1	(+ 0%)	128.9	(+0.8)
	800×100	511	511.1	511.2	(+0.1)	511.1	(+ 0%)	511.2	(+0.1)
	1000×100	798	797.5	798.2	(+0.7)	797.8	(+ 0%)	799.3	(+1.5)
	1000×1000	7843	7846.7	7847.1	(+0.4)	7846.3	(− 0%)	7847.5	(+0.2)

Table 4.1.: Average kernel run times for the different launch methods. All times in ms

INPUT	Size of the left matrix operand A as ROWS × COLUMNS
OCL/C	Running the OpenCL BRIG kernel from C
OCL/JL	Running the OpenCL BRIG kernel from Julia
OCL/ATHSA	Running the OpenCL BRIG kernel via @hsa
	In Parentheses: difference in execution time to OCL/JL
JL/JL	Running the Julia BRIG kernel from Julia
	In Parentheses: fractional difference in execution time to OCL/JL
JL/ATHSA	Running the Julia BRIG kernel via @hsa
	In Parentheses: difference in execution time to JL/JL

The code generator initialization is the construction of a few C++ objects that implement the HSAIL code generator. These are not created when Julia starts but when the first function is compiled for the HSAIL target. This process can be started explicitly by calling `HSA.init_codegen()` and takes around 4 ms on our machine.

Initialization of runtime objects is necessary before a kernel can be started. That includes finding an agent and creating a queue and a completion signal. This step is responsible for a large part of the start-up costs for any HSA application, written in any language, not just one using Julia and HSA.jl. In our tests, it takes around 550 ms.

Compiling a Julia kernel to BRIG is started by calling `HSA.brig(...)` and takes 55 ms to 70 ms for the various test kernels available to us. More complex kernels than `mmul` and `mmul2d` might take longer than that to compile, but this gives us an idea of what to expect. In any case, the resulting BRIG is automatically cached by `@hsa` which means that, when the same kernel is called again with the same argument types, it is already available.

Finalization of a BRIG kernel to obtain the device binary must happen in any HSA application using BRIG kernels, so this is not strictly overhead caused by HSA.jl. Nevertheless, we mention it here for completeness' sake and to be able to compare it to the other steps that contribute to the initial delay in running a kernel. The finalization step takes an average of 20 ms and 17 ms for `mmul` and `mmul2d` respectively. Just like the kernel's BRIG, the finalized binary is cached by `@hsa`.

Because this binary is specific to one device's ISA, a BRIG kernel could potentially be finalized multiple times for each of the system's devices. In practice this is not the case, because there currently are no systems with multiple kernel agents and ISAs. Our Kaveri APU based system, for example, has two HSA agents, the CPU and the GPU. However, only one of these, the GPU, is also a kernel agent so it is the only one we can finalize kernels for.

4.2. Limitations

Even though our work makes it significantly easier to use HSA in Julia, the current implementation has some limitations one should be aware of.

HSAIL supports programs that are composed of multiple functions and multiple modules. In its current form, our implementation does not allow calls to other functions from within a kernel function. There are some exceptions to that rule. One, obviously, are the builtin functions (`get_global_id(...)` and the like). Those are a special case of the class of functions that can be fully inlined. Other examples of this are the math operators

and some Julia intrinsics, as long as their implementation does not contain calls to any unsupported functions. Even user defined functions that only contain simple operations are supported *if* they can still be inlined. Calling unsupported functions inside a kernel will result in errors during compilation.

On a side note, the use of symbols that Julia cannot resolve does *not* immediately result in error messages to that effect. Instead, Julia emits code that tries to resolve the symbol at runtime. Unfortunately, that code includes calls to functions that are not supported inside of kernels. The overall effect of this is that instead of printing a message clearly stating that a symbol could not be resolved, a seemingly unrelated error during kernel compilation is encountered.

Another limitation is an upper ceiling on the complexity of a kernel arising from the lack of support for the Garbage Collector in kernels. Because the code that is generated for allocating the GC-frame contains references to global variables and calls to unsupported functions, we have to avoid that. Thankfully, if neither an argument nor any local variable has to be boxed, no GC-frame allocation code is emitted. Because this optimization is only aggressively applied in the release build of Julia, kernels that work there may not work using the debug version. With the release build, however, kernels can often be written in sufficiently simple fashion to avoid boxing issues.

Related to the previous two limitations is that on argument types. In general, only simple datatypes (`Int, Float, UInt ...`) are supported as well as contiguous arrays of these, which will be passed as pointers. Other types can only be passed by converting them to these simple types. While, in principle, any Julia object can be passed to a kernel, using that value inside the kernel quickly leads to either the boxing of values or a (possibly implicit) call to unsupported functions. Either of those, therefore, results in failure. When calling a kernel via `@hsa`, support for additional argument types can be enabled by adding new methods to the function `HSA.Execution.convert_arg` that take an argument of that type and convert it to one of the simple types mentioned above.

An unpleasant side effect of hijacking the compilation of kernel functions and letting them use unusual intrinsics is that they can no longer be called from plain Julia. More specifically, they can be called but that simply results in an error.

Lastly, there is, for now, no way to explicitly set the address space of an argument. HSA supports pointer arguments in the global, local or constant address space. Our implementation assumes that any pointer argument is in the global address space and automatically adds the necessary annotations during kernel compilation. In consequence, it is currently not possible to use constant or local memory from a Julia kernel.

4.3. Conclusion

Now that we have covered the limitations of our approach, it is time to evaluate it against the goals set for our implementation at the beginning of this text.

The HSA.jl package on its own already helps to simplify using the HSA runtime API. As illustrated in appendix B.1, just by moving from C to Julia with HSA.jl, the amount of code necessary can be reduced appreciably.

Also, because the package is only a thin wrapper on top of the C API and calls to a C function via the `ccall` intrinsic are very fast, the performance overhead incurred by HSA.jl is very small (see also section 4.1). The kernels themselves are still run using the same runtime mechanism which means they run just as fast as in a C application.

HSA.jl alone still requires the kernels to be written in OpenCL C or directly in HSAIL and then compiled to a BRIG file. That, of course, requires external tools like CLOC [20] or the HSAIL-Tools [31].

When paired with our modified Julia executable, these remaining disadvantages disappear. Now, kernel code in another language is no longer required and thus there is no need for external compilers.

The last remaining point is addressed by using the `@hsa` macro and the managed runtime configuration features of HSA.jl. These automate interaction with the runtime and lead to another dramatic reduction in code size as can be seen in appendix B.2.

Thus, in spite of the remaining shortcomings, we reached the objectives set forth in section 1.3.

4.4. Future Work

The limitations mentioned previously already suggest areas for further improvement. Most of them will have to be solved in concert with the Julia language developers and there is already talk of introducing changes to better support projects like this one. The PR for adding declarations to `llvmcall` [28] is a first sign of that. It has recently been accepted into main Julia.

Solutions for the problems with garbage-collection and function calls will have to be found. One attempt at solving the latter was made in the course of our work. We tried to recursively add any called functions to the module being generated while also carrying out any necessary type conversions because of changed address spaces. This effort has not succeeded so far and many problems remain to be solved.

The fact that kernel functions can not be called directly could also be remedied. One way to do that would be to automatically generate a wrapper function that is callable from Julia and takes care of correctly calling through to the kernel. Though much more

thought needs to be put into the design of this wrapper, one immediate issue is how to pass the range for this call to the wrapper.

Another interesting possibility for expansion upon this work is by exploiting the fact that we generate SPIR at an intermediate stage. This could allow running the same kernels destined for HSA using an OpenCL runtime instead by simply cutting the code generation phase short at that point. This way we could support a whole new computing platform with minimal changes.

A. Building the Project

Our software is composed of the following parts, each of which is available as a GitHub repository:

- LLVM 3.7 with HSAIL support [32], (slightly modified from upstream [10])
- libHSAIL [33], (slightly modified from upstream [31])
- **Julia** with HSAIL code generation [34] (based on Julia release-0.4 from [35])
- **HSA.jl** Package [36]

A.1. HSA.jl

The HSA.jl Package is self-contained and can be used on any compatible version of Julia. At the time of this writing, that means Julia 0.4 and later. As with any other Julia package it can be installed by running the commands shown in listing A.1 from Julia.

```
# to install the latest tagged version from the repository
Pkg.add("HSA")

# to retrieve the current development code
Pkg.clone("https://github.com/JuliaGPU/HSA.jl.git")
```

Listing A.1.: Two ways of obtaining the HSA.jl package.

A.2. Julia with HSAIL support

The modified Julia project has LLVM and libHSAIL as dependencies. Instead of building these first, the simplest thing to do is to clone the Julia code and checkout the branch for LLVM 3.7.

```
git clone https://github.com/rollingthunder/julia.git
↪ julia_hsail
cd julia_hsail
git checkout llvm37
```

Then follow the instructions in `README.md` at the root of the repository.

A.3. HSA.jl Package Development

If you want to make changes to the HSA.jl package source code you can just edit your local copy, as long as you are not working on generated code. When making changes to the generation scripts in the `gen` directory, it will be necessary to regenerate the package code by running the following from within this directory.

```
julia generate.jl
```

B. Usage Example

B.1. Direct Port from C

To illustrate the improvement in usability that comes from using Julia and HSA.jl, this section goes through a full example program for running an HSA kernel. This example is based on the official vector_copy sample distributed with the HSA runtime but was modified to instead run a kernel that performs a Julia matrix multiplication.

For each step we first present the necessary C code and then show the equivalent Julia statements. The C has been stripped of all error checking and most comments to make it less verbose. It is understood that you would usually save the `hsa_status_t` return value of each runtime call and check it for an error condition before proceeding. Where the Julia code is concerned, no such check is necessary, because it is performed by HSA.jl and any errors are surfaced as an exception. That allows consolidating the error handling for whole sections of Julia code into a single surrounding `try ... catch` block.

The full C example is available on GitHub [30].

Runtime initialization

The first step, as always, is to initialize the HSA runtime.

```
#include ”hsa.h”
#include ”hsa_ext_finalize.h”

hsa_init();
```

The only remarkable difference when using HSA.jl is that instead of performing the call directly, we construct an object that keeps a reference to the runtime. The advantage of that is that when the *Runtime* object goes out of scope and is garbage-collected, it automatically shuts down the runtime.

```
using HSA
using HSA.ExtFinalization
```

```
rt = Runtime()
```

Querying for Finalization support

Next, we have to determine if the HSA runtime has support for the finalization extension and if so, retrieve function pointers to the methods we need.

```
// Determine if the finalizer 1.0 extension is supported.
bool support;
hsa_system_extension_supported(
    HSA_EXTENSION_FINALIZER, 1, 0, &support);

// Generate the finalizer function table.
hsa_ext_finalizer_1_00_pfn_t table_1_00;
hsa_system_get_extension_table(
    HSA_EXTENSION_FINALIZER, 1, 0, &table_1_00);
```

In Julia, this is completely taken care of by the HSA.jl package. We can simply call one method that tells us, if the extension is supported and if so, use its methods.

```
if !HSA.has_ext_finalization()
    error("We need finalization support")
end
```

Finding a suitable Agent

Now we have to find the HSA Agent, that we want to run our kernel on. Generally, this will be the first (or only) GPU in the system. To find that using the C API, we have to write a callback and pass it to a function that iterates over all known agents.

```
// Determines if the given agent is of type HSA_DEVICE_TYPE_GPU
static hsa_status_t get_gpu_agent(
    hsa_agent_t agent, void *data) {
    hsa_status_t status;
    hsa_device_type_t device_type;
    status = hsa_agent_get_info(
        agent, HSA_AGENT_INFO_DEVICE, &device_type);
    if (HSA_STATUS_SUCCESS == status &&
```

```
        HSA_DEVICE_TYPE_GPU == device_type) {
        hsa_agent_t* ret = (hsa_agent_t*)data;
        *ret = agent;
        return HSA_STATUS_INFO_BREAK;
    }
    return HSA_STATUS_SUCCESS;
}

// Iterate over the agents and pick the gpu agent
hsa_agent_t agent;
hsa_iterate_agents(get_gpu_agent, &agent);
```

In Julia, while we still can write a callback if we need to, the most common cases, like filtering by the agent type, are already implemented.

```
agent = first(HSA.all_agents(dev = HSA.DeviceTypeGpu))
```

Creating a queue

With the newly obtained reference to the agent, we can create a queue for it.

```
hsa_queue_t* queue;
hsa_queue_create(agent, queue_size, HSA_QUEUE_TYPE_SINGLE,
↪ NULL, NULL, UINT32_MAX, UINT32_MAX, &queue);
```

The main advantage, here, using HSA.jl is that it fills all the optional parameters with sensible defaults. These can, of course, be overridden using keyword arguments, but they don't have to.

```
queue = Queue(agent, queue_size)
```

Loading the BRIG file

Before we can launch a kernel, we first have to load its code in the form of a BRIG file.

```
// Loads a BRIG module from a specified file
int load_module_from_file(const char* file_name,
↪  hsa_ext_module_t* module) {
    int rc = -1;
    FILE *fp = fopen(file_name, ”rb”);
    if (!fp) {
        printf(”Could not open module file %s \n”, file_name);
        exit(-1);
    }
    rc = fseek(fp, 0, SEEK_END);
    size_t file_size = (size_t) (ftell(fp) * sizeof(char));
    rc = fseek(fp, 0, SEEK_SET);
    char* buf = (char*) malloc(file_size);
    memset(buf,0,file_size);
    size_t read_size = fread(buf,sizeof(char),file_size,fp);
    if(read_size != file_size) {
        free(buf);
    } else {
        rc = 0;
        *module = (hsa_ext_module_t) buf;
    }
    fclose(fp);
    return rc;
}

/* ... */

// Load the BRIG binary.
hsa_ext_module_t module;
load_module_from_file(MODULE_FILE,&module);
```

Julia, in this case functionality from its standard library, makes this task very easy to accomplish.

```
# Load the precompiled .brig file containing the
# kernel to be executed
mod_bytes = open(readbytes, MODULE_FILE)

brig_ptr = pointer(mod_bytes)
```

Obtain the kernel code object (finalization)

Since the BRIG is only an Intermediate Representation, we have to use the finalization extension to obtain a binary for the agent.

```
// Create hsa program.
hsa_ext_program_t program;
memset(&program,0,sizeof(hsa_ext_program_t));

table_1_00.hsa_ext_program_create(
    HSA_MACHINE_MODEL_LARGE,
    HSA_PROFILE_FULL,
    HSA_DEFAULT_FLOAT_ROUNDING_MODE_DEFAULT,
    NULL,
    &program);

table_1_00.hsa_ext_program_add_module(program, module);

// Determine the agent's ISA.
hsa_isa_t isa;
hsa_agent_get_info(agent, HSA_AGENT_INFO_ISA, &isa);

// Finalize the program and extract the code object.
hsa_ext_control_directives_t control_directives;
memset(&control_directives, 0,
↪  sizeof(hsa_ext_control_directives_t));
hsa_code_object_t code_object;
table_1_00.hsa_ext_program_finalize(
    program, isa, 0, control_directives,
    "", HSA_CODE_OBJECT_TYPE_PROGRAM, &code_object);

// Destroy the program, it is no longer needed.
table_1_00.hsa_ext_program_destroy(program);
```

In Julia, this is simplified mainly by not having to pass optional arguments and not needing to keep track of the function pointers. This step is also the first to show our mechanism for automatic destruction of HSA objects upon finalization of their Julia counterparts. We go into more detail about that at the end of the example.

```
program = Program()

program_add_module(program, brig_ptr)
```

```
isa = HSA.agent_info_isa(agent)

code_object = program_finalize(program, isa, 0)

# explicitly destroy the object by calling its finalizer
# which, in turn calls hsa_ext_program_destroy
finalize(program)
```

Build the executable

We now use the code object obtained from finalization to build a runtime executable containing our kernel. Then, the executable is queried for information, such as the required amount of memory in the different special memory segments (kernarg, group and private), that we will need later on.

```
hsa_executable_t executable;
hsa_executable_create(
    HSA_PROFILE_FULL,
    HSA_EXECUTABLE_STATE_UNFROZEN,
    "", &executable);

hsa_executable_load_code_object(executable, agent, code_object,
↪ "");

hsa_executable_freeze(executable, "");

// Get the kernel symbol from the executable
hsa_executable_symbol_t symbol;
hsa_executable_get_symbol(
    executable, NULL, KERNEL_NAME, agent, 0, &symbol);

// Extract dispatch information from the symbol
uint64_t kernel_object;
hsa_executable_symbol_get_info(
    symbol,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_OBJECT,
    &kernel_object);

uint32_t kernarg_segment_size;
hsa_executable_symbol_get_info(
```

```
    symbol,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE,
    &kernarg_segment_size);

uint32_t group_segment_size;
hsa_executable_symbol_get_info(
    symbol,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE,
    &group_segment_size);


uint32_t private_segment_size;
hsa_executable_symbol_get_info(
    symbol,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE,
    &private_segment_size);
```

Since the HSA.jl property getters already know the right object type to return, the Julia code does not have to pre-allocate memory with the correct type.

```
executable = Executable()

HSA.executable_load_code_object(executable, agent, code_object)

HSA.executable_freeze(executable)

symbols = HSA.symbols(executable)
# the kernel is the only symbol in the executable
symbol = first(symbols)

kernel_object = HSA.executable_symbol_info_kernel_object(symbol)
kernarg_segment_size =
   HSA.executable_symbol_info_kernel_kernarg_segment_size(symbol)
group_segment_size =
   HSA.executable_symbol_info_kernel_group_segment_size(symbol)
private_segment_size =
   HSA.executable_symbol_info_kernel_private_segment_size(symbol)
```

Find a memory region for kernel arguments

Before we can allocate the kernel argument memory, we need to retrieve a compatible memory region for our agent. That is accomplished using a very similar mechanism to that for agent discovery, a callback and an iteration function. The criteria for a kernarg region are that it is in global memory, meaning it is accessible to both the CPU and the agent, and that the agent can read kernel arguments from there as indicated by the corresponding flag.

```
// Determines if a memory region can be used
// for kernarg allocations.
static hsa_status_t get_kernarg_memory_region(hsa_region_t
↪ region, void* data) {
    hsa_region_segment_t segment;
    hsa_region_get_info(
        region, HSA_REGION_INFO_SEGMENT, &segment);
    if (HSA_REGION_SEGMENT_GLOBAL != segment) {
        return HSA_STATUS_SUCCESS;
    }

    hsa_region_global_flag_t flags;
    hsa_region_get_info(
        region, HSA_REGION_INFO_GLOBAL_FLAGS, &flags);
    if (flags & HSA_REGION_GLOBAL_FLAG_KERNARG) {
        hsa_region_t* ret = (hsa_region_t*) data;
        *ret = region;
        return HSA_STATUS_INFO_BREAK;
    }

    return HSA_STATUS_SUCCESS;
}

/* ... */

// Find a memory region that supports kernel arguments.
hsa_region_t kernarg_region;
hsa_agent_iterate_regions(agent, get_kernarg_memory_region,
↪ &kernarg_region);
```

The Julia code in this case is mostly identical, though it is still more concise.

```
regions = HSA.regions(agent)

karg_region_idx = findfirst(r -> begin
    segment = HSA.region_info_segment(r)
    if (HSA.HSA_REGION_SEGMENT_GLOBAL == segment)
        flags = HSA.region_info_global_flags(r)
        return (flags & HSA.HSA_REGION_GLOBAL_FLAG_KERNARG) != 0
    end
    return false
end, regions)

kernarg_region = regions[karg_region_idx]
```

Allocate the kernel arguments

Knowing how much argument memory is required and where to put it allows us to allocate an appropriately sized buffer.

```
struct args_t {
    void* a; // first matrix
    void* b; // second matrix
    void* c; // output matrix
    uint64_t acols; // number of columns in a
} __attribute__ ((aligned(16)));

struct args_t args;

double* a = (double*)malloc(ASIZE);
hsa_memory_register(a, ASIZE);
double* b = (double*)malloc(BSIZE);
hsa_memory_register(b, BSIZE);
double* c = (double*)malloc(CSIZE);
hsa_memory_register(c, CSIZE);

args->a = a;
args->b = b;
args->c = c;
args->acols = ACOLS;

// Allocate the kernel argument buffer from the correct region.
```

```
void* kernarg_address = NULL;
hsa_memory_allocate(
    kernarg_region, kernarg_segment_size, &kernarg_address);
memcpy(kernarg_address, &args, sizeof(args));
```

The corresponding Julia goes through exactly the same steps. The only notable difference is that when interfacing with C code, a struct is replaced by an instance of a Julia immutable type.

```
immutable Args
    a::Ptr{Float64}
    b::Ptr{Float64}
    c::Ptr{Float64}
    acols::Uint64
end

a = Array(Float64, ASIZE)
HSA.memory_register(a, sizeof(a))
b = Array(Float64, BSIZE)
HSA.memory_register(b, sizeof(b))
c = Array(Float64, CSIZE)
HSA.memory_register(c, sizeof(c))

kernargs = Args(
    pointer(a),
    pointer(b),
    pointer(c),
    ACOLS)

alloc = HSA.memory_allocate(kernarg_region, kernarg_segment_size)
kernarg_address = convert(Ptr{Args}, alloc.ptr)
unsafe_store!(kernarg_address, kernargs)
```

Prepare the kernel dispatch packet

Having finished the preparations, we can finally fill out the kernel dispatch packet that has to be written to the queue to launch the kernel.

```
// Create a signal to wait for the dispatch to finish.
hsa_signal_t signal;
```

```
hsa_signal_create(1, 0, NULL, &signal);

// Obtain the current queue write index.
uint64_t index = hsa_queue_load_write_index_relaxed(queue);

// Write the AQL packet at the calculated queue index address.
const uint32_t queueMask = queue->size - 1;
const uint64_t maskedIndex = index & queueMask;
hsa_kernel_dispatch_packet_t* dispatch_packet =
 ↪ ((hsa_kernel_dispatch_packet_t*)(queue->base_address));
dispatch_packet = &dispatch_packet[maskedIndex]

dispatch_packet->setup  |= 1 «
 ↪ HSA_KERNEL_DISPATCH_PACKET_SETUP_DIMENSIONS;
dispatch_packet->workgroup_size_x = (uint16_t)1;
dispatch_packet->workgroup_size_y = (uint16_t)1;
dispatch_packet->workgroup_size_z = (uint16_t)1;
dispatch_packet->grid_size_x = (uint32_t) (N);
dispatch_packet->grid_size_y = (uint32_t) (1);
dispatch_packet->grid_size_z = (uint32_t) (1);
dispatch_packet->completion_signal = signal;
dispatch_packet->kernel_object = kernel_object;
dispatch_packet->kernarg_address = (void*) kernarg_address;
dispatch_packet->private_segment_size = private_segment_size;
dispatch_packet->group_segment_size = group_segment_size;

uint16_t header = 0;
header |= HSA_FENCE_SCOPE_SYSTEM «
 ↪ HSA_PACKET_HEADER_ACQUIRE_FENCE_SCOPE;
header |= HSA_FENCE_SCOPE_SYSTEM «
 ↪ HSA_PACKET_HEADER_RELEASE_FENCE_SCOPE;
header |= HSA_PACKET_TYPE_KERNEL_DISPATCH «
 ↪ HSA_PACKET_HEADER_TYPE;
```

On the Julia side again, the same actions take fewer lines to accomplish, mostly because of optional arguments. We also avoid writing code to pack the header fields because that is already implemented in HSA.jl.

```
signal = Signal(value = 1)

index = HSA.load_write_index(queue, Val{Relaxed})
```

```
dispatch_packet = KernelDispatchPacket(kernel_object, N;
    kernarg_address = kernarg_address,
    private_segment_size = private_segment_size,
    group_segment_size = group_segment_size,
    completion_signal = signal)
dispatch_packet.header.acquire_fence_scope = HSA.FenceScopeSystem
dispatch_packet.header.release_fence_scope = HSA.FenceScopeSystem
```

Dispatch the kernel

The last thing missing before the kernel is actually executed is to write the packet header to the queue and ring its doorbell. After that, we have to wait for the completion signal to tell us when we can proceed.

```
__atomic_store_n(
    (uint16_t*)(&dispatch_packet->header),
    header,
    __ATOMIC_RELEASE);

// Increment the write index and
// ring the doorbell to dispatch the kernel.
hsa_queue_store_write_index_relaxed(queue, index+1);
hsa_signal_store_relaxed(queue->doorbell_signal, index);

// Wait on the dispatch completion signal
// until the kernel is finished.
hsa_signal_value_t value = 1;
while(!(value < 1)) {
    value = hsa_signal_wait_acquire(
        signal,
        HSA_SIGNAL_CONDITION_LT,
        1,
        UINT64_MAX,
        HSA_WAIT_STATE_BLOCKED);
}
```

HSA.jl handles writing the packet to the queue correctly. Specifically, it writes the body first and then, atomically, the header. That is how it must be to avoid the agent's packet processor seeing an incomplete packet. The header's type field is used to tell it when a packet is ready for processing which might begin even before the doorbell signal has

been rung. This is the reason for writing the header including its type field only *after* the rest of the packet is written.

```
queue[index] = dispatch_packet
HSA.store_write_index!(queue, UInt64(index + 1))
HSA.store!(queue.doorbell_signal, Int64(index))
check("Dispatching the kernel")

value = 1
while !(value < 1)
    value = wait(
        signal, :(<), 1;
        wait_state_hint = HSA.WaitStateBlocked)
end
```

Clean up HSA objects

After the kernel execution completes and we are done with processing, the allocated resources have to be cleaned up. That means deallocating kernarg memory, destroying runtime objects and deregistering buffers with the runtime. After all that is done, the runtime itself is shut down.

```
// Cleanup all allocated resources.
hsa_memory_free(kernarg_address);

hsa_signal_destroy(signal);
hsa_executable_destroy(executable);
hsa_code_object_destroy(code_object);
hsa_queue_destroy(queue);

hsa_memory_deregister(a, ASIZE);
free(a);
hsa_memory_deregister(b, BSIZE);
free(b);
hsa_memory_deregister(c, CSIZE);
free(c);

hsa_shut_down();
```

Aside from the deregistration of buffers, all resource disposal in Julia is the job of finalizers. These are functions that HSA.jl has associated with the wrapper objects that

represent a runtime resource needing to be destroyed. The finalizers will be automatically invoked, when there are no more references to the object they are associated with and it is garbage-collected. That means any program that does not explicitly destroy these resources would still not be leaking resources because these would eventually be cleaned up. The finalizers provide a uniform and automatic way to control resource destruction, that is native to Julia. Rather than leaving it to the garbage-collector, the best thing to do is to explicitly finalize the objects after use.

```
# kernarg memory
finalize(alloc)

# arguments
HSA.memory_deregister(a, sizeof(a))
HSA.memory_deregister(b, sizeof(b))
HSA.memory_deregister(c, sizeof(c))

# HSA objects
finalize(signal)
finalize(executable)
finalize(code_object)
finalize(queue)

# shutdown the runtime
finalize(rt)
```

B.2. Using Managed Kernel Execution

While Julia code using HSA.jl is simpler and more concise than the corresponding C code, there still is a lot of room for improvement. Mainly, there is too much boiler-plate code that is essentially the same for any kernel or program. Ideally, we would like this code to be generated automatically so we can focus on the actual logic of our program. That is exactly what `@hsa` does. Below is an implementation of the same matrix multiplication sample as above, using `@hsa`. Even including the kernel function, which is not part of the earlier code snippets, this version is much shorter than the previous one. Certainly, this only works if you are happy with the default choices `@hsa` makes in creating runtime objects. But the most important parts of the configuration, like the target agent and queue, can be changed if necessary (see section 3.1.2).

This sample is an adaptation of one of several kernels that are run as part of the automated test suite of HSA.jl.

```
using HSA

@hsa_kernel function mmul(a,b,c,acols)
    # one kernel invocation per column of the result matrix
    arows = get_global_size(UInt32(0))
    # i = col
    i = get_global_id(UInt32(0))
    # j = row
    for j = 1:arows
        c_ij = 0.0
        for k = 1:acols
            a_idx = (k-1) * arows + j
            b_idx = i * acols + k
            c_ij += a[a_idx] * b[b_idx]
        end

        c_idx = i * arows + j
        c[c_idx] = c_ij
    end
    return nothing
end

N =  1024*1024

a = Array(Float64, N)
rand!(a)
b = Array(Float64, N)
rand!(b)
c = Array(Float64, N)
rand!(c)

expected = a * b

@hsa (N) mmul(a, b, c, ACOLS)

@assert expected == c
```

C. Kernels used in Performance Testing

mmul — linear range

Listing C.1.: mmul — Julia implementation

```
@hsa_kernel function mmul(a,b,c,acols)
    # one kernel invocation per column of the result matrix
    arows = get_global_size(UInt32(0))
    # i = col
    i = get_global_id(UInt32(0))
    # j = row
    for j = 1:arows
        c_ij = 0.0
        for k = 1:acols
            a_idx = (k-1) * arows + j
            b_idx = i * acols + k
            c_ij += a[a_idx] * b[b_idx]
        end

        c_idx = i * arows + j
        c[c_idx] = c_ij
    end

    return nothing
end
```

Listing C.2.: mmul — OpenCL implementation

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

kernel void mmul(
        global double* a,
        global double* b,
        global double* c,
        ulong acols) {
    ulong arows = get_global_size(0);
```

```
    ulong i = get_global_id(0);
    for(ulong j = 0; j < arows; ++j) {
        double c_ij = 0.0;
        for(ulong k = 0; k < acols; ++k) {
            c_ij += a[k*arows + j] * b[i * acols + k];
        }
        c[i*arows + j] = c_ij;
    }
}
```

mmul2d — two-dimensional range

Listing C.3.: mmul2d — Julia implementation

```
@hsa_kernel function mmul2d(a,b,c,acols)
    # one kernel invocation per cell of the result matrix
    arows = get_global_size(UInt32(0))
    # i = col
    i = get_global_id(UInt32(1))
    # j = row
    j = get_global_id(UInt32(0)) + 1

    c_ij = 0.0
    for k = 1:acols
        a_idx = (k-1) * arows + j
        b_idx = i * acols + k
        c_ij += a[a_idx] * b[b_idx]
    end

    c_idx = i * arows + j
    c[c_idx] = c_ij

    return nothing
end
```

Listing C.4.: mmul2d — OpenCL implementation

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

kernel void mmul2d(
        global double* a,
        global double* b,
        global double* c,
        ulong acols) {
    ulong arows = get_global_size(0);
    ulong i = get_global_id(1);
    ulong j = get_global_id(0);

    double c_ij = 0.0;
    for(ulong k = 0; k < acols; ++k) {
        c_ij += a[k*arows + j] * b[i * acols + k];
    }
    c[i*arows + j] = c_ij;
}
```

References

[1] *NVIDIA CUDA*. URL: https://developer.nvidia.com/cuda-zone (visited on 2015-11-16).

[2] *OpenCL. Open Compute Language*. Khronos. URL: https://www.khronos.org/opencl/ (visited on 2015-09-21).

[3] *HSA Foundation*. URL: http://www.hsafoundation.com/ (visited on 2015-09-21).

[4] *NumPy. Scientific Computing in Python*. URL: http://www.numpy.org/ (visited on 2015-09-21).

[5] *MATLAB. The Language of Technical Computing*. MathWorks. URL: http://www.mathworks.com/products/matlab/ (visited on 2015-09-21).

[6] *The Julia Language*. URL: http://julialang.org/ (visited on 2015-09-21).

[7] R.R. Schaller. "Moore's law: past, present and future". In: *Spectrum, IEEE* 34.6 (1997-06), pp. 52–59. ISSN: 0018-9235. DOI: 10.1109/6.591665.

[8] *HSA Platform System Architecture Specification*. Version 1.0 Final. HSA Foundation. 2015-01-23. URL: http://www.hsafoundation.com/?ddownload=4944 (visited on 2015-09-21).

[9] Blake A. Hechtman et al. Derek R. Hower. "Heterogeneous-race-free Memory Models". In: *ASPLOS'14* (2014-03-01).

[10] *HSAIL LLVM Tree. Repository for the ongoing development of the HSAIL Target for LLVM*. URL: https://github.com/HSAFoundation/HLC-HSAIL-Development-LLVM (visited on 2015-09-21).

[11] *MSDN: C++AMP. Accelerated Massive Parallelism*. URL: https://msdn.microsoft.com/en-us/library/hh265137.aspx (visited on 2015-11-16).

[12] *Kalmar: An open source C++ compiler for heterogeneous devices*. URL: https://bitbucket.org/multicoreware/cppamp-driver-ng/wiki/Home (visited on 2015-09-21).

[13] *A PARallel API. Running java kernels on GPUs*. URL: https://aparapi.github.io/ (visited on 2015-11-16).

[14] Yuan Yu et al. Christopher J. Rossbach. "Dandelion: a Compiler and Runtime for Heterogeneous Systems". In: SOSP'13: The 24th ACM Symposium on Operating Systems Principles, 2013-11. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=201110 (visited on 2015-10-21).

[15] *OpenCL.jl CL Compiler. Experimental support for translating Julia to OpenCL C.* URL: https://github.com/JuliaGPU/OpenCL.jl/tree/clcompiler2 (visited on 2015-11-16).

[16] *CUDA.jl with PTX Target. Julia Programming interface for CUDA.* URL: https://github.com/JuliaGPU/CUDA.jl (visited on 2015-09-21).

[17] Tim Besard. *Pull Request: NVPTX/Cuda backend support in Julia.* URL: https://github.com/JuliaLang/julia/pull/11516 (visited on 2015-09-21).

[18] *HSA Programmer's Reference Manual.* Version 1.0.1. HSA Foundation. 2015-07-02. URL: http://www.hsafoundation.com/?ddownload=4945 (visited on 2015-09-21).

[19] *HSA Runtime Programmer's Reference Manual.* Version 1.0. HSA Foundation. 2015-02-06. URL: http://www.hsafoundation.com/?ddownload=4946 (visited on 2015-09-21).

[20] *HSA OpenCL Offline Compiler. Repository for the OpenCL to HSA BRIG offline compiler.* URL: https://github.com/HSAFoundation/CLOC (visited on 2015-09-21).

[21] *Julia Documentation: Calling C and Fortran Code.* URL: https://julia.readthedocs.org/en/latest/manual/calling-c-and-fortran-code/ (visited on 2015-09-30).

[22] *Clang.jl. Julia interface to libclang and C wrapper generator.* URL: https://github.com/ihnorton/Clang.jl (visited on 2015-09-21).

[23] *SPIR Specification.* Version 2.0 Provisional. Khronos. 2014-06-04. URL: https://www.khronos.org/registry/spir/specs/spir_spec-2.0.pdf (visited on 2015-09-21).

[24] *Compiling Julia to OpenCL SPIR instead of OpenCL C.* URL: https://github.com/JuliaGPU/OpenCL.jl/issues/29 (visited on 2015-10-19).

[25] *HSA OpenCL Offline Compiler. Script.* URL: https://github.com/HSAFoundation/CLOC/blob/e0b3e0d24115bd3357c185306ff6687df280a330/bin/cloc.sh (visited on 2015-10-19).

[26] Tim Besard. *Pull Request: Address space preserving bitcasts.* URL: https://github.com/JuliaLang/julia/pull/9423 (visited on 2015-09-21).

[27] *Julia Documentation: LLVM Interface.* URL: https://julia.readthedocs.org/en/latest/stdlib/c/?highlight=llvmcall#llvm-interface (visited on 2015-10-21).

[28] Valentin Churavy. *Pull Request: Adding declarations to llvmcall.* URL: https://github.com/JuliaLang/julia/pull/11604 (visited on 2015-09-21).

[29] *HSA.jl timing scripts. Used to gather performance data on HSA.jl.* URL: https://github.com/JuliaGPU/HSA.jl/tree/gr/thesis/sample/timing (visited on 2015-11-22).

[30] *Example application, running Julia matrix multiplication on HSA. GitHub Repository.* URL: https://github.com/rollingthunder/hsa_mmul_sample/tree/master (visited on 2015-11-11).

[31] *HSAIL Tools. Repository for the HSAIL Dis-/Assembler and libHSAIL.* URL: https://github.com/HSAFoundation/CLOC (visited on 2015-09-21).

[32] *Patched LLVM 3.7 with HSAIL. GitHub Repository.* URL: https://github.com/rollingthunder/HLC-HSAIL-Development-LLVM/tree/hsail-stable-3.7 (visited on 2015-10-25).

[33] *Patched libHSAIL. GitHub Repository.* URL: https://github.com/rollingthunder/HSAIL-Tools/tree/master (visited on 2015-10-25).

[34] *Julia, modified for HSA code generation. GitHub Repository.* URL: https://github.com/rollingthunder/julia/tree/llvm37 (visited on 2015-10-25).

[35] *Julia GitHub repository.* URL: https://github.com/JuliaLang/julia (visited on 2015-09-21).

[36] *HSA.jl. GitHub Repository.* URL: https://github.com/JuliaGPU/HSA.jl/tree/gr/master (visited on 2015-10-25).

Glossary

API Application Programming Interface.

APU Accelerated Processing Unit.

AQL Architected Queueing Language.

AS Address Space.

AST Abstract Syntax Tree.

Boxing The process of allocating memory on the heap and storing a value there. This separates the lifetime of the value from that of the current function call. In garbage-collected languages, these values on the heap are under the control of the Garbage Collector.

BRIG HSAIL binary format.

CLOC OpenCL Offline Compiler.

CPU Central Processing Unit.

CU Compute Unit.

CUDA Compute Unified Device Architecture.

DSP Digital Signal Processor.

GC Garbage Collector.

GPGPU General Purpose GPU.

GPU Graphics Processing Unit.

HSA Heterogeneous Systems Architecture.

HSAIL HSA Intermediate Language.

IGP Integrated Graphics Processor.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JIT Just-In-Time.

Kernel The program to be executed by one thread in a data parallel computation.

LINQ Language INtegrated Query.

LLVM The LLVM Compiler Infrastructure, formerly called "Low-Level Virtual Machine", is an open source project that supplies all necessary components to build a compiler.

OpenCL Open Compute Language.

PR Pull Request.

PTX Parallel Thread Execution.

Range The set of kernel executions for some input and output data sizes.

REPL Read-Evaluate-Print Loop.

SEH Structured Exception Handling.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Threads.

SPIR OpenCL Standard Portable Intermediate Representation.

SSA Single Static Assignment.

Declaration of authorship

I, Georg Rollinger, hereby declare that I have authored this work on my own and without using any other sources or tools than those mentioned here. I have never before submitted this work to attain an academic degree.

Erklärung über Urheberschaft

Ich, Georg Rollinger, erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und dabei keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Ich habe die Arbeit noch nicht zur Erlangung eines akademischen Grades eingereicht.