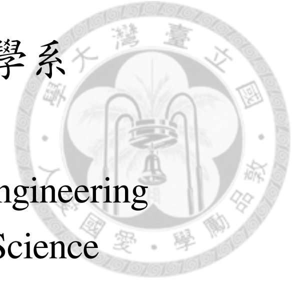


國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

在 HSA 系統上以間接 Finalizer 進行程式碼最佳化

Code Optimization in an Indirect Finalizer for an HSA-based System

李弘宇

Hong-Yu Lee

指導教授：徐慰中博士

Advisor: Wei-Chung Hsu, Ph.D.

中華民國 105 年 7 月

July, 2016

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

摘要

異質系統架構 (HSA) 是由非營利組織 HSA 基金會提出，旨在降低異質環境中開發難度。一個 HSA 系統可以包括多種不同指令集的裝置。

由該基金會提出之 HSA 中介語言 (HSAIL) 用以抽象化在異質計算環境中不同裝置的實作細節。程式設計師將程式碼以 HSAIL 表示之，這些 HSAIL 程式碼便可以在多種裝置上執行，例如中央處理器，圖形處理器，或訊號處理器。一個 HSA 的系統實作包括硬體元件以及 HSA 執行時期軟體程式庫，其中包括 finalizer，而一個 finalizer 可將 HSAIL 中介語言翻譯爲指定裝置的指令集。

本學位論文以 AMD Carrizo APU 處理器內的圖形處理器爲目標裝置語言，探討 OpenCL C 核心語言 (kernel language) 的間接翻譯 HSAIL 中介語言 (indirect finalization) 上的議題。爲了間接翻譯 HSAIL 程式碼，作者開發一個 HSAIL 語言前端用以轉換 HSAIL 中介語言成 LLVM 中介語言模組，該前端進而與 LLVM 中的 AMDGPU 後端整合。作者開發八個基於 AMD APP SDK 的 HSA 應用程式。本學位論文亦討論三個 LLVM 最佳化選項在間接翻譯 HSAIL 程式中的影響，其中一個爲作者所開發。

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

Abstract

Heterogeneous System Architecture (HSA) is an architecture standard developed by the non-profit HSA foundation aiming to make it easier to program for heterogeneous computing. A single HSA system may include an assortment of devices with distinct instruction set architectures (ISAs).

HSA Intermediate Language (HSAIL) is an intermediate language defined by the foundation to abstract away the heterogeneity in the HSA computing environment. A programmer builds code using HSAIL in order to allow it to be executed on a wide range of devices, such as CPU, GPU, DSP, etc. An HSA implementation encompasses the hardware components and an HSA runtime including the finalizer. A finalizer translates HSAIL modules into a given ISA.

This work investigates the issues of indirect finalization, kernel code optimization and performance on an HSA compliant system with an AMD Carrizo APU targeting its GPU architecture. To finalize the HSAIL code indirectly, a HSAIL frontend is developed to translate HSAIL codes to LLVM modules, which is then integrated with the LLVM AMDGPU backend. The author develops eight OpenCL benchmarks whose kernel codes are from AMD APP SDK. The work also shows the impact of three LLVM optimizaton passes, one of

doi:10.6342/NTU201601752

which is written by the author, on indirect finalization.

doi:10.6342/NTU201601752

Contents

摘要 i

Abstract iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background 7

2.1 Overview of AMD GCN Architecture 7

2.2 OpenCL Programming Model 9

2.3 The HSA Architecture 10

2.3.1 HSAIL 11

2.4 The LLVM Compiler Infrastructure 12

2.4.1 Clang and libclc 13

2.4.2 llc and AMDGPU backend 13

2.5 OpenCL Kernel Code Execution and Compilation on a HSA-Compliant System 13

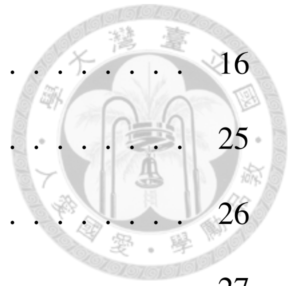
2.5.1 Kernel Code Compilation/Finalization Process 14

2.5.2 Kernel Code Execution 15

2.6 HSAIL Frontend 16

2.6.1 HSAIL-Tools and HSAIL Frontend 16

doi:10.6342/NTU201601752



2.6.2 Design and Implementation of the HSAIL Frontend 16

3 Experimental Methodology 25

3.1 Methodology 26

3.2 Benchmarks Development 27

4 Results and Analysis 29

4.1 Kernel Finalization Time 29

4.2 Kernel Execution Time 30

4.2.1 Comparison between Indirect and Direct Finalization 31

4.2.2 Comparison between Indirect Finalization and Direct Compilation 33

4.3 LLVM Optimization Options 35

4.3.1 Machine Instruction Scheduling 35

4.3.2 Peephole Optimization in GCN MAD and MAC Instructions . . . 36

5 Related Work 41

6 Conclusion 43

A Detailed System Configuration 45

Bibliography 47

doi:10.6342/NTU201601752

List of Figures

2.1 AMD GCN Generation 3 Series Block Diagram [3] 8

2.2 OpenCL Offline Compiler (CLOC) [1] 12

2.3 OpenCL Kernel Code Compilation Processes. 14

4.1 Kernel compilation of Indirect and Direct Finalization. 30

4.2 Kernel Execution Time. 32

4.3 The performance of manually reordered MatrixMultiplication HSAIL code. 33

4.4 Comparison between Bottom-Up and Default Top-Down Instruction Scheduling Schemes. 37

4.5 Constant Folding in Madmk and Mul Instruction. 39

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

List of Tables

3.1 Benchmark Description 25

3.2 System Environment 25

A.1 Detailed Software Stack 45

doi:10.6342/NTU201601752

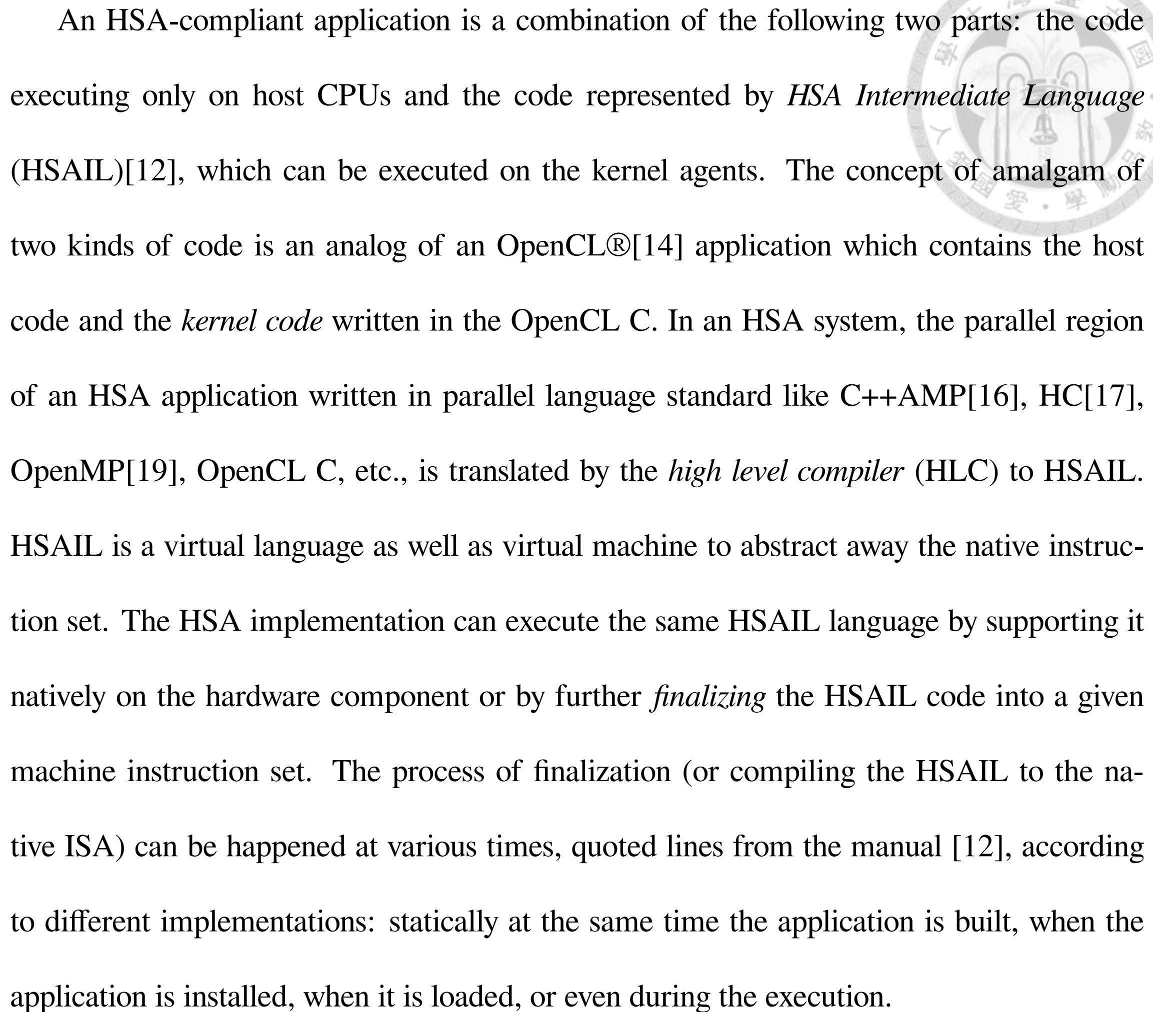
doi:10.6342/NTU201601752

1 Introduction

Over the past few years, heterogeneous computing has become prevalent in the computer society. For instance, a computing system may use multiple devices like *graphics processing units* (GPUs), co-processor like Intel ® Xeon Phi, *digital signal processor* (DSP), etc., to accelerate some suitable programs. For regular programs like matrix multiplication, GPU can offer substantial speedup over a computing platform with traditional CPUs. However, the legacy GPU compute on a heterogeneous platform has several limitations such as separate address spaces, high overhead dispatch, etc. For GPU and other devices, a raw pointer to a location in the CPU main memory can't be recognized and vice versa. In order to make the GPU process the data, the programmer must explicitly call the data copy application program interface (API) to move the data back and forth between CPU main memory and GPU global memory through the PCIe bus. This requires programmers to handle the data movements and do the data marshalling explicitly in order to leverage the computing resources on devices other than CPUs. In addition, when a program dispatches tasks to GPU, the cost of these dispatches is rather high since it may involve system calls to let the GPU driver do the rest of the work in the implementation.

Heterogeneous System Architecture (HSA) is an architecture standard proposed by the non-profit HSA foundation [10] aiming to address the aforementioned limitations and make it easier to program for heterogeneous computing. The essence of HSA is to provide a unified computing platform to let computing units cooperating with each other.

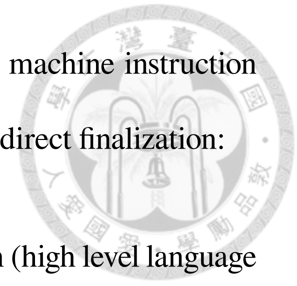
doi:10.6342/NTU201601752



An HSA-compliant application is a combination of the following two parts: the code executing only on host CPUs and the code represented by *HSA Intermediate Language* (HSAIL)[12], which can be executed on the kernel agents. The concept of amalgam of two kinds of code is an analog of an OpenCL®[14] application which contains the host code and the *kernel code* written in the OpenCL C. In an HSA system, the parallel region of an HSA application written in parallel language standard like C++AMP[16], HC[17], OpenMP[19], OpenCL C, etc., is translated by the *high level compiler* (HLC) to HSAIL. HSAIL is a virtual language as well as virtual machine to abstract away the native instruction set. The HSA implementation can execute the same HSAIL language by supporting it natively on the hardware component or by further *finalizing* the HSAIL code into a given machine instruction set. The process of finalization (or compiling the HSAIL to the native ISA) can be happened at various times, quoted lines from the manual [12], according to different implementations: statically at the same time the application is built, when the application is installed, when it is loaded, or even during the execution.

Suppose a vendor would like to support multiple devices with different ISAs and these devices can thus participate in an HSA system as kernel agents. Note that as a kernel agent, the device joins the unified HSA memory model, and is able to access the same pointer to memory location no matter where it is located in the kernel agents. To meet this goal, a vendor should implement the *HSA finalizer*, a utility turning the HSAIL code into a given device ISA. If a vendor implements the finalizer from the scratch by *directly* translating the HSAIL code to a native ISA, the various N sets of backend implementations and optimizations should be implemented correctly and be verified thoroughly. The implementation cost of direct finalization is prohibitively high to a vendor. Another viable solution to a vendor is that trying to leverage the existing compiler framework like LLVM[23] or GCC[9] to implement the finalizer by *indirectly* translating the HSAIL code to their intermediate rep-

doi:10.6342/NTU201601752



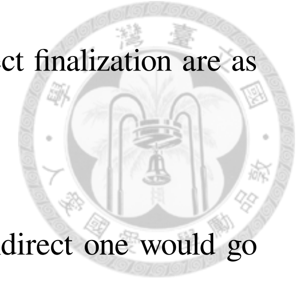
resentations, respectively. These IRs are further translated to a native machine instruction set. The indirect finalization using LLVM has several advantages over direct finalization:

1. *Leverage existing LLVM IR optimizations*. The three-phase design (high level language frontends, common LLVM IR optimizations, and architecture-specific backends) in the LLVM draws more attention to the programmers to use this infrastructure than it would if the LLVM infrastructure were to support one target and one backend. The LLVM community implements many common optimizations, and the existing IR optimizations have been well-tested. A vendor can leverage the LLVM and find out the critical transformations to its backend without implementing them directly. A vendor can then further develop the suitable transformations for the native ISA and integrate them into the LLVM pass manager. This lightens the burden of a vendor to craft the finalizer.

2. *Leverage existing LLVM backends*. A vendor may choose a popular CPU architecture such as X86 or ARM to integrate it into the system. If this is the case, it can leverage the existing backends and their own specific backend optimizations to generate the device code. Leveraging robust and reliable open source compiler makes a vendor take much less effort than crafting a finalizer from the scratch. In the LLVM, a CPU backend like x86 has many complicated optimization passes and numerous options for the programmers to utilize. The cost of implementing these optimizations again is not realistic for a vendor. Thus, it is an applicable solution for a vendor to prefer to refine an existing backend.

3. *Fast to market*. By leveraging the existing codebase, a vendor can release an HSA compliant system faster than the other competitors who implement the direct finalizers.

doi:10.6342/NTU201601752



On the other hands, the pros of direct finalization over the indirect finalization are as follows.

1. *Generate code faster*. As opposed to direct finalization, the indirect one would go through at least several extra steps. These steps are ordered as follows.

   (a) Convert HSAIL to raw LLVM module.

   (b) HSAIL frontend optimization.

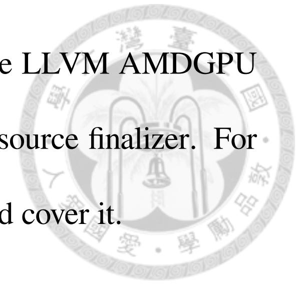
   (c) Apply some ineffective LLVM optimization passes. Note that the ineffectiveness of some specific optimization passes is due to the HLC does some kind of optimizations before emitting the HSAIL code, and this makes the indirect finalization doesn't need to go through these optimizations again.

   (d) Allocate registers in two steps. The indirect method would lose the purpose of early RA done on the HSAIL by HLC, and turn the HSAIL registers as well as the memory allocations into LLVM infinite register sets in static single assignment (SSA) form[20] and *alloca* instructions, respectively. The indirect finalizer can then apply one of existing LLVM register allocation schemes, such as fast, basic, greedy, or pbqp, to allocate the device registers.

   These extra steps in the indirect finalization make the indirect finalization slower than the direct finalization.

2. *Generate higher quality code* . For a vendor provides a closed source compiler, which usually generates higher quality code than the code emitted by open source alternatives since the developers of the vendor fully undertand its device microarchitecture. For instance, the proprietary compiler, icc, provided by Intel, usually produces better code than other open source compiler like GCC or LLVM. It is the case when it

doi:10.6342/NTU201601752



comes to one of the founder of the HSA foundation, AMD. The LLVM AMDGPU backend generates inferior code than that emitted from closed source finalizer. For more information of this comparison, the rest of the thesis would cover it.

3. *A smaller finalizer executable and less library dependency.* The indirect finalization is dependent on the LLVM compiler infrastructure, which would inevitable link extraneous LLVM shared library at the runtime. As the LLVM is a fast-paced project, the finalizer codebase would need to port to the newer LLVM version once the LLVM C++ API is updated. What's more, the LLVM bitcode compatibility issue may also complicate the issue. On the other hand, the direct finalizer can only implement the essential functionality without depending on the LLVM infrastructure. This would lead to smaller executable and reduce shared library dependencies in the direct finalizer.

This work investigates the issues of indirect finalization, kernel code optimization and performance on a HSA compliant system with an AMD Carrizo APU targeting its GPU architecture. For its GPU architecture, there are 8 compute units (CUs) running on 300–800 MHz clock rate.

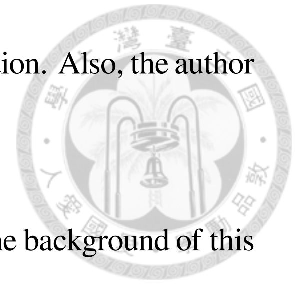
The works in this thesis are as follows.

1. Develop the HSA benchmarks based on the AMD APP SDK[5][4] in order to exploit the potential of the HSA architecture and explore the code optimization in the indirect finalizer.

2. Develop the preliminary HSAIL frontend and integrate it with LLVM AMDGPU backend to do the indirect finalization.

3. Evaluate the kernel code compilation time and execution time of indirect and direct finalization.

doi:10.6342/NTU201601752



4. Identify some LLVM optimization passes for the indirect finalization. Also, the author develops a peephole optimization pass.

The rest of the thesis is organized as follows. Section 2 introduces the background of this work. Section 3 describes the benchmarks and experiment environment. Section 4 discusses and analyzes the results. Section 5 presents the related works. Finally, section 6 concludes the work.

doi:10.6342/NTU201601752

2 Background

In this section, the author briefly introduces the AMD Graphics Core Next (GCN) architecture, OpenCL programming model, HSA architecture, and LLVM compiler infrastructure. Also, LLVM IR and HSAIL are introduced. It is then followed by the introduction of OpenCL kernel code generation and HSA application execution on a HSA-compliant system. In the remainder of the section, the author shows the design and implementation issues of a HSAIL frontend, which is based on an open source project HSAIL-Tools[2].

2.1 Overview of AMD GCN Architecture

The target GPU architecture of the OpenCL kernel execution in this work is GCN Generation 3 Volcanic Islands. The Figure 2.1 gives the series block diagram of the architecture. It includes an array of data-parallel processor (DPP) and other logic.

The DPP is a set of compute units (CUs), and it is the fundamental unit of computation. Each CU contains several items including (1) four vector single instruction multiple data (SIMD) compute units, each of which has a vector ALU, vector general-purpose registers (VGPRs), etc., (2) scalar ALU and scalar GPRs (SGPRs), (3) workgroup-private memory, Local Data Share (LDS), (4) access permission to other memory subsystems and cache system. In this work, the author uses an AMD Carrizo APU targeting its GPU architecture, which contains eight CUs, to construct a HSA compliant system.

doi:10.6342/NTU201601752

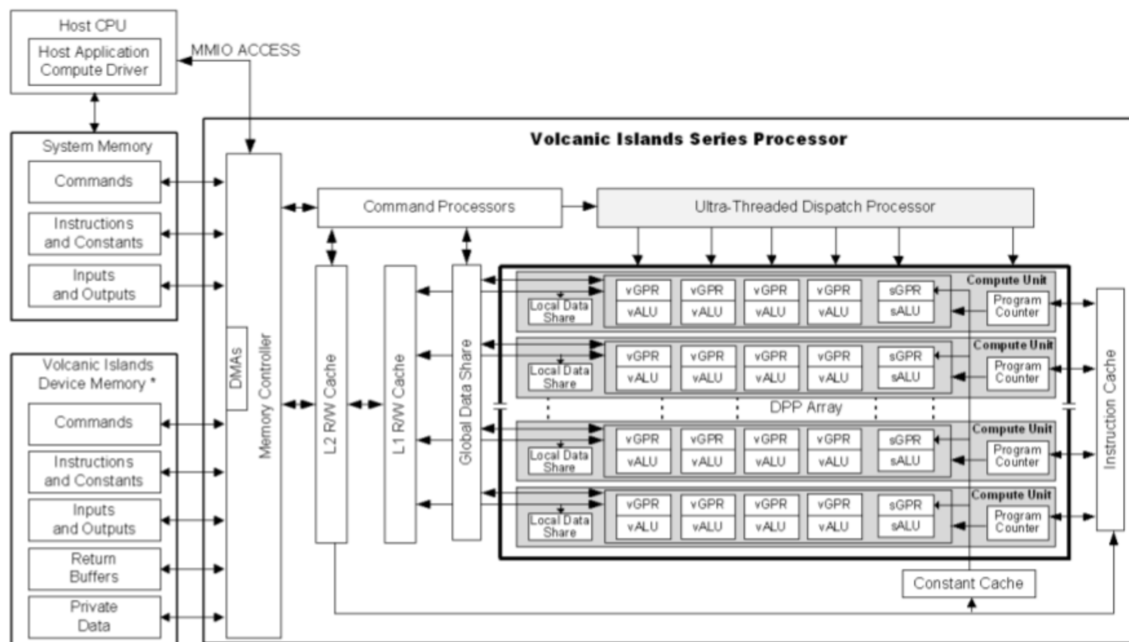
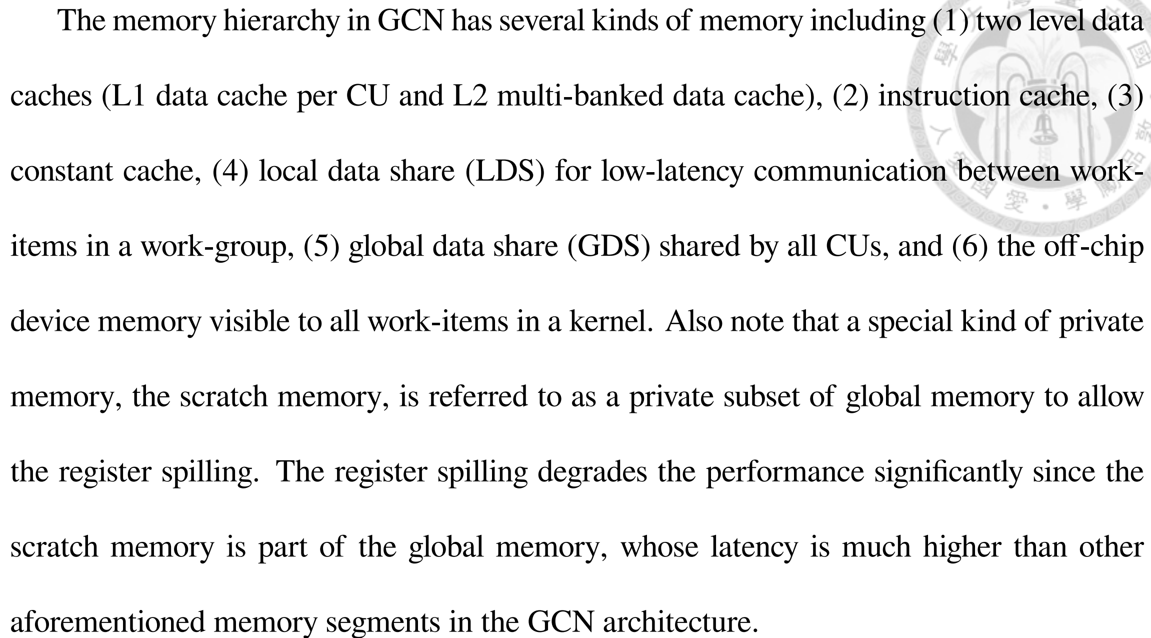


Figure 2.1: AMD GCN Generation 3 Series Block Diagram [3]

Before the tasks are executed, the GCN hardware divides the input threads into blocks of 64 threads called *wavefronts* and delegates these wavefronts to the CUs. A wavefront shares the program counter and is a schedule unit of the hardware, which implies that if the divergence paths are happened in the flow control, all 64 threads would go both ”if” and ”else” paths. Whether the thread should execute the vector instruction is determined by a hardware state, *EXECute Mask* (EXEC). If the corresponding bit of a thread in the specific wavefront in EXEC is 1, it should execute this vector instruction, or it would be idle. In GCN architecture, each vector SIMD processes 16 different data at a time (i.e., 16-wide SIMD), which makes a vector ALU operate one vector instruction on wavefronts of 64 work-items in four clock cycles [8].

In contrast, a scalar ALU processes the scalar instructions and operates on SGPRs which are common to a wavefront. These scalar instructions can also affect the control flow of the kernel program. Since the fact that one wavefront shares the values in the SGPRs, the scalar ALU can do an uniform calculation in a wavefront per cycle.

doi:10.6342/NTU201601752



The memory hierarchy in GCN has several kinds of memory including (1) two level data caches (L1 data cache per CU and L2 multi-banked data cache), (2) instruction cache, (3) constant cache, (4) local data share (LDS) for low-latency communication between work-items in a work-group, (5) global data share (GDS) shared by all CUs, and (6) the off-chip device memory visible to all work-items in a kernel. Also note that a special kind of private memory, the scratch memory, is referred to as a private subset of global memory to allow the register spilling. The register spilling degrades the performance significantly since the scratch memory is part of the global memory, whose latency is much higher than other aforementioned memory segments in the GCN architecture.

2.2 OpenCL Programming Model

OpenCL is an open standard for parallel programming of heterogeneous systems[14]. In OpenCL, the compute kernel is a function defined to be executed in parallel on OpenCL devices. It defines the *work-group* (WG) as a concept of the software level data-parallel granularity that packs up a fixed number of work-items, and requires it to be executed on the same compute unit. Once a WG is assigned to a CU, it is then divided into one or several wavefronts in the hardware perspective. Threads in the WG can share the data in *local memory*, which is corresponding to the LDS in the GCN hardware terminology.

It also defines four named device memory regions including global, constant, local, and private memory. They can be mapped to the corresponding memory segments of an OpenCL device like the GPU in an AMD Carrizo APU. Note that the global memory mentioned in the GCN architecture includes two types of memory, the GDS and the off-chip device memory, while the global memory in the OpenCL spec is the region that can be accessed by all work-items on an OpenCL context. Since OpenCL 2.0, the shared virtual memory

doi:10.6342/NTU201601752

(SVM) extends the global memory region into the host memory region, and this makes an OpenCL kernel to use the host memory or global memory on any OpenCL device directly.

OpenCL defines the runtime API for the vendors and an OpenCL program can use these API to develop the applications. Also, it defines the kernel code to be written in OpenCL C language which is based on C99 specification and adds some modifications like vector data types and operations to these new types, address space qualifiers to mark the disjoint memory regions on an OpenCL device, a set of built-in functions like kernel dispatch functions, math functions, etc.

2.3 The HSA Architecture

Heterogeneous System Architecture (HSA) is a standard to support a wide range of data-parallel and task-parallel programming models[11]. The requirements for the HSA-compliant system and participating devices are in several aspects including (1) SVM, flat memory addressing, and adherence to HSA memory model, (2) user level dispatch, (3) architected queuing language (AQL), (4) scheduling and preemptive kernel agent, (5) HSA Intermediate Language (HSAIL), and (6) a set of HSA runtime API for an HSA application uses the platform.

Note that HSA isn't an alternative of OpenCL but an optimized platform architecture for OpenCL. Thus, OpenCL on HSA can benefits from the SVM requirement of HSA, improved memory model, low latency kernel dispatch. The HSA requirements is more focused on the system and hardware and makes other high level language benefit from these requirements.

Additional high level language except OpenCL that will be supported by HSA are Java, C++AMP, OpenMP, hc, etc.

doi:10.6342/NTU201601752

2.3.1 HSAIL

Just as the OpenCL defines OpenCL C to explicitly mark the codes that can be executed in parallel on the compute units as the kernel functions, the HSA defines the HSAIL[12] to abstract away the underlying hardware ISA details and express the parallel region of the programs that can be executed on the compute units. The binary format of HSAIL is called BRIG. It is generated by a HLC like GCC[9] or LLVM.

The HSAIL programming model shares similar concepts of OpenCL execution model. It defines 136 instructions ranging from the basic arithmetic, memory, branch instructions to image-related, synchronization, function, and special instructions. These instructions look like the three-address code, support base, packed, array types, and vector operands on several instructions. For HSAIL registers, it has 4 types of width: 1, 32, 64, 128 bits. 1-bit (**c** register) is for condition code, 32 and 64-bit register (**s** and **d** register) support the floating point and integer data type. 32, 64, and 128-bit (the **q** register) can be used to store several smaller data type of equal size called the packed data type. For instance, a **s** register can hold `u8x4` packed data type, which packs up 4 elements of unsigned 8 bit integers.

Compared to the LLVM IR, the HSAIL is more low level and has quiet different features. For instance, the HSAIL has the fixed number of register usage while LLVM IR can use unlimited number of virtual registers. In specific, the HSA PRM defines the register usage equation, $((s_{max} + 1) + 2 * (d_{max} + 1) + 4 * (q_{max} + 1))$ mustn't exceed `2048`, or it is considered an invalid HSAIL module. As HSAIL provides a fixed-size register file, the HLC has to perform the register allocation, which is a time-consuming part in the backend code generation, and it may choose to spill the register to the *spill segment* even if the total register resource usage doesn't reach 2048 and let the finalizer promote these spills to device registers if the target has adequate register resources.

doi:10.6342/NTU201601752

Developers write the high level language like OpenCL or OpenMP, the HLC performs optimization passes based on the spec of HSA system architecture, and it emits the HSAIL/BRIG code. The HSA runtime API can call the HSA finalizer to finalize the HSAIL/BRIG code to the target ISA before running the HSAIL/BRIG module.

In this work, the author chooses the OpenCL as the target language, the Figure 2.2 shows the code generation on HSA platform for OpenCL. The work uses CLOC[6] to generate the HSAIL code from OpenCL.

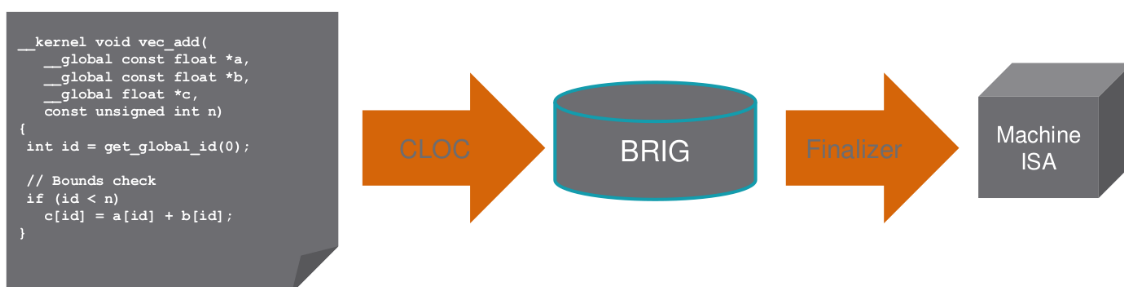


Figure 2.2: OpenCL Offline Compiler (CLOC) [1]

2.4 The LLVM Compiler Infrastructure

LLVM is an open source compiler framework that is started in University of Illinois. It defines the LLVM IR to represent a program from different high level language. The LLVM IR is then used for the input of a set of common optimization passes, and different back-ends can then generate device ISA. This three-phase design follows the advantage of retargetablity. CLOC is a living example of the retargetability, as it generates the HSAIL code by developing only a new backend, *HSAIL*, in the `lib/Target/HSAIL` directory of the LLVM source. It can leverage the existing frontend, optimization passes, register allocation (RA) schemes, instruction scheduling algorithms, etc. in the LLVM codebase.

doi:10.6342/NTU201601752

2.4.1 Clang and libclc

For OpenCL language, the LLVM native C-family frontend *clang* can process the OpenCL C language with the help of another project *libclc*[15] that implements the standard OpenCL C language requirement. Currently, not all the requirements of OpenCL C 2.0 are implemented in the clang+libclc.

2.4.2 llc and AMDGPU backend

The LLVM llc is the static compiler in the LLVM infrastructure. It supports several backends, such as x86, ARM, AArch64, Mips, NVPTX, or AMDGPU. The llc static compiler is responsible for lowering the LLVM IR to target ISA through several representations in the memory including `SelectionDAG`, `MachineInstrs`, `MCInstr`.

As HSAIL is another backend for LLVM-based CLOC compiler, AMDGPU is the backend in LLVM targeting AMD GCN architecture. Note that in CLOC, there is a path that directly generates the code from OpenCL to target ISA without going through the HSAIL.

2.5 OpenCL Kernel Code Execution and Compilation on a HSA-Compliant System

Three OpenCL kernel code compilation processes are introduced in the subsection 2.5.1. With the knowledge of kernel compilation (or finalization), the way of OpenCL kernel code execution is discussed in 2.5.2.

doi:10.6342/NTU201601752

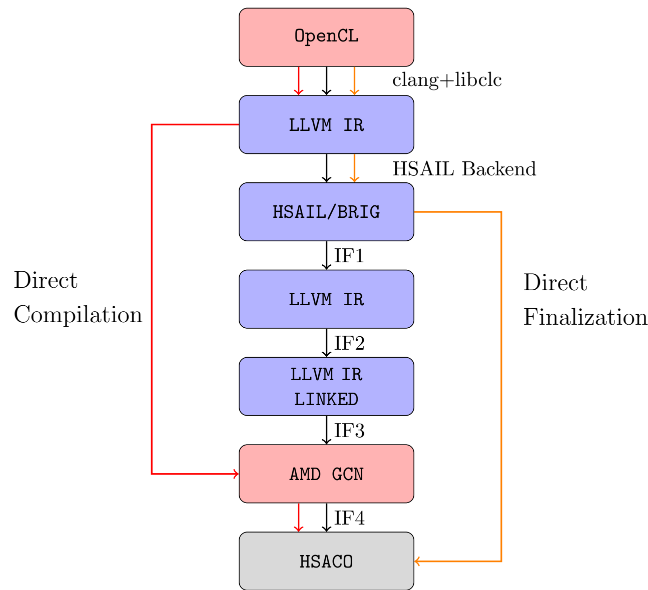


Figure 2.3: OpenCL Kernel Code Compilation Processes. There are three different paths to generate a code object containing GPU device code. The Direct Compilation (in red path) is provided as the default option since CLOC version 1.0. The Indirect Finalization is shown in the black path. The Direct Finalization is shown in the orange path.

2.5.1 Kernel Code Compilation/Finalization Process

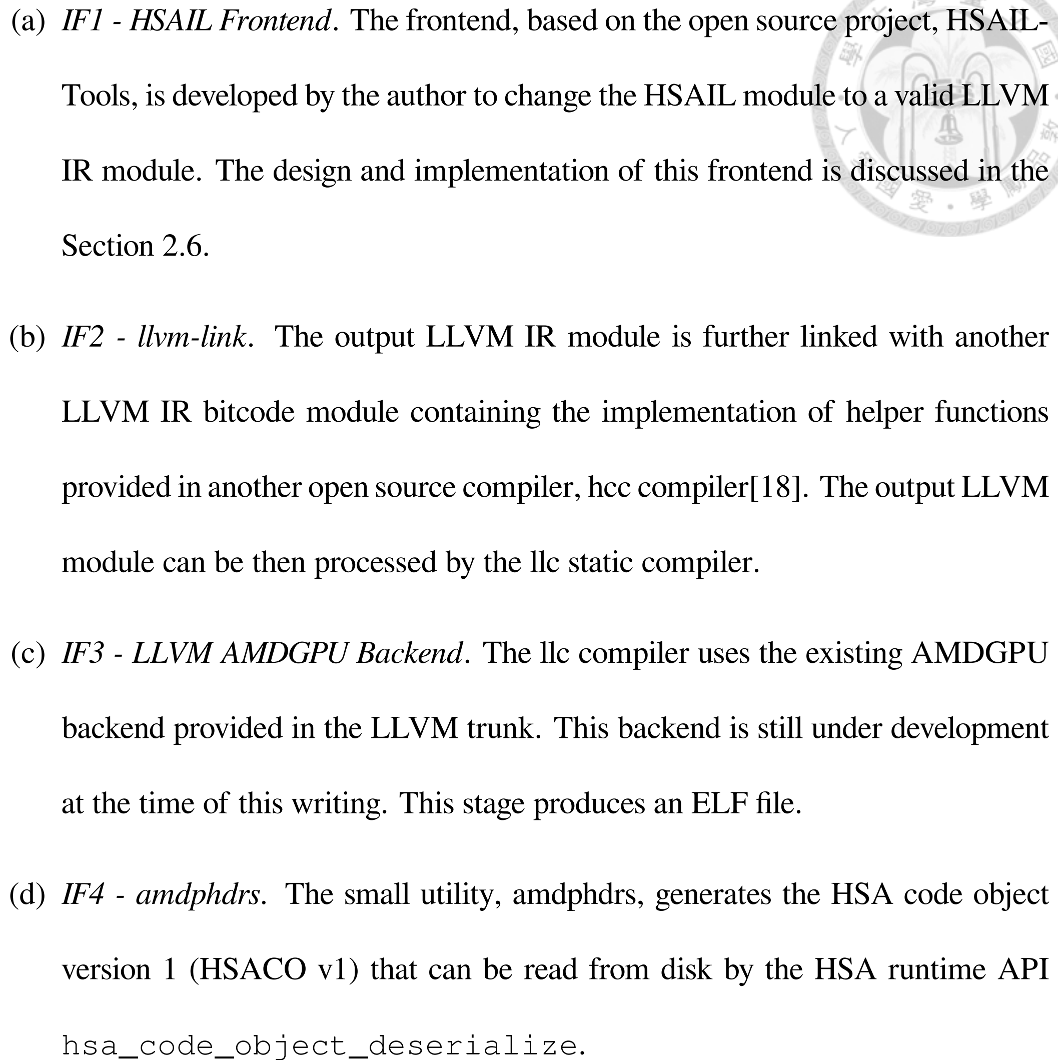
The Figure 2.3 shows the compilation and finalization processes of an OpenCL kernel to GCN device code, including the following:

1. *Direct Finalization*. AMD provides the closed source finalizer that can be called on the runtime to finalize a given BRIG module to AMD GPU code. The HSA Runtime API is

   `hsa_ext_program_finalize`[13], which creates a code object handle in the memory.

2. *Indirect Finalization*. The indirect finalization is the path that leverages the existing LLVM compiler infrastructure to generate the AMDGPU code. There are four steps denoted by `IFN - Step Name`, where N ranges from 1 to 4, including the following:

doi:10.6342/NTU201601752



(a) *IF1 - HSAIL Frontend*. The frontend, based on the open source project, HSAIL-Tools, is developed by the author to change the HSAIL module to a valid LLVM IR module. The design and implementation of this frontend is discussed in the Section 2.6.

(b) *IF2 - llvm-link*. The output LLVM IR module is further linked with another LLVM IR bitcode module containing the implementation of helper functions provided in another open source compiler, hcc compiler[18]. The output LLVM module can be then processed by the llc static compiler.

(c) *IF3 - LLVM AMDGPU Backend*. The llc compiler uses the existing AMDGPU backend provided in the LLVM trunk. This backend is still under development at the time of this writing. This stage produces an ELF file.

(d) *IF4 - amdphdrs*. The small utility, amdphdrs, generates the HSA code object version 1 (HSACO v1) that can be read from disk by the HSA runtime API `hsa_code_object_deserialize`.

3. *Direct Compilation*: the direction compilation is the new default path of CLOC since version 1.0 which directly generates the HSACO without the HSAIL involved. As its name (compilation) suggests, this isn't the finalization. However, the code from this compilation path can be compared with that from other two paths.

2.5.2 Kernel Code Execution

There are two ways of developing a HSA host program given a GPU kernel on a HSA-compliant system. One way is to use the a set of low-level HSA runtime API, which needs many lines of code to program on the host side, while the other is to use the SNACK API provided as an AMD extension in CLOC project.

doi:10.6342/NTU201601752

SNACK (Simple No Api Compiled Kernels) uses CLOC to help the host program launch GPU kernels as host-callable functions with structured launch parameters. CLOC takes care of the code generation of other structures, header files, etc. in the SNACK API. Thus, the programmer writes less code than directly uses HSA runtime API.

In this work, the author develops eight benchmark using the HSA runtime API, and the eight OpenCL kernel code is obtained from AMD APP SDK suite[5][4]. The author tries to wrap the low-level HSA runtime API into a set of helper functions.

2.6 HSAIL Frontend

In the process of indirect finalization, the first step is to generate a valid LLVM module given a input HSAIL module. It's necessary to develop a frontend to do the work. In this section, the author presents the design and implementation of the HSAIL frontend, which is based on the HSAIL-Tools[2]. The Section 2.6.1 first introduces the HSAIL frontend and the rest of this section discusses the design issues of the frontend.

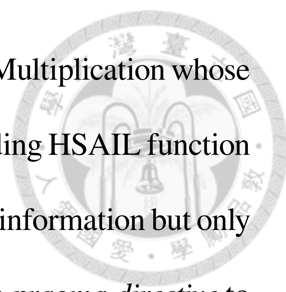
2.6.1 HSAIL-Tools and HSAIL Frontend

HSAIL-Tools is used for parsing, assembling, and disassembling HSAIL. Based on the functionality in this open source project, the author extends the top-down parser to generate the valid LLVM IR module.

2.6.2 Design and Implementation of the HSAIL Frontend

To discuss the implementation issues in the frontend, the author uses some code snippet from the matrix multiplication HSAIL module as a running example.

doi:10.6342/NTU201601752



1. *Additional Information in Pragma*: for an OpenCL kernel MatrixMultiplication whose function signature is declared in Code Snippet 2.1, the corresponding HSAIL function declaration is in Code Snippet 2.2. As we can see, there is no type information but only the argument length in the formal parameters. The HLC uses the *pragma directive* to pass the information to the finalizer or other components that process HSAIL. For the pragma string that can't be recognized by the HSAIL consumer, it must be ignored by the consumer and shouldn't cause any errors.

```
1 __kernel void mmmKernel(
2     __global float4 *matrixA,
3     __global float4 *matrixB,
4     __global float4* matrixC,
5     uint widthA, uint widthB);
```

Code Snippet 2.1: OpenCL Matrix Multiplcation Function Header

```
1 module &mm:1:0:$full:$large:$default;
2 extension "amd:gcn";
3 extension "IMAGE";
4 prog kernel &mmmKernel(
5         kernarg_u64 %__global_offset_0,
6         kernarg_u64 %__global_offset_1,
7         kernarg_u64 %__global_offset_2,
8         kernarg_u64 %__printf_buffer,
9         kernarg_u64 %__vqueue_pointer,
10        kernarg_u64 %__aqlwrap_pointer,
11        kernarg_u64 %matrixA,
12        kernarg_u64 %matrixB,
13        kernarg_u64 %matrixC,
14        kernarg_u32 %widthA,
15        kernarg_u32 %widthB) { ... }
```

Code Snippet 2.2: HSAIL Matrix Multiplication Function Header

Code Snippet 2.3 gives the some pragma directive emitted by CLOC. The line 1 and 8 of pragma mark the starting and ending lines with the string `ARGSTART` and `ARGEND` followed by the kernel function name, respectively.

The line 2 gives the private memory usage of HSAIL module. The private memory includes both the spill stack and private stack since they will be put in the scratch memory on the GPU. The line 3 shows that the type of the argument `__global_offset_0` is a value, and the actual type information in the OpenCL kernel code is given in the line 6. The `0` in line 6 indicates the argument index. Thus, the HSAIL frontend could conclude that `__global_offset_0` is a value of type `size_t` of 32-bit. In a

doi:10.6342/NTU201601752

similar way, the HSAIL frontend could infer that `matrixA` is a 64-bit `float4*` pointer type.

With the knowledge of the type information, the frontend could emit better LLVM IR instructions, such as the LLVM `getelementptr` instructions.

```
1 pragma "AMD RTI", "ARGSTART:__OpenCL_mmmKernel_kernel";
2 pragma "AMD RTI", "memory:private:144";
3 pragma "AMD RTI", "value:__global_offset_0:u64:1:1:0";
4 pragma "AMD RTI", "pointer:__printf_buffer:u8:1:1:48:uav:7:1:RW:0:0:0";
5 pragma "AMD RTI", "pointer:matrixA:float:1:1:96:uav:7:16:RW:0:0:0";
6 pragma "AMD RTI", "reflection:0:size_t";
7 pragma "AMD RTI", "reflection:6:float4*";
8 pragma "AMD RTI", "ARGEND:__OpenCL_mmmKernel_kernel";
```

Code Snippet 2.3: An Example of Pragma Directive

2. *HSAIL State Mapping*: the architecture state mapping of four types of registers is quite straight-forward. When the frontend encounters a new virtual register that is unused before, say register `$d0`, it allocates a stack variable, which has the same name as the virtual register, in the entry basic block by using the LLVM `alloca` instruction, and its type is like the union type in the C language. The reason why the frontend allocates the union type, as mentioned in section 2.3.1, is that the `s` and `d` registers can hold different types so they are untyped.

   How an instruction treats the bytes in the specific register can be found in the instruction mnemonic. For instance, an instruction `add_u64 $d0, $d0, $d1` implies that two 64-bit registers `$d0` and `$d1` are treated as unsigned 64-bit integers. The translation of this instruction could be found in Code Snippet 2.4.

   For the `add_u64` instruction, the frontend encounters two registers `d0` and `d1`.

   It finds that they are first used in this module, and allocates two 64-bit variables of type `type{ i64 }` in line 4–5. Then, it determines the expected type to be unsigned 64-bit integer from the instruction mnemonic, does the `bitcast` to convert the pointer

doi:10.6342/NTU201601752

type to i64*, load the values from the memory locations, add two values, and finally store the result to the desired memory location. Note that in the legitimate HSAIL module, the two registers should be first initialized, or the result of add_u64 are undefined. This implies the two alloca instructions in line 4–5 will be happened before this HSAIL add_u64 instruction.

To sum, the HSAIL frontend translation logic doesn't complicate the state mapping since the redundant instructions, such as the bitcast instruction in line 11, can be removed by the built-in LLVM optimization passes.

```
1  ; HSAIL instruction: add_u64 $d0, $d0, $d1;
2  %union.hsailSregister = type { i32 }
3  %union.hsailDregister = type { i64 }
4  %d0 = alloca %union.hsailDregister
5  %d1 = alloca %union.hsailDregister
6  %1 = bitcast %union.hsailDregister* %d0 to i64*
7  %2 = load i64, i64* %1
8  %3 = bitcast %union.hsailDregister* %d1 to i64*
9  %4 = load i64, i64* %3
10 %5 = add i64 %2, %4
11 %6 = bitcast %union.hsailDregister* %d0 to i64*
12 store i64 %5, i64* %6
```

Code Snippet 2.4: LLVM IR Translation of an add_u64 Instruction

3. *HSAIL Address Expressions and LLVM GEP Instructions*: there are two kinds of address in the HSAIL, flat and segment address. The flat address is a general address that can be used to address any HSAIL memory, while the segment address is just an offset within a segment, and the segment is specified in the instruction.

   In all testing benchmarks, the HLC generates segment addresses and thus the HSAIL frontend now supports only the segment address. A typical segment address expression, such as [$d2], in the memory instructions is shown in line 11 of Code Snippet 2.5. As we can see, the offset is pre-computed and stored in the register $d2.

   For the HSAIL frontend, this would be translated to a so called ptrtoint-inttoptr pattern, which is unfriendly to other LLVM optimization passes. When the fron-

doi:10.6342/NTU201601752

tend translates the `ld_kernarg` instruction in line 6, it stores the base address of `%matrixB` in the register `%d1` after making use of `ptrtoint` instruction to convert the pointer `%matrixB` to an `i64` value. After doing some address calculation and storing the results in `$d2`, the frontend would use `inttoptr` to convert the content in `$d2` to a pointer of LLVM vector type `<4 x f32>`. The corresponding LLVM IR code is shown in Code Snippet 2.6.

Although the way of this `ptrtoint-inttoptr` pattern calculates the address offset is similar to LLVM `getelementptr (GEP)` instruction, this however causes other LLVM optimization passes, such as the pointer aliasing analysis, to be conservative and prevents the other optimizations from doing the transformation. Thus, one of the targets in the frontend is to prefer the generation of GEP instructions.

```
1  // float4 temp = matrixB[get_global_id(0)]
2  workitemabsid_u32        $s0, 0;
3  cvt_u64_u32     $d0, $s0;
4  ld_kernarg_align(8)_width(all)_u64      $d1, [%__global_offset_0];
5  add_u64 $d2, $d0, $d1;
6  ld_kernarg_align(8)_width(all)_u64      $d1, [%matrixB];
7  shl_u64 $d2, $d2, 32;
8  shr_s64 $d2, $d2, 32;
9  shl_u64 $d2, $d2, 4;
10 add_u64 $d2, $d1, $d2;
11 ld_v4_global_align(16)_f32      ($s9, $s12, $s8, $s0), [$d2];
```

Code Snippet 2.5: A Example of Simplified Address Expression

```
1  %tmp8 = ptrtoint float addrspace(1)* %matrixB to i64
2  ; some offset calculation and store it in %tmp14
3  %tmp15 = inttoptr i64 %tmp14 to <4 x float> addrspace(1)*
4  %tmp16 = load <4 x float>, <4 x float> addrspace(1)* %tmp15
5  ; extractelement 4 times and store the value into the corresponding memory location
6  %tmp17 = extractelement <4 x float> %tmp16, i32 0
7  %s9 = alloca %union.hsailSregister
8  %tmp18 = bitcast %union.hsailSregister* %s9 to float*
9  store float %tmp17, float* %tmp18
10 ...
```

Code Snippet 2.6: The Simplified LLVM IR Translation of Code Snippet 2.5

In the implementation, the frontend would prefer to generate the GEP instructions in some common cases. For instance, Code Snippet 2.7 shows a case of a register spilling, and its corresponding LLVM IR translation emitted by the frontend is given in

doi:10.6342/NTU201601752

Code Snippet 2.8. The frontend can determine the stack offset is `124` when processing the HSAIL instruction `st_spill` in line 3 of Code Snippet 2.7, so it emits the `getelementptr` instruction in line 7 of Code Snippet 2.8 that uses $31(= 124/4)$ as the second operand.

```
1 align(4) spill_u8 %__spillStack[144];
2 // put the result in $s2
3 st_spill_align(4)_u32   $s2, [%__spillStack][124];
```

Code Snippet 2.7: Prefer GEP Instruction

```
1 %union.hsailSregister = type { i32 }
2 %__spillStack = alloca i8, i32 144, align 4
3 %tmp1 = bitcast %union.hsailSregister* %s2 to i32*
4 %tmp2 = load i32, i32* %tmp1
5 %tmp3 = ptrtoint i8* %__spillStack to i64
6 %tmp4 = inttoptr i64 %tmp3 to i32*
7 %tmp5 = getelementptr i32, i32* %tmp4, i64 31
8 store i32 %tmp2, i32* %tmp5
```

Code Snippet 2.8: LLVM IR Translation of Code Snippet 2.7

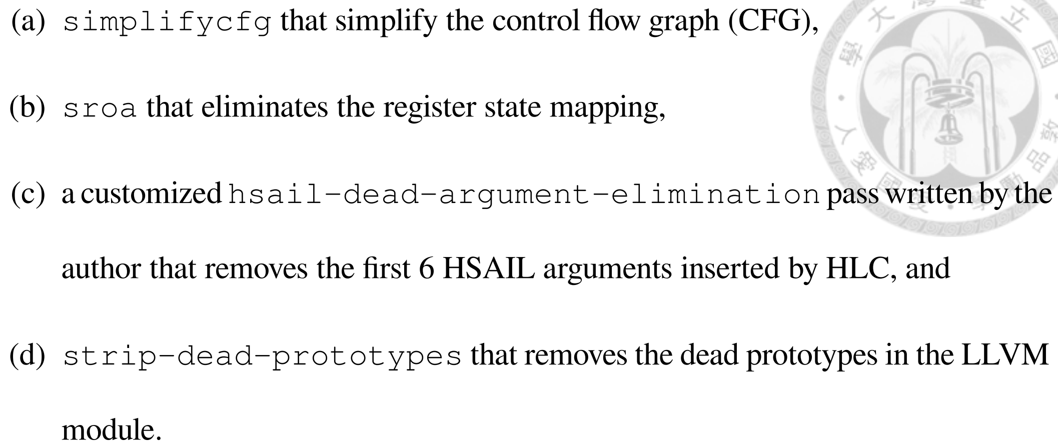
In sum, the frontend should prefer the GEP instructions to the "`ptrtoint-inttoptr` pattern" in order to trigger the other optimization passes.

4. *Preliminary LLVM Optimizations in the Frontend*: the naïve implementation of HSAIL frontend makes sure that each HSAIL instruction is mapped to a sequence of LLVM IR instructions. The raw LLVM module emitted by the frontend has rooms for optimizations. For instance, the frontend maps the HSAIL registers to the stack, which can be promoted to LLVM virtual registers.

   Thus, the frontend integrates some preliminary LLVM optimizations, including three selected passes and one low-cost, customized optimization pass in order to improve the code quality before the AMDGPU backend processes the module.

   These developer-selected passes including:

doi:10.6342/NTU201601752



(a) `simplifycfg` that simplify the control flow graph (CFG),

(b) `sroa` that eliminates the register state mapping,

(c) a customized `hsail-dead-argument-elimination` pass written by the author that removes the first 6 HSAIL arguments inserted by HLC, and

(d) `strip-dead-prototypes` that removes the dead prototypes in the LLVM module.

The first two passes are managed by LLVM function pass manager and run in the function scope, while the other two passes are managed by LLVM module pass manager and run in the module scope. The frontend first runs the function passes for each function, and apply the other two module passes for the module. The pass order is the same as the listing order above.

5. *Other Issues in the HSAIL Frontend*: when the author designs and implements the frontend, several issues worth mentioning are as follows.

   (a) *Register Allocation in the HLC*. The HLC performs the register allocation, but the implementation of state mapping in the frontend chooses to map it to the stack memory, which is then promoted to the LLVM virtual registers. The process of target language register allocation is done again by LLVM infrastructure. In other finalizer implementations, it may simply map the HSAIL registers to the target registers or develop other more efficient state mapping algorithms to solve the problem. The author believes that there is no known work that encodes the register allocation information in other intermediate language in the LLVM IR level. Encoding the register information in LLVM IR is not attractive to the author because in LLVM IR, it has unlimited registers and the other existing optimization passes could effectively reduce the usage of memory stack from the

doi:10.6342/NTU201601752

register state mapping. Thus, the author chooses to map the HSAIL architecture states in the memory stack and integrate other passes to eliminate the stack.

(b) *Preliminary Optimization Passes and Order*. The frontend selects fixed optimization passes by heuristic without considering the fact that different programs should have different program features, and need different frontend optimizations. However, the pass order and important optimization passes are well-known research problems. The goal of HSAIL frontend is to ensure the correctness of the code and apply some effective and general optimizations to eliminate the overhead in the frontend code generation, such as the stack access. The optimization passes and order aren't the problem the author addresses in this work.

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

3 Experimental Methodology

In this section, the author presents the environment setup and experimental methodology in Section 3.1. Section 3.2 provides the information for these benchmarks, the host program development, and kernel modification for the HSA platform.

Table 3.1 describes the various HSA applications, and their characteristics. These kernels are taken from AMD APP SDK[4][5]. Since the HSA applications use the HSA runtime API on the host side, the author takes only the GPU kernel and develops the host program using the HSA runtime API. The system configuration is shown in Table 3.2. The Appendix A.1 gives the software version on the HSA platform.

Application	Input Data Size	GlobalWorkSize	WorkGroupSize
AESEncrypt	3840x2160 BMP Picture	{960, 2160, 1}	{64, 4, 1}
BinomialOption	1048576	{66846720, 1, 1}	{255, 1, 1}
BitonicSort	4194304	{2097152, 1, 1}	{512, 1, 1}
BlackSholesDP	8388608	{1280, 1280, 1}	{16, 16, 1}
MatrixMultiplication	4096x4096	{1024, 1024, 1}	{16, 16, 1}
MonteCarloAsianDP	256 steps	{256, 512, 1}	{256, 1, 1}
RadixSort	65536	{65536, 1, 1}	{256, 1, 1}
SimpleConvolution	8192x8192	{67108864, 1, 1}	{256, 1, 1}

Table 3.1: Benchmark Description

APU (CPU+GPU)	Carrizo APU (AMD FX-8800P)
CPU (Part of APU)	4-Core Excavator@2.1GHz-3.4 GHz
GPU (Part of APU)	8 CUs. GCN 1.2 (Volcanic Islands) Radeon R7@300-800MHz
Memory	8GB
OS/Kernel	Ubuntu 14.04.4 / 4.4.0-kfd-compute-rocm-rel-1.1-15

Table 3.2: System Environment

doi:10.6342/NTU201601752

3.1 Methodology

For each time measurement, the author repeats the experiments independently for eight times and takes the average of results. Experiments in this work are as follow.

1. *Kernel Finalization and Compilation Time*. In this experiment, the author compares the compilation time of kernel code in three different paths: direct compilation, indirect finalization, and direct finalization, as mentioned in Figure 2.3.

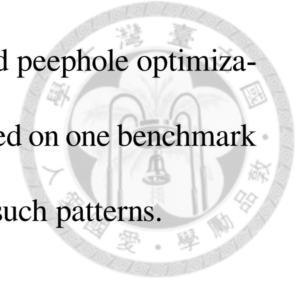
   For direct finalization, the BRIG code is in memory and only the HSAIL code finalization is measured while the indirect finalization and direct compilation have the I/O time included in the measurement. The impact of I/O time will be alleviated when the evaluation is repeated because the input file is in the disk cache.

2. *Kernel Execution Time*. In this experiment, the kernel execution time of eight benchmarks is measured given the fixed input parameter listed in Table 3.1. The measurement excludes the initialization before the first kernel launch. For four of the benchmarks, *BitonicSort, MonteCarloAsianDP, RadixSort, and SimpleConvolution*, which issue multiple kernel launches, the measurement of these benchmarks includes the data preparation and marshaling time on the host side among 2nd to the last kernel launches.

3. *LLVM Optimization Options*. The two experiments related to LLVM optimization passes are carried out to reduce the performance gap between indirect and direct finalization.

   The first experiment is to measure the impact of different instruction scheduling schemes. The author changes both the pre-RA and post-RA instruction scheduling.

doi:10.6342/NTU201601752



The second experiment is to measure the impact of a customized peephole optimization developed by the author. The optimization pass is only applied on one benchmark `BinomialOption` since the other seven benchmarks has no such patterns.

3.2 Benchmarks Development

This section briefly introduces the development of these HSA benchmarks and encountered issues.

1. *Host Program Development.* The host program is written in the HSA runtime API wrapper functions developed by the author. The process isn't that trivial when it comes to properly setup the *kernel arguments* since the argument padding behavior is different in code objects from open source compiler and closed source finalizer. That is, there are two different kernarg segment sizes in code objects given the same GPU kernel. This behavior makes the host program need to be carefully checked and error-prone. The developer must take care of the subtle difference by check the *kernel argument segment size* and write simple kernel to make sure that every kernel argument can be accessed correctly.

   In terms of local memory on the kernel arguments, there is no working examples and no tutorial on how to set up correctly on the host side until the issue is raised and answered on the repository.

   The setup of AQL packet is designed to be simple but the error is silent. Developers are responsible to make sure that every single bit in the AQL packet is setup correctly. When any value in the AQL packet is wrong, the host gets wrong results without any hint. Even worse, a simple AQL packet dispatch can make the whole system crash.

   Last but not least, the LLVM backend is a rough path and under development. For

doi:10.6342/NTU201601752

instance, the developer has no idea if each system upgrade passes the regression test, which makes some working kernels give the wrong answer after upgrading the compiler toolchain. Another example is that some kernel code need to be modified in order to work-around the existing backend problems. To sum, the unstable system environment could significantly make the development time longer.

2. *Kernel Program Modification*. As stated above, the LLVM backend is under development. Some encountered compiler issues include the lack of support to some cases in OpenCL generic address casting and the structure pass-by-value in global memory. The developer has to modify the kernel code to work-around these compiler issues. In this work, the modified kernel is BlackSholesDP and MonteCarloAsianDP.

doi:10.6342/NTU201601752

4 Results and Analysis

In this section, the thesis evaluates three aspects including (1) compilation time of indirect and direct finalizations, (2) the execution time of HSA benchmark suite mentioned in Section 3 with the GPU kernel code from different paths, (3) the impact of different instruction schedulers, and (4) the impact of a peephole optimization pass by the author. The results are shown in Section 4.1, 4.2, and 4.3 respectively.

4.1 Kernel Finalization Time

The Figure 4.1 shows the kernel finalization time of indirect finalization versus direct finalization.

The direct finalization by AMD finalizer is normalized to 1, while the stacked bar on the right hand side shows the normalized time of each step in indirect finalization.

Recall that the indirect finalization consists of four steps, IF1 to IF4. As the figure shows, the first step (IF1-HSAIL frontend) as well as the third step (IF3-LLVM AMDGPU Backend) take the majority of time in the indirect finalization, which takes for 61.8 to 96.5 percent of finalization time.

All benchmark except *BitonicSort* takes much more finalization time in the process of indirect finalization. For *BitonicSort*, the kernel code length is short, and the AMD finalizer applies other vendor-specific optimizations which takes more time than the optimizations in

doi:10.6342/NTU201601752

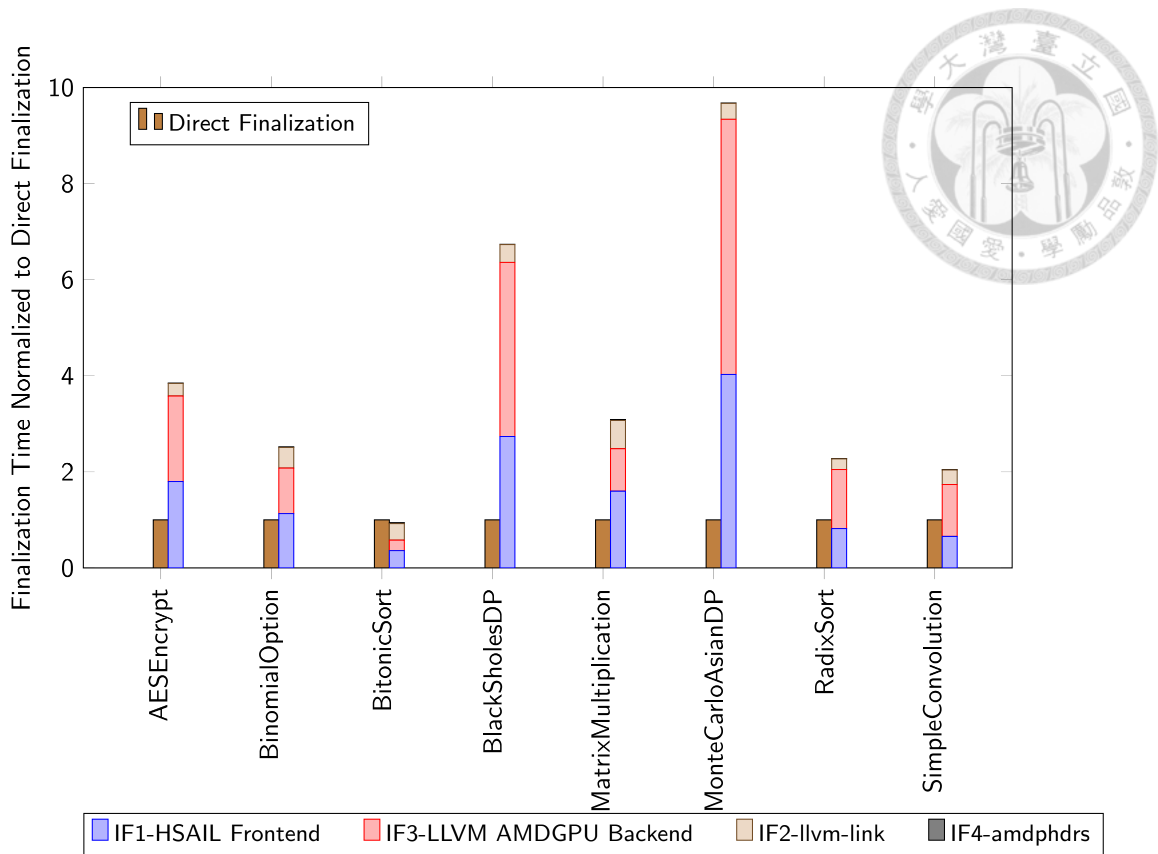


Figure 4.1: Kernel compilation of Indirect and Direct Finalization. The number followed by IF designate the order in Indirect Finalization, and it's followed by the name of step. For example, the blue stacked y bar IF1-HSAIL Frontend shows the normalized time of the first step of indirect finalization.

indirect finalization. In *MonteCarloAsianDP*, it takes the indirect finalization 8.7x more time to generate the AMDGPU device code.

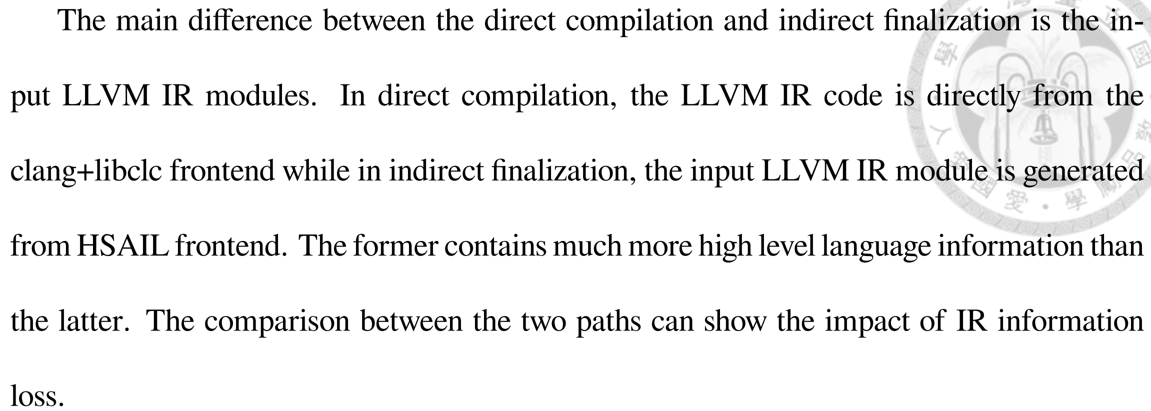
It follows that the direct finalizer generates the code faster than the indirect finalizer.

4.2 Kernel Execution Time

The Figure 4.2 gives the measurement of the normalized kernel execution time of three different compilation paths.

The results contain three bars, indirect finalization, direct compilation, and direct finalization. Note that Figure 4.2 introduces direct compilation, another code compilation path provided as the default option since CLOC version 1.0.

doi:10.6342/NTU201601752



The main difference between the direct compilation and indirect finalization is the input LLVM IR modules. In direct compilation, the LLVM IR code is directly from the clang+libclc frontend while in indirect finalization, the input LLVM IR module is generated from HSAIL frontend. The former contains much more high level language information than the latter. The comparison between the two paths can show the impact of IR information loss.

4.2.1 Comparison between Indirect and Direct Finalization

The results show that the code of direct finalization outperforms the counterparts of the other paths in all benchmarks except *MatrixMultiplication and BitonicSort.* This implies that the closed source finalizer can generate more competitive code quality compared to the open source solution.

For *BitonicSort*, all three compilation paths have almost the same performance because the tiny kernel code and the compiler can't help much in terms of irregular memory access pattern.

For *MatrixMultiplication*, the finalizer generates worse code which is about 35 percent slower than the open source counterparts. This results from the memory load instructions reordering, which is the transformation performed by HLC at the time when the device ISA is unknown.

Figure 4.3 shows the performance of the manually reordered HSAIL code to support the aforementioned observation. The result shows that without the misleading information passed by HLC, the AMD finalizer emits more optimized code.

The reordering of load instructions doesn't take the hints in the source code into consideration. That is, the ordering written by the programmer has taken the advantages of multiple work-items load in the kernel grid can coalesce the memory transaction and these

doi:10.6342/NTU201601752

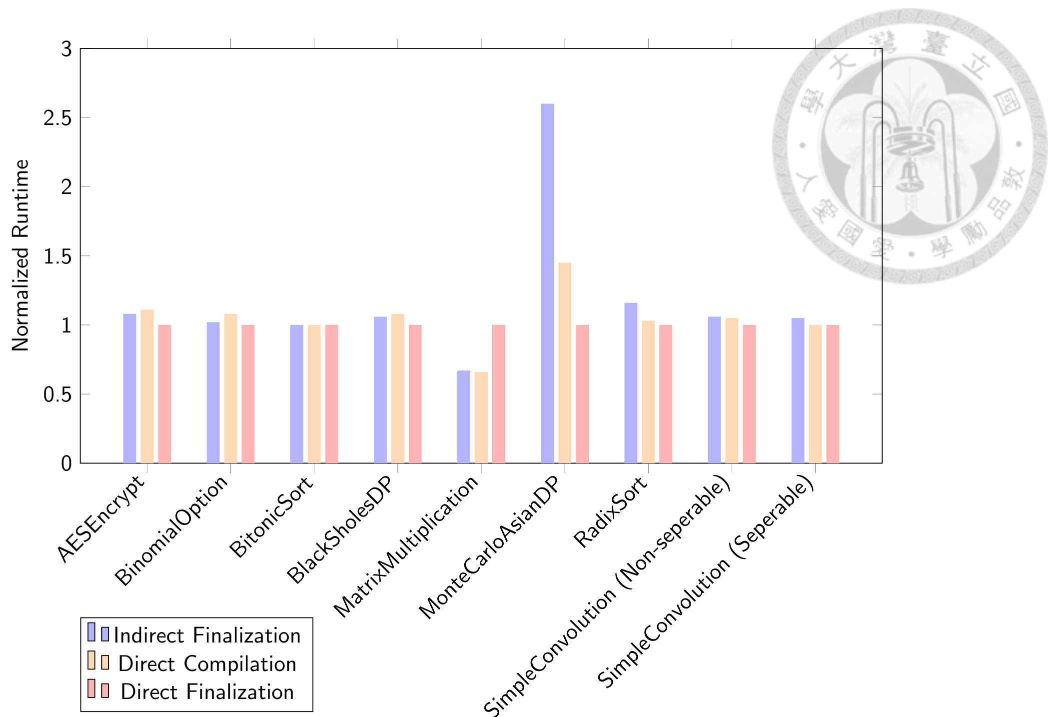


Figure 4.2: Kernel Execution Time. The direct compilation is the path which translates OpenCL to LLVM IR, which then directly translated to GCN assembly code and packed to HSA Code Object (HSACO) v1. Note that this is however not part of the finalization but is added for comparison of the other two alternatives.

four vector loads in each work item would access the global memory contiguously, which is friendly to hardware. The AMD closed source finalizer takes the sub-optimal HSAIL code generated by HLC, and emits the GPU code that fetches global memory almost 98 percent more than that of the open source solution. The specific *FetchSize* HSA performance counter (hsapmc) is obtained in AMD ROCm-Profiler[7]. This demonstrates a case that passing the misleading information by HLC causes the performance to degrade significantly. Although most optimizations are intended to be performed on the HLC, the finalizer should have some chances to optimize the code which makes the finalizer more than just a translator. It is arguable that the finalizer would take more effort to carry out high level code optimizations after a HLC implementation fails to recognize the optimization patterns or applies the wrong transformations (just like the load instruction reordering) because the information loss in low-level HSAIL.

doi:10.6342/NTU201601752

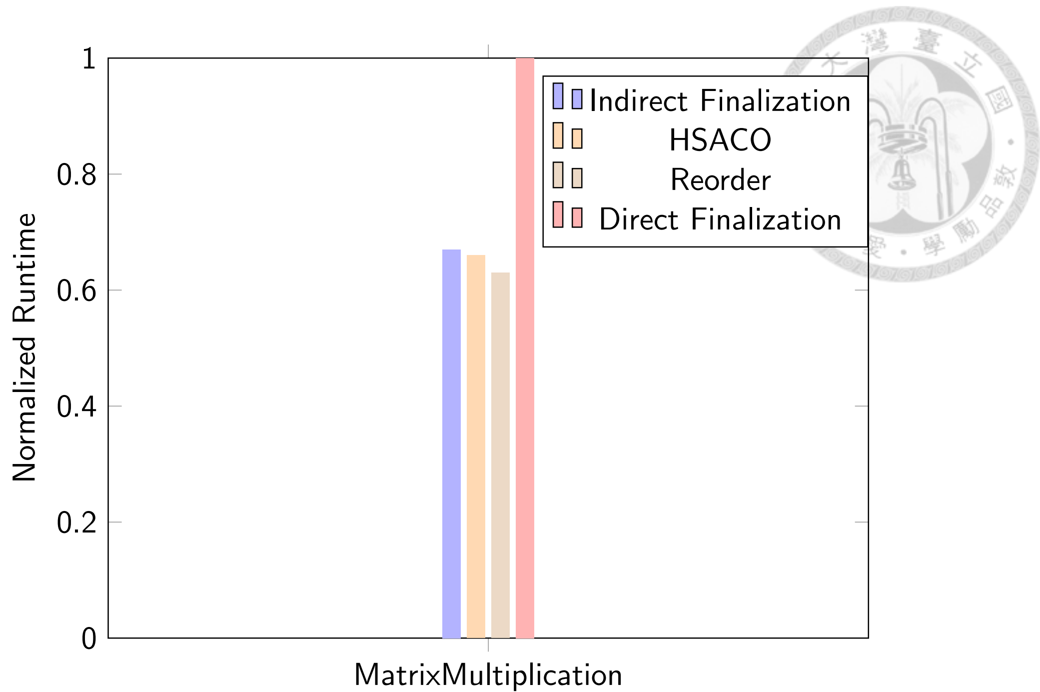


Figure 4.3: The performance of MatrixMultiplication with manually reordered HSAIL code shown on *Reorder*. The author modifies the HSAIL to eliminate the impact of load instruction reordering made by the high level compiler. The result shows that the AMD finalizer can emit the code slightly better than the other open source alternatives (*Indirect Finalization* and *Direct Compilation*). The original HSAIL input is shown on the *Direct Finalization*.

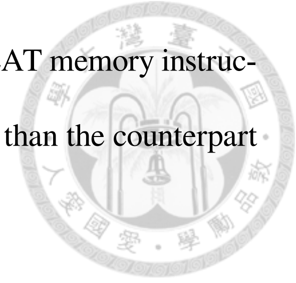
4.2.2 Comparison between Indirect Finalization and Direct Compilation

This subsection compares the code from indirect finalization and direct compilation. Both paths leverage existing LLVM AMDGPU backend, but the input LLVM IR code is very different. In the indirect finalization, the input LLVM module is emitted from the HSAIL frontend developed by the author, while the LLVM module in direct compilation is generated from clang+libclc.

The results show that in four benchmarks, *MatrixMultiplication, MonteCarloAsianDP, RadixSort, SimpleConvolution*, the direct compilation generates better code.

For *MatrixMultiplication*, the performance difference is about 1 percent since the device codes are similar. In terms of the machine code length, number of used SGPRs and VGPRs, they share the same number. While in terms of instructions, the ROCm-Profiler shows that

doi:10.6342/NTU201601752



indirect finalization generates less vector ALU instruction but more FLAT memory instructions than the counterpart, while the cache hit rate is 0.25 percent less than the counterpart (74.30 percent vs. 74.05 percent).

For *RadixSort*, while the performance difference is about 10 percent (but just 1.5 milliseconds since the kernel execution time is short), this is the accumulation of 28 kernel launches of seven different kernels. In one kernel *permute*, the author finds that direct compilation path unrolls the loop which accounts for 2 percent faster than the indirect finalization. While in the indirect finalization, the corresponding loop in HSAIL module isn't unrolled, and the indirect translation fails to recognize the pattern for loop unrolling. The other 24 kernel launches have tiny difference including instruction length and register numbers, which account for 8 percent performance difference.

For *MonteCarloAsianDP*, the indirect finalization produces the code that is 1.15x slower than the direct compilation one. The root cause is that the HLC allocates `%__spillStack` of size 508 bytes and `%__privateStack` of size 224 bytes for spill segment and private segment, respectively. Recall that in the HSAIL virtual machine, the spill segment can be used to load or store register spills, and the private segment can be used to hold variables that are local to a single work-item. For the HSAIL frontend, these stacks are translated into the `alloca` instructions in the LLVM IR code. What's more, some of the address calculations of these HSAIL memory load/store instructions with respect to these two memory stacks would be further translated to *ptrtoint-inttoptr* pattern mentioned in Section 2. The pattern hinders the LLVM optimizations and causes a backend *promotealloca* pass to fail to promote the additional alloca to registers, so indirect finalization generates lots of global memory load/store to handle the register spilling. In contrast, the direction compilation has the input LLVM IR from clang+libclc, and the input doesn't contain the additional segment hints from HLC. It puts the results in the unlimited LLVM virtual registers, use `getelementptr`

doi:10.6342/NTU201601752

instructions to calculate the memory locations of load/store instructions. Another main difference is that the LLVM IR module doesn't go through the register allocation and left for the AMDGPU backend to go through the code generation process. The distinct characteristics of input LLVM IR code makes the AMDGPU backend work better in this case without the additional information passed by the HLC.

For BitonicSort, the reason why they have similar performance is the same as the reason mentioned in the Subsection 4.2.1.

For the other benchmarks, *AESEncrypt, BinomialOption, BlackSholesDP*, indirect finalization generates slightly better code than the direct compilation does ranging from 2 percent to 6 percent.

4.3 LLVM Optimization Options

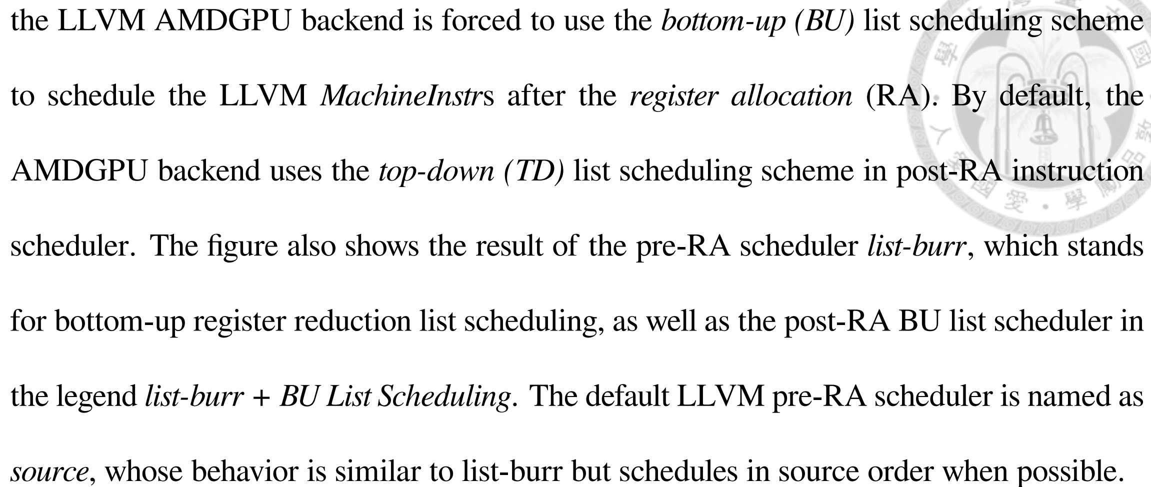
In this section, the author shows that (1) instruction scheduling improves the code quality in the *indirect finalization*, and (2) an optimization developed by the author, which is similar to the peephole optimization while the scope is cross basic blocks. The author presents the result and analysis in the subsections 4.3.1 and 4.3.2.

4.3.1 Machine Instruction Scheduling

In LLVM backend code generation, the instruction scheduler runs before and after *the register allocation* (RA). However, they schedule the instructions in different internal representations. In specific, they schedule the *SDNode and MachineInstr* respectively. This subsection changes the default pre-RA and post-RA scheduler to other instruction scheduling algorithms that are also provided in the LLVM infrastructure.

The Figure 4.4 shows speedup of the kernel execution time of indirect finalization if

doi:10.6342/NTU201601752



the LLVM AMDGPU backend is forced to use the *bottom-up (BU)* list scheduling scheme to schedule the LLVM *MachineInstr*s after the *register allocation* (RA). By default, the AMDGPU backend uses the *top-down (TD)* list scheduling scheme in post-RA instruction scheduler. The figure also shows the result of the pre-RA scheduler *list-burr*, which stands for bottom-up register reduction list scheduling, as well as the post-RA BU list scheduler in the legend *list-burr + BU List Scheduling*. The default LLVM pre-RA scheduler is named as *source*, whose behavior is similar to list-burr but schedules in source order when possible.

The result shows that in the condition without changing the pre-RA instruction scheduler, all benchmarks using BU list scheduling scheme in post-RA instruction scheduler outperform the default TD list scheduling scheme. For AESEncrypt and MonteCarloAsianDP, the BU list scheduling scheme achieves 7 percent and 76 percent faster than the counterpart, respectively. In MonteCarloAsianDP, the changing of post-RA instruction scheduler reduces register spilling in several basic blocks from 440 bytes to 296 bytes, which significantly reduces the global memory traffic and also reduces the code length. If the pre-RA instruction scheduler is further changed to list-burr, i.e, combining it with the BU post-RA instruction scheduler, the MonteCarloAsianDP can further improve to 2.92x as fast as the baseline. The spill stack is further reduced from 296 bytes to 240 bytes, and the register pressure is also reduced. The other benchmarks could also benefit from applying the different instruction scheduler schemes. But they don't have to use the stack to store the work-item private data or do the register spill, so the speedup isn't that significant.

4.3.2 Peephole Optimization in GCN MAD and MAC Instructions

The author investigates the GCN assembly code in the BinomialOption and finds that a certain pattern in the LLVM trunk which moves one constant into register and uses the specific register in the later basic block. The Code Snippet 4.1 shows a common pattern

doi:10.6342/NTU201601752

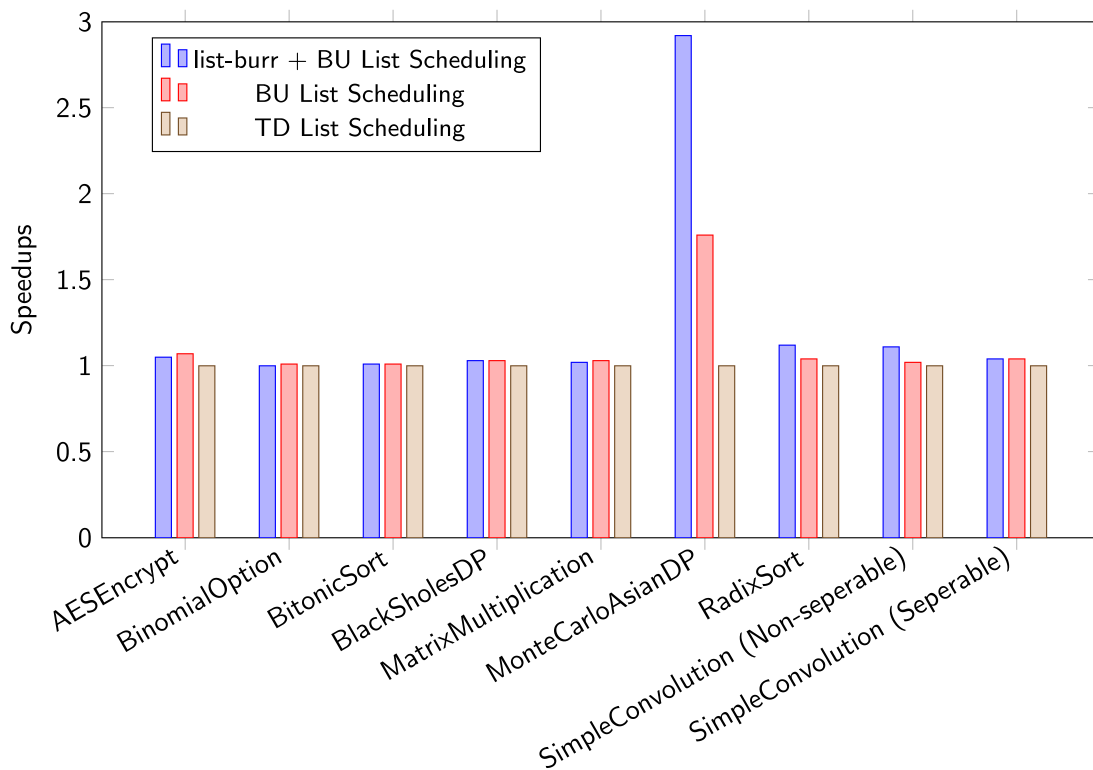


Figure 4.4: Comparison between Bottom-Up and Default Top-Down Instruction Scheduling Schemes. The kernel execution time with default LLVM machine instruction, top-down list scheduling, is normalized to 1.

doi:10.6342/NTU201601752

generated by the AMDGPU backend.

```
1 v_mov_b32_e32 v7, 0x41200000
2 v_mul_f32_e32 v2, v7, v6 ; v2 = v7 * v6
3 v_mad_f32 v9, v7, v1, -v8 ; v9 = v7 * v1 + (-v8)
```

Code Snippet 4.1: Move a Constant into the Register and Use it Later

These literal constants can be folded into the MAD and MAC instructions if the other two input operands are not an inline constant, i.e. a constant selected by a specific VSRC value [3], and turns them into instructions like `v_madak_f32` or `v_madmk_f32`. Also, the VOP2 instruction like `v_mul_f32` can be followed by a 32-bit literal constant.

```
1 v_mov_b32_e32 v7, 0x41200000
2 v_mul_f32_e32 v2, 0x41200000, v6
3 v_madmk_f32_e32 v9, v1, 0x41200000, -v8
```

Code Snippet 4.2: Fold a constant into vector `mul` and `mad` instructions

The Code Snippet 4.2 shows the result of constant folding. As we can see, the register `$v7` is redundant and can be used to hold other values.

These patterns can only be found in the BinomialOption. The other benchmark has no impact on this peephole optimization. As we can see in the Figure 4.5, the code performance is slightly better than the indirect finaliziation for about 2 percent.

The statistics shows that there are 12 additional `madmk` instructions are generated from this passes and 40 constants that folds into the `v_mul_f32`, and generates slightly less code length.

doi:10.6342/NTU201601752

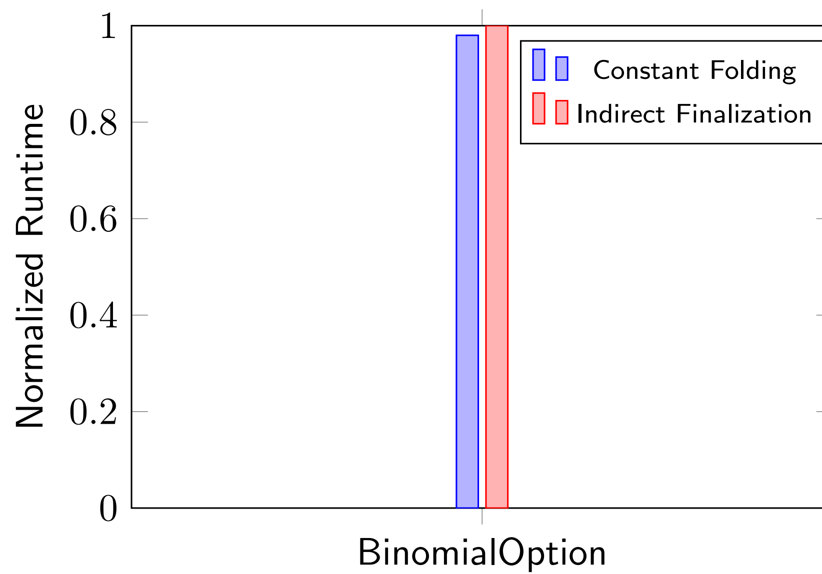


Figure 4.5: Constant Folding in Madmk and Mul Instruction. The indirect Finalization runtime is normalized to 1. The Constant Folding bar shows the difference when applying this peephole optimization.

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

5 Related Work

The author is unaware of other publications that targets HSAIL finalization to the GPU architecture. This work tries the indirect finalization by using the existing LLVM AMDGPU backend. The available finalizer is AMD closed source finalizer, which is not suitable for compiler-based optimizations on AMDGPU.

HSAemu[21] is a system emulator for HSA platforms, and it has the customized HSAIL finalizer targeting the simulators. That work doesn't focus on the code generation of real hardware.

In regards to GPU compiler optimizations, many works are based on source-to-source compiler or PTX-level transformation. Most of them target the branch divergence and utilizing different GPU memory. For example, Yang et al.[25] describes the optimizing compiler to address the utilization of GPU memory hierarchy and management of parallelism. Han et al.[22] describes a work to reduce the branch divergence based on LLVM in loop merging. Wu et al.[24] proposes an GPGPU open-source compiler targeting CUDA and gives some LLVM IR-level optimizations. Some optimizations are already existed but not work well when the target is NVIDIA GPU architecture.

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

6 Conclusion

Indirect finalization leverages existing retargetable compiler infrastructures, such as LLVM or GCC, to quickly craft a finalizer. It offers several advantages, including fast development and deployment of finalizers, avoid costly design and implementation of complicated code optimizations, leveraging reliable and robust open source compiler infrastructures. However, indirect finalization incurs compilation overhead, and since it leverages general purpose optimization passes, it might yield sub-optimal code. This work investigates such issues of indirect finalization. We have developed an HSAIL frontend which translates the HSAIL code to the LLVM IR, and integrating it with the existing LLVM AMDGPU backend. Therefore, instead of translating HSAIL code directly to the target GPU, our translator goes through LLVM to target AMDGPU.

We have applied our indirect finalizer to evaluate kernel code compilation time and execution time on AMD Carrizo APU, targeting its GPU architecture. In contrast, this indirect finalization approach is compared to direct finalization where the HSA runtime invokes the vendor-supplied AMD Carrizo finalizer to generate GPU code. Our experiments have also examined the impact from using various optimization pass combinations in LLVM.

As expected, our results indicate that indirect finalization takes more time to compile, and yields slower code than the direct finalization approach. However, the increased finalization time may be less critical than it looks since the modern runtime system could often keep translated kernel code in some forms of code cache to avoid re-translation overhead. As for

doi:10.6342/NTU201601752

the performance of generated code, our experiments show that the performance impact is there, yet not significantly enough to overshadow the advantages of indirect finalization.

In order to make the indirect finalizer more attractive to the developers, reducing the performance gap between indirect and direct finalizer is much needed. Based on existing LLVM infrastructure, we find that developers could have the performance gain either by changing the LLVM options or by developing low-cost optimization passes.

We expect many finalizers will take the indirect finalization approach in the near future due to its reduced development/deployment time, increased reliability, and superior re-targetability. However, in certain performance critical applications, direct finalization may still be the preferred choice.

doi:10.6342/NTU201601752

A Detailed System Configuration

NAME	VERSION	DATE	TYPE
Linux Kernel Version	4.4.0-kfd-compute	2016-May-6	kernel
AMD KFD Kernel Driver	2.0.0	2016-05-07	kernel
HSA Runtime	1.0.0	2016-04-19	runtime
CLOC Compiler	1.0.10	2016-04-29	compiler
AMD llvm	3.9.0svn	2016-04-28	compiler
HSAIL HLC3.2	3.2svn	2016-04-27	compiler

Table A.1: Detailed Software Stack

doi:10.6342/NTU201601752

doi:10.6342/NTU201601752

Bibliography

[1] 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA) Tutorial. Heterogeneous System Architecture (HSA): Architecture and Algorithms. http://www.hsafoundation.com/isca-2014-tutorial-2/. Retrieved: 2016-07-12.

[2] AMD HSA Team. HSAIL-Tools are used for parsing, assembling, and disassembling HSAIL. https://github.com/HSAFoundation/HSAIL-Tools. Retrieved: 2016-07-20.

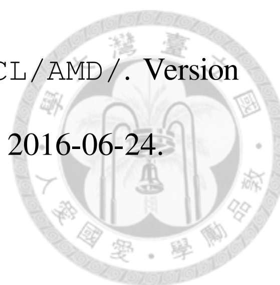
[3] AMD, Inc. AMD Graphics Core Next Architecture, Generation 3. http://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/. Released on 2015. Retrieved: 2016-06-25.

[4] AMD, Inc. AMD OpenCL™ Accelerated Parallel Processing (APP) Software Development Kit (SDK) Verison 3.0. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/. Released on 2016. Retrieved: 2016-06-24.

[5] AMD, Inc. AMD OpenCL™Accelerated Parallel Processing (APP) Software Development Kit (SDK) Version 2.5.

doi:10.6342/NTU201601752

http://uni-smr.ac.ru/archive/dev/cc++/OpenCL/AMD/. Version 2.5 is removed from the official site. Released on 2011. Retrived: 2016-06-24.



[6] AMD, Inc. CL Offline Compiler (CLOC) version 1.0.10. https://github.com/HSAFoundation/CLOC. 2016.

[7] AMD, Inc. ROCm-Profiler. https://github.com/RadeonOpenCompute/ROCm-Profiler. Retrieved: 2016-07-11.

[8] AMD, Inc. Southern Islands Series Instruction Set Architecture. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_Southern_Islands_Instruction_Set_Architecture1.pdf. Released on 2012. Retrieved: 2016-06-25.

[9] Free Software Foundation, Inc. The GNU Compiler Collection. https://gcc.gnu.org/. HSAIL 1.0 support was added to GCC6 on Jan. 2016.

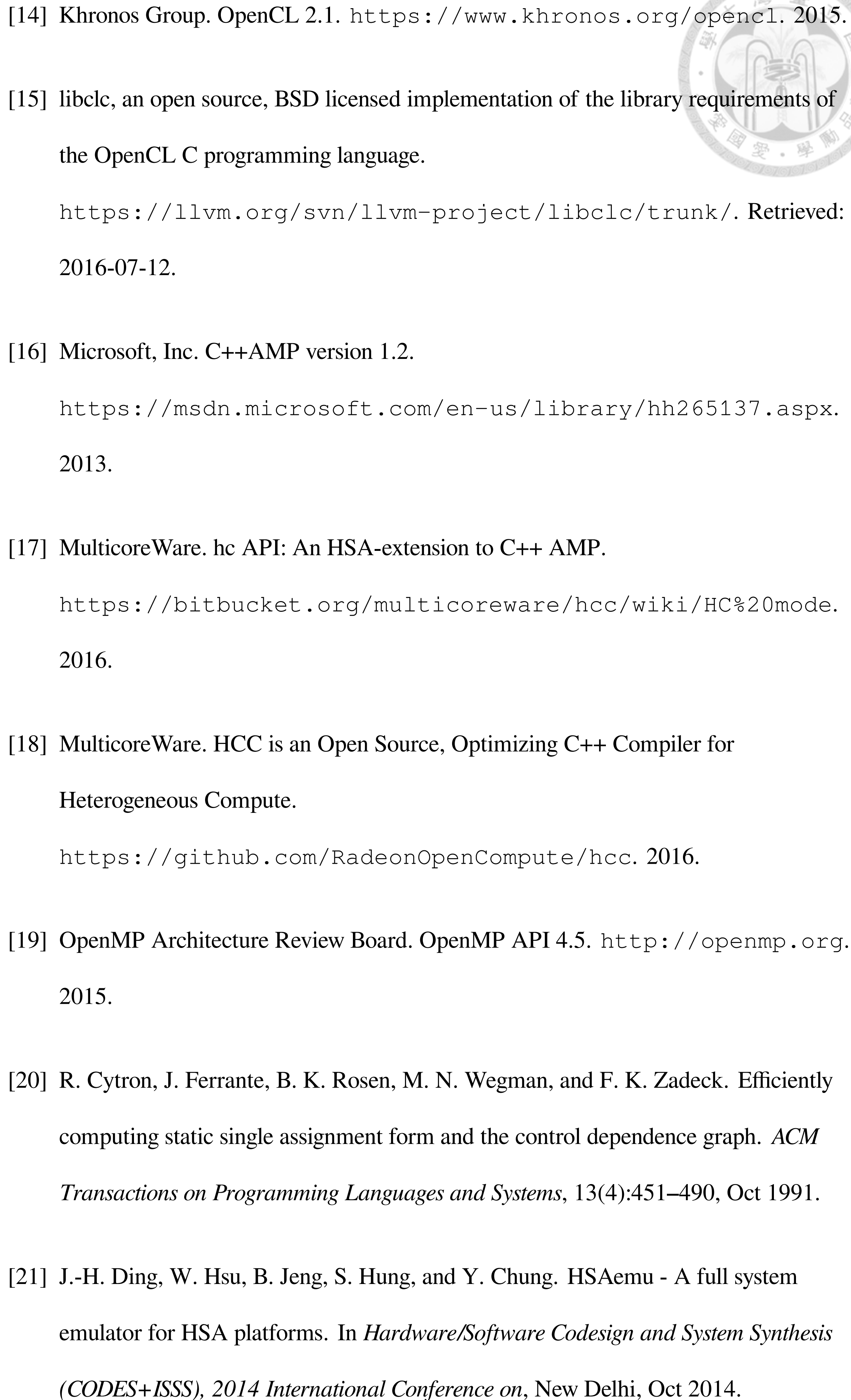
[10] HSA Foundation. http://www.hsafoundation.com/. Retrieved: 2016-06-22.

[11] HSA Foundation. HSA Platform System Architecture Specification 1.00. http://www.hsafoundation.com/standards/. Retrieved: 2016-07-20.

[12] HSA Foundation. HSA Programmer Reference Manual Specification 1.01. http://www.hsafoundation.com/standards/. Retrieved: 2016-06-23.

[13] HSA Foundation. HSA Runtime Programmer’s Reference Manual 1.0. http://www.hsafoundation.com/standards/. Retrieved: 2016-07-20.

doi:10.6342/NTU201601752



[14] Khronos Group. OpenCL 2.1. `https://www.khronos.org/opencl`. 2015.

[15] libclc, an open source, BSD licensed implementation of the library requirements of the OpenCL C programming language. `https://llvm.org/svn/llvm-project/libclc/trunk/`. Retrieved: 2016-07-12.

[16] Microsoft, Inc. C++AMP version 1.2. `https://msdn.microsoft.com/en-us/library/hh265137.aspx`. 2013.

[17] MulticoreWare. hc API: An HSA-extension to C++ AMP. `https://bitbucket.org/multicoreware/hcc/wiki/HC%20mode`. 2016.

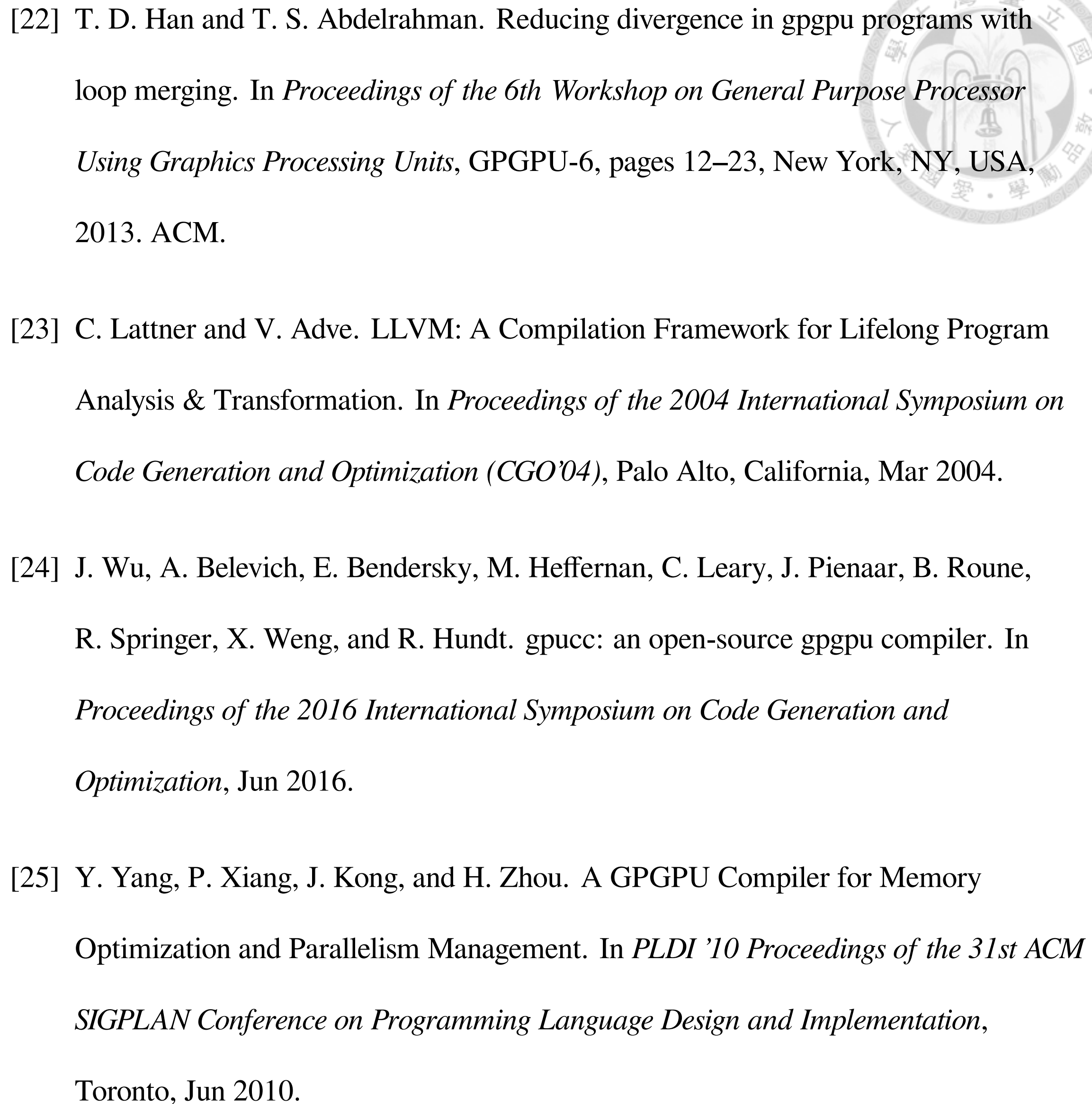
[18] MulticoreWare. HCC is an Open Source, Optimizing C++ Compiler for Heterogeneous Compute. `https://github.com/RadeonOpenCompute/hcc`. 2016.

[19] OpenMP Architecture Review Board. OpenMP API 4.5. `http://openmp.org`. 2015.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[21] J.-H. Ding, W. Hsu, B. Jeng, S. Hung, and Y. Chung. HSAemu - A full system emulator for HSA platforms. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014 International Conference on*, New Delhi, Oct 2014.

doi:10.6342/NTU201601752



[22] T. D. Han and T. S. Abdelrahman. Reducing divergence in gpgpu programs with loop merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 12–23, New York, NY, USA, 2013. ACM.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[24] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. gpucc: an open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, Jun 2016.

[25] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *PLDI '10 Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Jun 2010.

doi:10.6342/NTU201601752