國立交通大學

資訊科學系

碩 士 論 文

一個為異質系統仿真器以 LLVM 為基準的二元轉譯器

An LLVM-based Binary Translator For A Heterogeneous System Architecture Simulator

研 究 生：梁日滔
指導教授：徐慰中　教授

中 華 民 國 一 百 零 二 年 七 月

摘要

通用圖形處理單元（GPGPU）的計算可以更有效的方式與高度並行的加快程序運行。然而，編程模型對程序員不太友好。內存模型是異質的，這樣的編程需要明確的數據傳輸控制系統主內存和 GPU 設備內存。在另一方面，其他如基礎的除錯調試和代碼分發缺乏支持。來自 AMD 的異構系統架構（HSA）以紓緩在 GPGPU 編程複雜性的軟件開發。特色包括共享內存模型，可用於不同廠商硬體上的中介語言（IR）及更具體的操作控制，如控制 GPGPU 的環境中跨工作群的內存存取。在本文中，我們提出的以 LLVM 為基準開發的 HSA 轉譯器為了在一個 HSA 仿真器上提供一個快速的 HSAIL 轉譯。手寫的 HSAIL benchmark 以及 HSAIL 的二元組譯器協助確認功能性上的正確性。

誌謝

感謝這兩年裡面不斷支持與幫助我的每一個人。特別是我的指導教授，徐慰中老師，從他身上我不止學到做研究的學術知識與做研究的態度，更學習到很多的人生道理，做人處事的態度，讓我好好的學習了一門課。感謝口試委員：鍾葉青教授，游逸平教授及楊武教授，在口試時提出不同角度及報告時的問題，讓我了解到更多我沒有考慮到的地方。

感謝清華大學鍾葉青教授的實驗室的研究生，特別感謝郭舟東同學及高崇閔同學，在一起做整個計劃時不斷的與我學術交流及在友情上不斷的支持我。感謝實驗室的同學，大家都沒有介意一個澳門人跟他們一起共事，包容我的澳門國語口音。

感謝港澳會的各位同學，在我這兩年的研究生生活中，給了我很多的勉勵及支持。

最後，感謝我的家人。在我離開家裡的七年歲月中，我一直感到放眼世界的美好。但畢業在即，考慮到的是七年裡都沒有在家裡陪伴父母，而父母卻沒有對我有一絲意見，只有不斷的鼓勵。

未來新一頁的故事，有你們我才可以走下去，感謝你們。

梁日滔

2013/07/22

於國立交通大學

(EC446A)

Agenda

ABSTRACT

General purpose graphical processing unit (GPGPU) computation can speed up the programs with high degree of parallelism in a more power efficient way. However, the programming model is not programmer friendly. The memory model is heterogeneous thus such programming needs explicit data transfer control between system main memory and the GPU device memory from the programmers. On the other hand, other infrastructures such as the debugging and the code distribution are lack of support as well. The Heterogeneous System Architecture (HSA) from AMD rises with such issues to ease the software development in the GPGPU programming. Features including the shared memory model and the re-targetable intermediate representation (IR) with more specific operation controlling such as the cross work group controlling ease the software development in the GPGPU environment. In this paper, we present the HSA Translator for the fast simulation of the HSAIL in the functional level system mode simulator called the HSA Simulator performing the simulation of the HSA environment. It consists of the simulator based on the PQEMU for the simulation of the processing unit in the GPGPU environment. The HSA Translator is implemented in the simulator for the native code translation. The HSA Translator leverages the LLVM infrastructure to translate the kernel source code from the Heterogeneous System Architecture Intermediate Language (HSAIL) to the native re-locatable code. The linking of the native binary is done by a self-implemented link-loader called the HSA Link-Loader implemented in the simulator. The simulation of the kernel processing device is performed by using the host threads in order to speed up the simulation. We evaluate the simulation with the self-translated HSAIL benchmark based on the Rodinia benchmark and the AMD OpenCL samples.

1. INTRODUCTION

Applications having high data parallelism enjoy greater performance and power efficiency from SIMD (Single Instruction Multiple Data) computing devices. Consider the x86 architecture, for example, four operations in one SSE (Streaming SIMD Extension) instruction and eight operations in one AVX (Advanced Vector Extension) instruction can be processed simultaneously. Such performance and power efficiency in SIMD processing has been further pushed up to GPGPU, where a much greater number of cores can be used to compute in SIMD or SIMT (Single Instruction Multiple Thread) fashion.

SIMT is the strategy of using a large of number of threads in parallel, but each thread executes the same instruction on a different data section allocated to this thread. Like SIMD processing, the overhead of instruction fetch, decode, and speculation can be effectively eliminated, SIMT has a higher power efficiency. The device memory hierarchy in GPGPU also contributes to the greater power efficiency of SIMT processing. Using CPU and GPGPU collaboratively to achieve greater performance and power efficiency is the current trend of computing. This is often referred to as ***heterogeneous computing*** since the GPGPU is often using a different ISA (Instruction Set Architecture) from the CPU.

CPU, as designed for general-purpose computing, is less power efficient for SIMD processing. General purpose computing often incurs complex instruction execution control flow, which requires sophisticated branch prediction, speculation, out-of-order execution, and cache hierarchies. For processing a large and regular section of data, SIMD or SIMT architecture could yield much greater power efficiency. The motion for heterogeneous computing is to leave the logic control portion of an application to the CPU and let the GPGPU handles the regular data parallel sections.

One challenge of heterogeneous computing is how to program for the two different ISAs within the same application. CUDA from NVIDIA is one of the earliest programming model for heterogeneous computing. However, CUDA is designed for NVidia devices only, not portable for other GPGPU devices. OpenCL (Open Computing Language) is a programming model, initially developed by Apple, and later promoted by the Khronos Group. As the open standard for heterogeneous computing platforms. Early programming models for heterogeneous computing platforms focus too much on the efficient use of device memory hierarchies. They are not programmer friendly. For example, the current existing GPGPU programming model requires explicit control of data transfers between the host memory and the device memory. Furthermore, debugging based on such models is difficult, as the kernel functions running on the device fails to provide the basic debugging functions like the single step execution and break point setting. Due to the separate memory spaces, a device memory pointer can access the device memory only. Such difficulties calls for a revised programming model based on HSA (Heterogeneous System Architecture) from AMD for heterogeneous computing platforms.

The main idea of HSA is to make the GPGPU software development easier. The memory model of HSA is called HUMA (Heterogeneous System Architecture), which offers a shared virtual memory space between the host and the device. Since the memory pointers are now shared, such sharing strategy allows programmers accessing the device and host memory with the same memory pointers. Thus software developers can focus more on computing algorithms rather than on managing explicit memory copies between the separate address spaces. In addition, HSAIL (Heterogeneous System Architecture Intermediate Language) is an intermediate representation of HSA for machine independent code distribution. The machine independent code distribution is achieved through the on device code generation from HSAIL to the native code. The native code generated is linked and loaded by an on device linker and loader. Precise definition of parallel data syntax, no high level structure representations and a finite register set in HSAIL allow programmers to get a more thorough understanding of their code. Vector instructions in HSAIL offer chances of straightforward SIMD instruction generation with less analysis in speeding up the device code generation. Cross work group and cross lane operations in HSAIL also provide finer control on data computing. . The rest of this paper uses the term *agent* for the host code following the terminology of HSA.

Our work is to design a functional level system mode simulator for HSA. The simulator runs two guest machines at the same time. One guest machine is an ARM processor as the current embedded systems are mostly based on ARM processors. The other guest machine we simulate is the GPGPU. Both the ARM guest machine and the GPGPU are emulated by the x86-based host machine. The ARM guest is emulated by one x86 core, while the GPGPU is emulated by many x86 cores on the host machine. Therefore, the emulated GPU can make use of the multi-core power available in the host machine in order to achieve faster simulation.

As one of the first few groups developing for the HSA framework, we are facing many challenges. For example, no OpenCL/HSA compilers are available at this time. In order to test our HSA simulator, we must come up with our own HSAIL code. We have implemented a binary generator for translating HSAIL text into HSAIL binary code called Brig. This tool is called Brig generator. Following the HSA specification, the communication between the host and the device is via queues which are explicitly managed through AQL (Architected Queuing Language). AQL packets are used to specify the required kernel execution information. In the QEMU-based HSA simulator, the simulation of GPGPU is based on binary translation, which the Brig code is translated into native binaries to be executed directly on the host machine. On a real heterogeneous machine, this step is called finalization which translate the Brig code into device code to be running on the GPGPU device. Since we do not know which GPGPU device will be used, the functionality of the device is simulated by the host machine. Therefore, the current finalizer simply translate the Brig code into native binaries to be executed on the host machine. This Brig-to-Host-binary translator is the main theme of this thesis work. It is built as a library for the QEMU, performs the device code generation. Native code is linked through the HSA link-loader implemented in the QEMU. Thereby, the HUMA and the HSAIL on device code generation requirements are met in our simulator.

Our implementation of the HSA simulator follows the HSA 1.0 specification. In the rest of this thesis, we will introduce the background of this work in chapter 2. Related work will be introduced in chapter 3. Design of our project framework will be introduced in chapter 4. Implementation details will be described in chapter 5. Experimental results will be discussed in chapter 6 and conclusion as well as future work will be given in chapter 7.

2. BACKGROUND

2.1 GPGPU (General-Purpose Graphical Processing Unit) Computation

The GPGPU computation such as the CUDA and the OpenCL is the practice of the combination of the SIMT (Single Instruction Multiple Threads) and the SIMD (Single Instruction Multiple Data). The SIMT leverages the fact that the GPU has abundant number of cores providing parallel computation from collections of threads. Such parallel computation can be done due to the regularity of the kernel function instructions. Branch divergence is solved from the use of a control bit mask recording the jumping of basic block in each thread. Each thread dispatched is considered to be independent to each other. The SIMD vector units available in the cores improve each thread in each computation.

The collection of threads in the GPGPU is divided into groups called work-groups. Each work-group is executed by a group of cores in the GPU, which is called the streaming multi-processor. Each thread in the work-group is called work-item. Each work-item is executed by a core in the group of processor running the work-group the work-item belongs to. A fixed number of work items executed in parallel specified as the wavefront. The size of a wavefront is defined by the devices. During the execution, work-groups can share the same memory space called global memory. Within the same work-group, work-items in the same work-group shared the memory space called local memory. Every work-item has a private memory space for the data that no synchronization is needed. Whenever synchronization is necessary, barrier calls are provided for the programmers to perform such data synchronization activity. The barrier calls can be applied to the level of the local memory or the whole global memory for the synchronization of the whole execution or the synchronization of each work group. Different calls are specified by specific API calls such that programmers can choose the suitable one to perform the necessary synchronization.

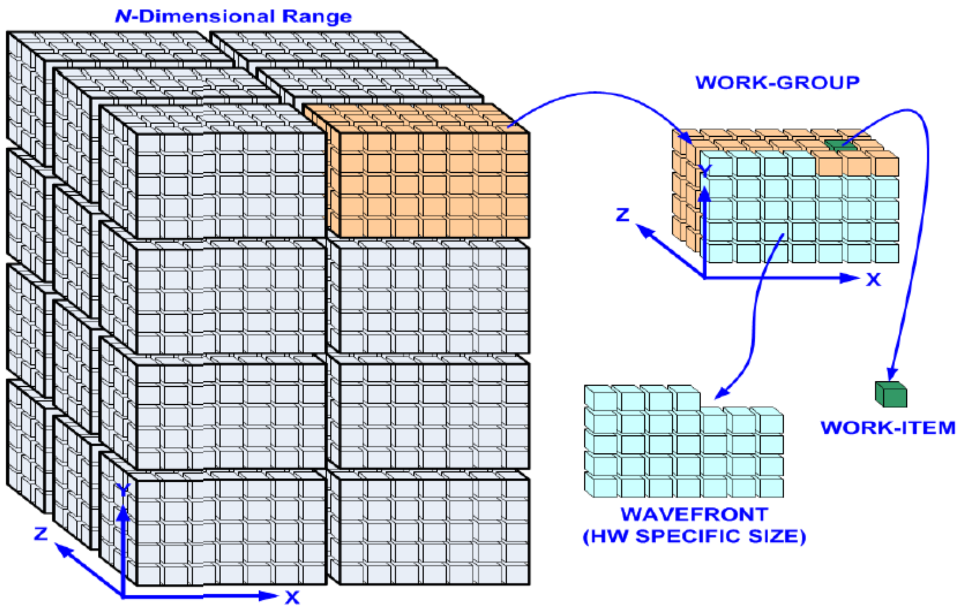

Figure 2. 1. The GPGPU computation model is an N-Dimensional Range. Each work group consists of a group of work-item. Hardware specific wavefront size is decided by the size of work that can be computed at a particular instance.

The memory model of the GPGPU computation is a heterogeneous memory model consisting the main memory of the system and the device memory in the graphics card. Before the kernel execution, data allocated in the host memory is copied to the device memory via API (Application Programming Interface) calls from programmers. When the kernel execution is done, another API call is needed for the data transfer from the device memory to the host memory. Such messy API calls are needed due to the limitation that the GPU can only access the GPU memory. Which means, the device memory pointer can only access the device memory. Data size exceeds the size of the device memory needs to do chunking by the logic of the programmers. However, the CUDA has introduced the cudaHostAlloc API call in order to solve the problem. The cudaHostAlloc API call performs pseudo-shared virtual memory effect, such that the chunks can be manipulated by the runtime.

Nevertheless, the cudaHostAlloc has solved one of the difficult parts of memory management, the difficulties in using explicit call syntax of reading and writing device memory is still unchanged.

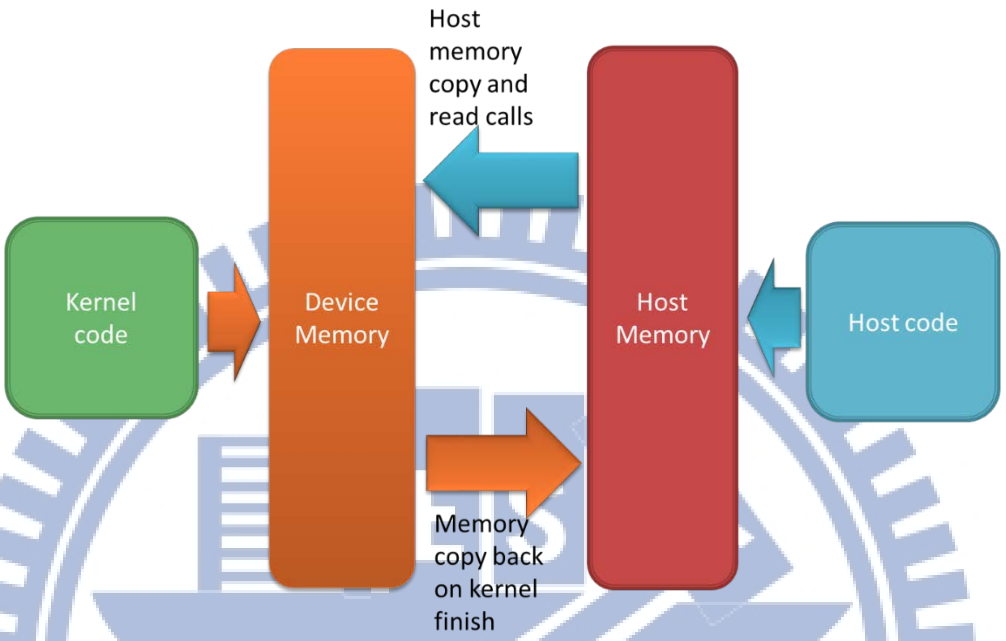
Host memory copy and read calls
Kernel code
Device Memory
Host Memory
Host code
Memory copy back on kernel finish

Figure 2.2. The GPGPU programming model consists of the device memory and the host memory. In programmers' view, these two memories should be maintained with two different syntaxes. All memory allocations and memory movements need thorough understanding in order to prevent naïve programming and errors.

2.2 Heterogeneous System Architecture (HSA)

With abundant number of cores favoring the practice of multi-thread programming, the importance of the GPGPU grows within the past decade in both the high performance computing and the practical application such as gaming and the computation of the large data digesting programming such as the computational fluid dynamics. However, several difficulties stop programmers in developing application using such solution, including the explicit memory syntax, debugging difficulties and the device dependent recompilation.

The data transfer is needed to be controlled by the programmers in the GPGPU programming for the memory read/write from/to between the host memory and the GPU device memory. For example, in order to have an array addition of $a + b = c$,

1. Three device memory allocation calls,
2. Three device memory writing calls. (Two for preparing the data in array A and B, and one for the initialization of the buffer for array C)
3. One device memory reading call for the answer extraction of the array C from the device memory to the host memory.
4. Three device memory release calls.

These data transfer controlling causes a difference with the traditional high level language programming such as C/C++ and JAVA because the memory models are different. The memory syntax in controlling the device memory is described as API calls. Beginners cause severe performance decrease of the GPGPU program because naïve writing of the data transfer controlling may lead to cyclic copying.

Debugging is hard in the GPGPU program due to the incomplete definition of the debugging features of the GPGPU framework. Current existing debugger from the vendors fail to provide the debugging features of a traditional CPU program debugger, such as break point, single-step execution and variable watches etc. These features need some kind of a support in the compilation stage, as some of the program information should be included in the program binary. However, the current existing tool-chain provides no debugging features at all. Besides, the current existing GPGPU programming model provides only single direction calls. Debugging is nearly impossible if the bug is appeared in the side of the device memory because there is no device to host call available.

The GPGPU code needs to be recompiled if the code is shipped from one vendor machine to another due to the binary compiled is for only a single vendor.

Toolchains provided by the vendors may not be able to compile the code for the entire device. Vendor specific optimization also needs to be specified and recompiled for each device specification. The virtual GPU ISA (Instruction Set Architecture) such as the AMDIL (AMD Intermediate Language) and the PTX (Parallel Thread Execution) can only be run on the AMD and NVIDIA graphics card respectively. In addition, programmers may need to modify the code even though no vendor specific optimization needed. No cross vendor re-targetable ISA is available in order to ease the code distribution inconvenience.

Solution called the Heterogeneous System Architecture (HSA) is carried out from AMD in solving the software development difficulties. The HSA is aimed in supporting CPUs and GPUs from multiple vendors. The vital idea is that programmers can have an easier way to program the GPGPU code with only the knowledge of the high-level language. System software manipulates the lower level, such as the actual memory copy as well as making use of the programmable compute elements from different vendors seamlessly.

The implementation of the HSA includes both the hardware and the software. The HSA conformed hardware implementation requirements in the specification 1.0 are,

I. Shared Virtual Memory Model between the CPU(s) and the GPU(s)
II. Cache Coherency Domains
III. Memory-Based Signaling and Synchronization
IV. User Mode Queuing
V. Preemptive HSA Component Context Switching
VI. Architected Queuing Language (AQL)
VII. HSA Component IEEE754-2008 Floating Point Exception Reporting

VIII. HSA Component Hardware Debug Infrastructure

IX. Efficient System Call Infrastructure

X. HSA Platform Topology Discovery

The shared virtual memory model is defined as the HUMA (Heterogeneous Unified Memory Access). The need of the explicit device memory movement syntax is due to the separate virtual memory space of the host virtual memory and the device virtual memory. Thereby, the HUMA defines the shared virtual memory space within the host and the device. The device can access all the shared memory including the main memory if the HSA conformed hardware is used. Since the programs with the HSA accesses the same virtual memory space, the explicit memory syntax of the device memory can be replaced by the syntax of the host memory. In the other words, programmers can use the same high-level language memory syntax for both the host and the device memory operations. In addition, the actual copy operation is controlled by the runtime functions instead of the syntax from the programmers. Under this mechanism, programmers have the view of a single piece of flat virtual memory. The unified syntax of the C/C++ style makes the programming language be friendly to the programmers. Furthermore, since the memory copy is done by the runtime functions, naïve memory copy such as cyclic copying can be prevented. Software development using the HUMA feature can have a neat and tidy view replacing a set of APIs. GPGPU computing beginners can catch up quickly as the way of programming is the same as the high-level languages instead of the necessity of learning sets of APIs.

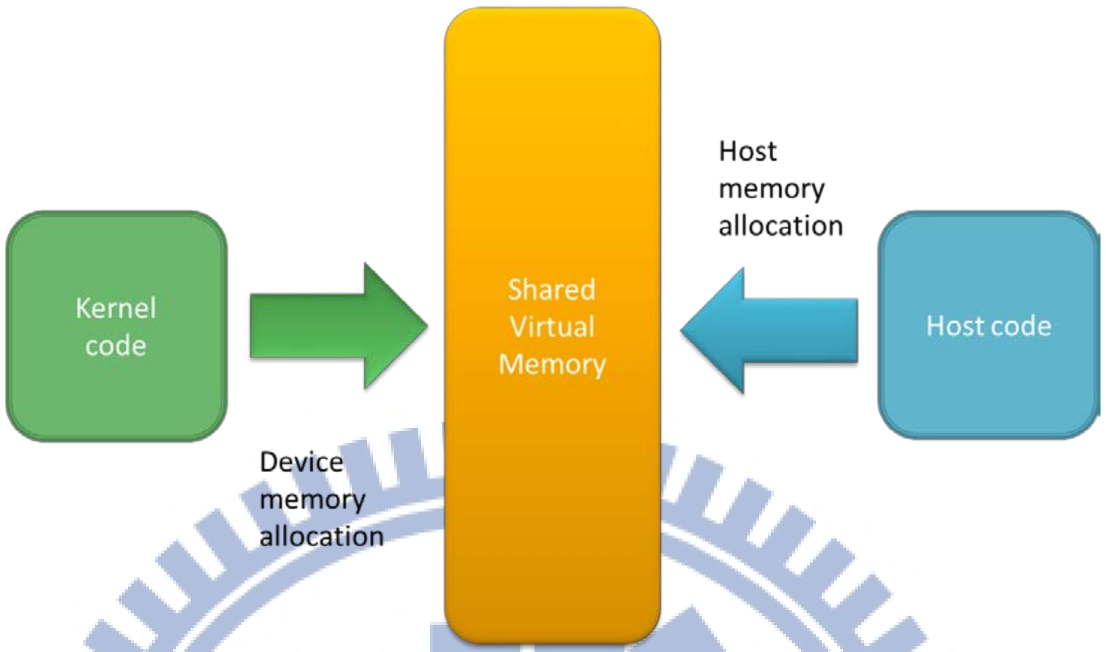

Figure 2.3. The shared virtual memory model of HUMA in programmers' view

Data memory accesses in HSA can be available in any sharable regions. Current existing GPGPU memory model raises the difficulties in cache/data coherency due to the separate copy of the memory pointers. Nevertheless, the HSA exists actual copying of the data from host to memory but the memory pointers exist in the same memory space. Data access request to the host main memory from the device can be done without any copies if the HSA conformed hardware are used. However, read-only data needs to be constant during the execution, the copies still have to be done. On the contrast, such constant data can skip the data coherence problem. Thereby, the difficulties between the host and device memory coherency maintenance can be solved.

The synchronization in the HSA is done through the signaling raising in the memory. Such signaling mechanism provides the ability of controlling a memory object, there are four types of signaling may occur,

a. Allocation of a memory object.

b. Destruction of a memory object.

c. Control signal of a memory object, may occur when we need to do any atomic operation in the HSAIL.

d. Waiting of a memory object, such as the resource waiting in a race problem.

The signals are allocated by the runtime. Such allocations are triggered by the HSAIL description. In addition, such mechanism is controlled and manipulated by the HSA agents. Inter-process accessing is not supported such that data within kernels are not the same copy.

Multi-level user queue are supported in the HSA. The necessity of this issue is due to the multiple CPUs and GPUs support of the HSA. A single queue may cause serious performance loss. Multi-level queue may practice several optimization techniques such as priority queue in preventing the resource abundant device running a resource non-thirsty kernel. The queue contains the packets in the structure of the AQL (Architected Queuing Language), which we introduce in this article later.

The context switch in the HSA including stopping running wave-fronts, saving/restoring the current context state to/from memory, recovering the execution from saved state. This requirement is defined in the preemption of the components. The mechanism provide,

a. Preemption and forced exit of the currently running wave-fronts.

b. Saving the context on a per-compute-unit granularity.

c. Saving all the state and restoring to restart the preempted context.

The importance of such preemption exists due to the GPGPU computation needs the full occupation of the device. Such that when users need to perform a computation with another device needed action such as high-definition display, existing GPGPU solutions are failed to provide full performance ensuring. Recovering the context per-compute-unit can provide the availability of the computation in a more precise granularity. In addition, Optimization with historical record needs the change in HSA component context thus the support for the saving of all state and recovering to restart the preempted context is vital in reaching better performance gain.

The communication between the host and the device codes in the shared are defined by the AQL (Architected Queuing Language). The packet written in AQL structure is manipulated by the host/agent providing the information to the device in order to setup the kernel execution. The information in the AQL packet including,

i. The synchronization information, including native synchronization function, the necessary cache flushing mechanism, cache level action, performance counter sampling rate and the dimension of the kernel dispatching.

ii. Completion object and Kernel object address. This is necessary because the kernel code is provided in the shared virtual memory space. The address in the AQL packet is needed for the device to retrieve the kernel code. The kernel code available has already translated to native code.

iii. Kernel arguments addresses. The kernel arguments are located in the kernel argument segment (kernarg) of the shared virtual memory.

iv. The size of the grid and work group is written in three dimensions.

v. The runtime-defined information including the size of the work group segment, work item private segment, work item spill segment, work item argument segment. The sizes of the segments are in bytes.

The IEEE754 floating-point exception defines the unified format of the HSA floating exception reporting. Five types of floating exception are defined,

Exception code 0: INVALID OPERATION

Exception code 1: DIVIDE BY ZERO

Exception code 2: OVERFLOW

Exception code 3: UNDERFLOW

Exception code 4: INEXACT

The existing GPGPU programming model is hard to debug due the lack of support in the hardware. In addition, the debugging features existing comparing to that of the traditional high-level programming makes a great difference such as the absence of the instruction breakpoints. Profilers require the information from the performance monitor and the sampling information from the memory accesses. Such information is limited in the current existing hardware as well. Such that the debugging infrastructure in the HSA is defined to have the minimum requirement of the instruction breakpoint mechanism and others are set to be implementation defined. We expect the debugging infrastructure will be as thorough as usual in the traditional CPU only application running environment.

The system call infrastructure in HSA supports the bi-directional system call structure. Such that the device can raise various system call in order to mention the current situation, such as the exceptions, the finish of the kernel execution etc. With the bi-directional call, the execution is more efficient because there is no need for the host to trace the device execution for minimizing the signaling of the kernel return.

As a platform supporting CPUs and GPUs from multiple vendors, the discovery of the HSA platform topology is vital for the execution of an application. An HSA Memory Node (HMN) represents a set of HSA components. Each HMN can access to the memory attached to such HMN and other regions with system-defined attributes. The design of the memory access is an extension of the NUMA (Non-Uniform Memory Access).

The software implementation of the HSA includes the high-level language compiler, linker and loader, the finalizer and the HSAIL (Heterogeneous System Architecture Intermediate Language). The HSA tends to support multiple CPUs and GPUs from different vendors. To achieve this issue, the HSAIL is introduced as a re-targetable intermediate representation and providing the necessary components for on device linking. The software flow should be,

1. The high-level compiler compiles the high-level language to the HSAIL.
2. The finalizer provides the native code translation from the HSAIL to the native code.
3. The linker and loader link the native code with the native library to have the native binary.

2.3 HSAIL (Heterogeneous System Architecture Intermediate Language)

The HSAIL is the re-targetable intermediate language for the representation of the kernel operations in the HSA. The major features are the memory model, the fixed register set and the large number of operation codes with precise syntax for the synchronizations. The binary format of the HSAIL is called Brig.

The memory resource in the HSAIL is divided into three types, the flat memory, the registers and the image memory. The flat memory can be accessed through a certain segment base address plus offset. The segments are divided the way data can be shared and intended usages. Seven segments are divided as the following,

a. Global – shared by all the agents.
b. Group – shared by work-items of a work-group.
c. Private – local to a single work-item.
d. Kernarg – read only, used to pass arguments to kernel.
e. Readonly – holding the constants.
f. Spill – load or store register spill.
g. Arg – used to pass arguments in and out of functions.

The system software maintains the starting addresses of the segments. Image memory is used for image operations only for current specification.

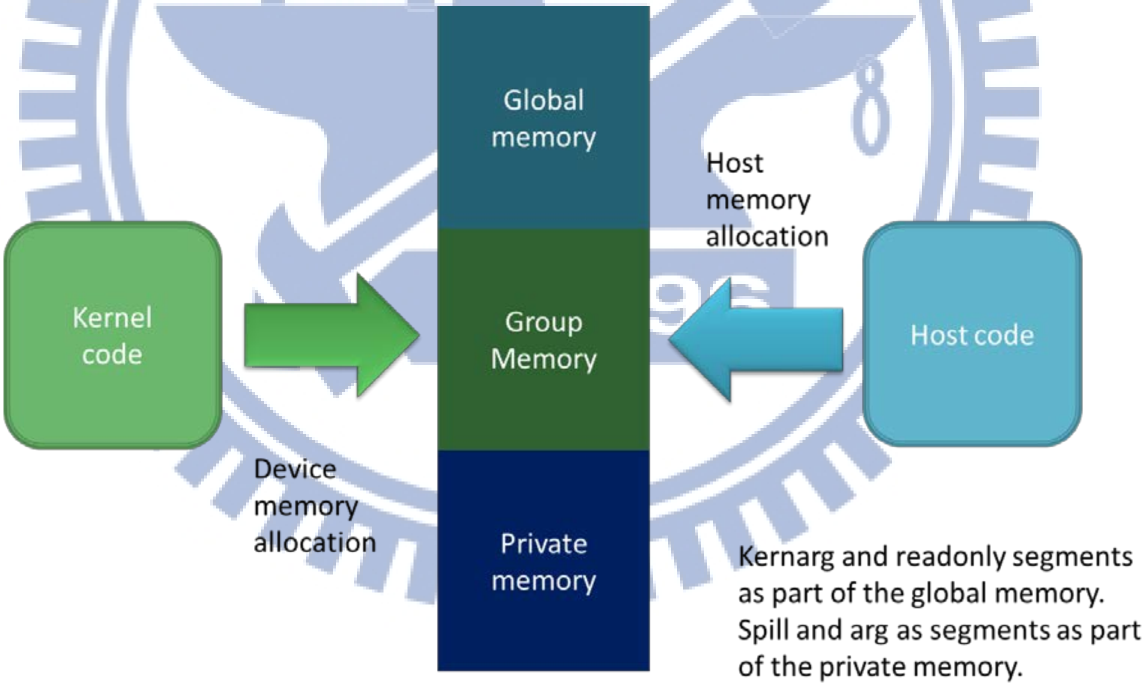
Global memory
Host memory allocation
Kernel code
Group Memory
Host code
Device memory allocation
Private memory
Kernarg and readonly segments as part of the global memory. Spill and arg as segments as part of the private memory.

Figure 2.4. The segment memory model in HSAIL is divided into the global, group and private memory. In between, the Kernarg and the readonly segments are implemented as part of the global memory while the spill and arg segments are implemented as part of the private memory.

Registers in the HSAIL are limited resources. Results from the all operations are stored in the registers. Types of registers are divided into four categories by the bit width in 1-bit, 32-bit, 64-bit, and 128-bit as shown in table 2.1. In addition, S, D and Q registers can be used in the vector instructions. C is used for control signals. Such limited register set favors the speed of the on device translation, as lesser analysis needs to be performed in the register mapping. Syntax of the vector instructions not only favors the programmers in exploiting the performance of the GPGPU programs, translation to the on device SIMD (Single Instruction Multiple Data) is more straightforward as mapping of the instructions is relatively easy. In case the registers is not enough, spill memory segment works as a swap for the registers.

Register types	Number of registers	Bit width
C	8	1
S	128	32
D	64	64
Q	32	128

Table 2.1. Limited registers with various bit width.

The HSAIL has 133 operation codes in the HSA version 1.0. Operations are in variable length. There are several categories,

1. Arithmetic operations
2. Memory operations
3. Image operations
4. Branch operations
5. Parallel synchronization and Communication operations
6. Operations related to functions
7. Special operations

One or more modifiers for the definition of each operation follow after the operation codes. Compare to the current GPGPU programming language syntax, HSAIL has special branch and synchronization syntax. Making use of the fine-grained barrier, these two kinds of operation can follow by the fine-grained barrier modifier to specify the number of the work items in performing the branches and the barriers.

Both text and binary formats are supported in the HSAIL. The Brig, which is the binary format of the HSAIL, has five sections. Each component in each section has a corresponding byte offset for searching the information in the other sections.

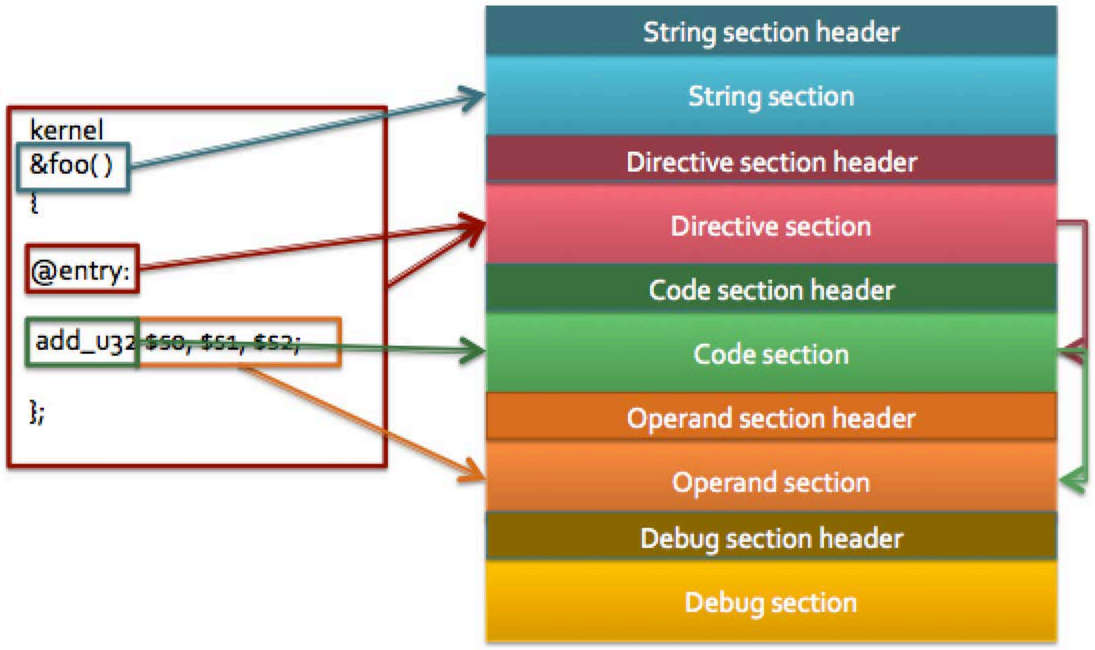

Figure 2.5. Sections in Brig start with a section header to specify the size of the section. Different components are encoded in the corresponding sections.

Comparing to the PTX, the HSAIL is more flexible in both programming and easier for distribution. The PTX fails to provide precise syntax in some complex logic needed operations such as cross lane, cross work groups. Lack of supports in this syntax may cause serious difficulties in both debugging and programming since programmers may not get thorough understanding of their code. Dealing with no memory operations, using the HSAIL favors the beginners while the PTX does not. Adding the support of bi-directional calls, dynamic memory can be allocated during the kernel runtime. Since the HSAIL is re-targetable, PTX is able to be translated, but not vice versa.

Comparing to the SPIR, which is the LLVM IR with OpenCL support from Intel. The LLVM IR does not have a complete parallelism model. Acting as a re-targetable IR, the LLVM IR is a good practice but may not be the case of GPGPU programming. With the messy memory operations, programmers have to deal with the parallelism model themselves. LLVM IR also raised debugging issue. High-level structures are kept in the IR thus programmers are hard to debug without a debugger.

2.4 QEMU/PQEMU

The design of architecture requires certain level of evaluation. The cost of manufacturing the real hardware for the evaluation only is non-cost effective. Incredibly high capital in developing the hardware for a whole system is considerably impossible using practical hardware evaluation. Nowadays, lots of hardware design industries prefer software simulation to achieve the evaluation to minimize the cost of the hardware design.

The simulation of a system should provide the system level evaluation. User level simulation fails to provide the observation of some architectural details that are vital to the design flow, such as memory access, device queuing etc. Thereby, the importance of the full system simulation in simulating a specific architecture is a crucial research area. The most remarkable open source full system simulator should be the QEMU.

The QEMU is an open source system emulator, supporting the X86, X86_64, ARM, MIPS and POWERPC environments system virtualization. The QEMU provides the ability of running operating system with the support of the preferred architecture below. The emulation of the QEMU runs through the dynamic binary translation engine called TCG (Tiny Code Generator). The dynamic binary translation achieves very good performance as the guest code is translated once only. Certain translated code is stored in the code cache for future reuse. Such that, the most time costly procedure is run in minimal time leads to a better performance. However, the disadvantage of the QEMU is that the multi-thread support is limited. The con due to the QEMU is developed in the era of single core. Multiple threads are mapped to a single core even if the host machine is multi-core. With the growing complexity of the operating system and the application, performance of the simulators with sufficient multi-core support can be magnificent. Thereby, the PQEMU[1] based on QEMU is introduced.

The PQEMU[1] (Parallel QEMU) is aimed to improve the multi-core support in the QEMU. By extending the ability of emulating multi-core on multi-core, the PQEMU[1] exploits the power of multi-core in the simulation of the multi-core system. In addition, the unified code cache strategy minimizes the usage of the memory from no sharing components duplication. Separate code cache is used for the duplication of the sharing components for individual thread emulation. Such mechanisms lead to a very good performance in the emulation of systems with multi-thread support.

Comparing the model of the GPGPU device computation with the multi-thread computation, the PQEMU[1] satisfies the issue in simulating such device. Therefore, we decide to make use the PQEMU[1] in our simulation in the future. The simulator in this article is based on the QEMU. We expect the performance of the simulator can be speed up with the use of the PQEMU[1].

3. RELATED WORK

Our work is related to several research areas, such as dynamic binary translation, GPGPU simulation and system emulation.

We use the LLVM infrastructure [5] introduced by Chris Lattner as a base of the translation in our HSAIL converter. Translating HSAIL to LLVM IR is a new branch of the LLVM frontend. Binary translator such as the LLBT[3] and LNQ[9] are binary translation of executable binary. Our HSA Translator is a finalizer that translates the HSAIL to LLVM IR and LLVM IR to native re-locatable object. We provide HSA Link-Loader for the Linkage process to form a native executable binary.

GPGPU-sim [2] introduced by Tor Aamodt and his graduate students provides a detailed simulation of a contemporary GPU running CUDA and OpenCL workloads with an integrated power model at micro-architectural level. Barra-sim [6], a simulator of GPGPU based on the UNISIM framework developed by Daumas M running CUDA, does the simulation at functional level. Ocelot [7], developed by Gregory Diamos is a modular dynamic compilation framework for heterogeneous system, targeting to several back-ends with self-developed translator. AMD FusionSim [8] based on GPGPU-sim[2] introduces a CUDA workloads simulation on X86 systems with an additional cache simulation. Concurrently simulates an X86 out-of-order CPU, a CUDA-capable GPU and a CPU/GPU interconnect memory system.

Most of the simulators mentioned above rely on CUDA or OpenCL runtime, which is provided by each hardware vendor. The HSA is introduced to have support multi-target, thus with the support of the HSA runtime the issue of relying unique runtime can be solved. The full system simulation in our work provides verification of the new runtime components in the future. In addition, the HSA supports bi-directional calls that allows the GPU to access the I/O devices. The above works are hard to achieve such simulation because this model is rather distinct with the existing model. In our work, we achieve such system mode simulation with a QEMU based full system simulator. Besides, re-targetable code is not provided. Linking to the native library is available in our work as well to perform the on device linking.

GPGPU-sim[2] is micro-architectural level simulator while Ocelot[7], Barra-sim[6] and our work are at functional level. On the contrast, only our work is a system mode simulator. With the support of QEMU, we are able to simulate other development board such as versatile Pico board, Panda board as well as ARM-Vexpress.

Recompilation is not needed in all of the above cases. Unfortunately, except our work all the simulators need an exceptional dynamic linking to the system's vendor supporting library, such as libcudart.so. Relinking is not needed in our approach as once linking is done in the first run; linked code is stored in the code cache and reused in the future.

The HSAIL is a brand new intermediate language to the existing simulators. We preserve the opportunities of optimization in running the HSAIL, as the translator of such intermediate language is self-developed. Unlike the existing simulators, most of them use the vendor providing tool chain thus optimization may be narrowed, but Ocelot[7] also implemented its own compiler from the LLVM IR to native code.

GPGPU-sim[2] provides functional level simulation in PTX and NVIDIA native ISA. We are not providing native GPU ISA simulation as GPU ISA keeps changing. Also, HSAIL is an intermediate language targeting different native ISA, unlike PTX only targeting its own vendor ISA.

Our work also provides a simulation to the current embedded system environment. In the experiment we boot up with Linaro Linux images, simulating the hardware of ARM-vexpress-a9 board. The GPGPU-sim[2], Barra-sim[6] and Ocelot[7] target both agent and kernel code in the same environment as they are in user mode simulation. System mode simulation raises chances to debugging, profiling as well as further research in the embedded system.

4. PROJECT DESIGN

The main achievement of this work is the implementation of a translator called the HSA Translator for the Heterogeneous System Architecture (HSA) functional level simulator. An additional BRIG generator is implemented for the binary generation of the HSAIL. The HSA Simulator and the HSA Link-loader are not the achievement of this paper. The kernel execution in the HSA Simulator is explained for the sense of readers may not understanding the use of the HSA Translator.

4.1. HSA Simulator

The HSA Simulator is a functional level system mode simulator that emulates the HSA conformed GPGPU computation environment. The whole simulation is run on the CPU without the support of the GPU. The implementation of the HSA Simulator is based on the QEMU. In the HSA Simulator, the ARM guest performs the role of the CPU in the GPGPU environment. The ARM guest runs with the TCG IR provided in the QEMU. We emulate the GPU guest with a large number of host thread in order to speed up the simulation. In addition, the GPU guest runs the kernel function translated from the HSAIL. Therefore, we implement a finalizer called the HSA Translator to do this task.

The shared virtual memory model requirement states that,

a. The full address range should be visible to all components,
b. Non-shared memory addresses are maintained by system software,
c. Virtual address translation is done by the system software,
d. Minimum 48-bit address length for 64-bit architecture and 32-bit address length for 32-bit architecture.

Point (a) is achieved by the implementation of shared QEMU memory between the emulated CPU and the emulated GPU. During the application execution, all sharable

QEMU memory is accessible through the use of QEMU memory helper functions that perform the virtual address translation. Accesses to the non-shared virtual memory such as Operation system preserved memory addresses are avoided by these QEMU memory helper functions. Thus point (b) and (c) are achieved. For point (d), we use 32-bit virtual address length in our implementation.

Allocating and freeing the memory object in the agent code only achieve the Memory-Based Signaling mechanism. Such that the agents have full control of the entire memory object and no inter-process memory transaction occurs. The virtual memory objects (aka. The registers in the HSAIL) is left to the LLVM analysis.

The current HSA Simulator emulates only one GPU device. Thus we consider having the support for multi-level queue later as evaluation to the effect of the multi-level queue is not convincible if we emulate only one device. Thereby, the multi-level user queue is left to the future work.

The Architected Queuing Language (AQL) is a command interface for the dispatch of the HSA components commands. This command interface allows application to build and enqueue the HSA components own command packets, enabling fast dispatch. These AQL packets provide the information including the size of the grid, the addresses of the kernel arguments, the kernel object address etc. We achieve this part by allowing the agent code writing this packet in the shared virtual memory. The emulated GPU owning the AQL packet memory pointer gain access to the packet. Thus the function of this AQL packet is achieved.

The efficient system call infrastructure requires the system call should be resolved to the correct system call API, such as interrupts caused by instructions during execution. This part we discuss in two categories, the CPU and the GPU. The CPU is emulated using the ARM virtual machine. The interrupt handler is implemented in the original QEMU. Therefore, we rely on the QEMU to solve this task for the emulated CPU execution. The GPU emulating environment is the X86. We modified to make the execution of the GPU code running directly on the host machine. The host machine architecture is X86_64 architecture. Thus, we may rely the host machine reacting to the exact interrupts.

The preemptive HSA component context switching and the HSA component topology discovery are used when we have more than one homogeneous HSA components, aka two or more CPUs, two or more GPUs. The current version of the HSA Simulator emulates single CPU and single GPU environment. Thus we have not implemented these two parts yet. The debug infrastructure, preemptive HSA component context switching and the HSA component topology are considered as the future works of this achievement.

The cache coherency domains refer to the data coherency in the cache level within the HSA components. As the HSA Simulator performs a functional level simulation, we do not provide the micro-architectural model for the cache simulation.

The running flow of the HSA Simulator is shown in figure 4.1. Before execution, both the agent and kernel binaries are placed in the ARM guest machine memory. Users execute the agent binary in the ARM guest machine. When the kernel function call in the agent binary is being executed, three steps are taken.

1. The kernel binaries have to be copied to the shared QEMU memory because the two individual guest memory is not visible to each other. The virtual address translation is performed by the QEMU.
2. The AQL packet information is written by the agent code.
3. The memory pointer of the AQL packet is transferred to the GPU

emulating guest machine. This triggers the emulated GPU to read the necessary information to perform the kernel execution.

The kernel execution consists of three parts, the HSAIL to native code translation, the native code linking and the native code execution.

4.2. HSA Translator

The start of the kernel execution is the kernel binary to native translation. The HSA Translator translates the HSAIL binary format to the re-locatable native code. Such task is categorized as the finalizer in the software implementation in the HSA requirement. The task of a finalizer including the address translation, the native operation translation and the register allocation.

The HSA Translator is implemented based on the LLVM infrastructure. The translation is in a static form. Comparing to the TCG (Tiny Code Generator), we prefer the LLVM infrastructure with the following reasons,

i. The design of TCG is for the system emulation. System binary is not necessary to have a static translated binary because we may not have such capability to keep the translated binary. On the contrast, the kernel binary is relatively small in size.

ii. The optimization in TCG is not enough. The TCG is designed for fast translation. Thus simple does fast. Comparatively, the LLVM infrastructure is a relatively complete. With the support of the first point, we may get more chances in optimization as the translation is static.

iii. The Kernel function is more regular compare to the system emulation. Optimization affects the performance more significantly in a regular case as the analysis can be done easily and more straightforwardly.

The shared memory model is implemented using the QEMU memory. Such that the memory translation needs the help of the QEMU virtual memory translation. Thereby, the memory operation is translated to the self-implemented QEMU helper functions. The helper functions load the necessary memory pages and perform the load/store operations to the corresponding HSAIL memory operations.

The HSAIL operations are translated to the native operations with the help of the LLVM. We map the HSAIL operations to the suitable LLVM operations. The kernel functions are a group of HSAIL operations. Each kernel function is translated to an LLVM module and the whole program is translated to the native code through the LLVM Infrastructure.

The register allocations is achieved through the register mapping from the HSAIL register to the LLVM register. Since the mapping is a fixed register set to the infinite register set, a mapping table is necessary to be implemented in the HSA Translator. The register allocations in the host environment is left to the LLVM Infrastructure.

The generated re-locatable native code is linked by the HSA link-loader.

4.3. HSA Link-loader

The native code generated by the HSA Translator leaves several types of function unlinked,

a. The QEMU memory helper functions

b. The native library functions, such as math library

The LLVM infrastructure aims to have IR as the media of re-targetable ability. Whenever we need to generate a certain native code, we can use the LLVM infrastructure performing the JIT (Just in Time) compilation, which is a dynamic binary compilation. Thus, no linking methods are provided in static compilation.

In order to achieve a linking process for our re-locatable native code, we implement a linker called the HSA Link-loader. This Link-loader is implemented in the QEMU. After the process of the HSA Link-loader, the kernel function is linked and stored in the code cache. The binary stored in the code cache can be reuse during the same execution runtime. After the execution, the binary is released from the code cache.

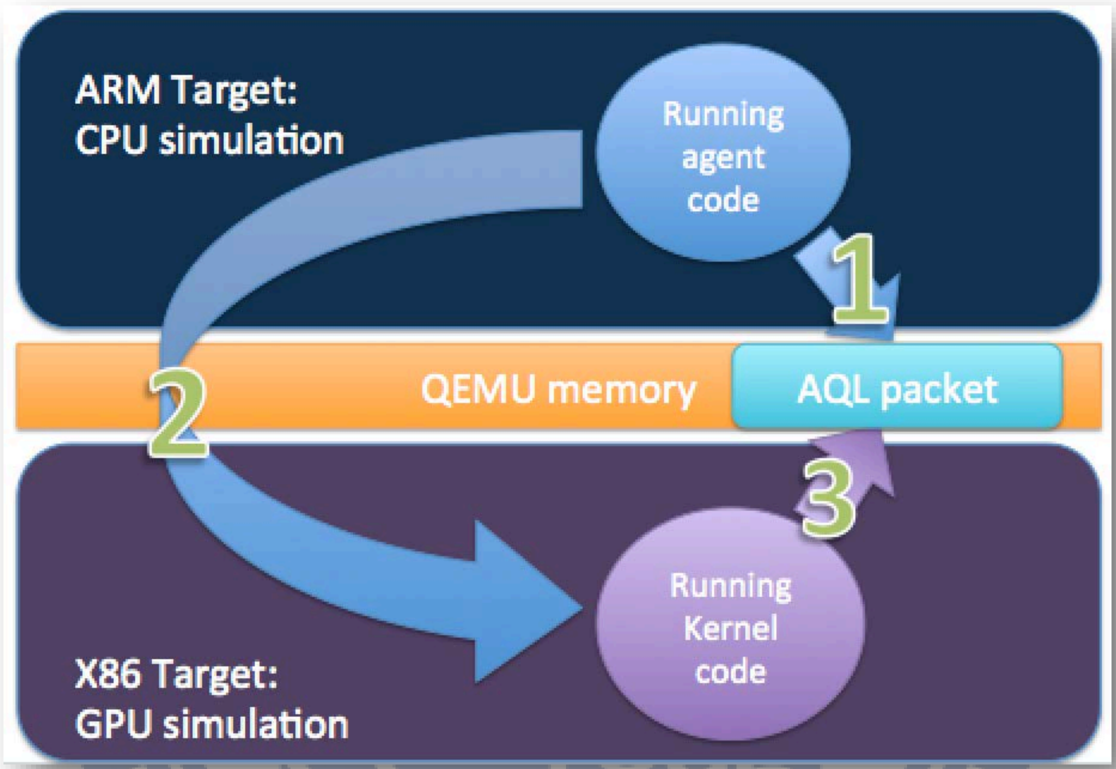

Figure 4.1.The overall flow diagram of the HSA simulation starts with the running of the agent code. First, the CPU simulating machine writes kernel execution required information into the AQL packet. Second, the GPU simulating target will be triggered to read the AQL packet. Finally GPU simulating target obtains the required information in the QEMU memory according to the AQL packet described and the translation of the kernel code is started.

4.4. HSA Kernel Execution

The linked binary is run directly on the host machine. The number of thread used to run the execution is stated by the boot up argument in the startup script of the HSA Simulator.

During the execution, all the threads share the same virtual memory. Memory helper function calls are used for the accesses of the memory for the load/store operations. Memory addresses of data for an individual thread is computed in the HSAIL in order to make use of the flat, shared virtual memory space.

Work groups are dispatched serially. Such that work items within a work group is processed in parallel but only one work group can be processed at a time. When threads within the processing work group have finished, synchronization is done at the end. Barrier operation stated in the kernel function is executed with the help of pthread_barrier_wait. The pthread_barrier_wait forces to have a barrier which all the threads are not allowed to proceed before all the threads waits in the barrier.

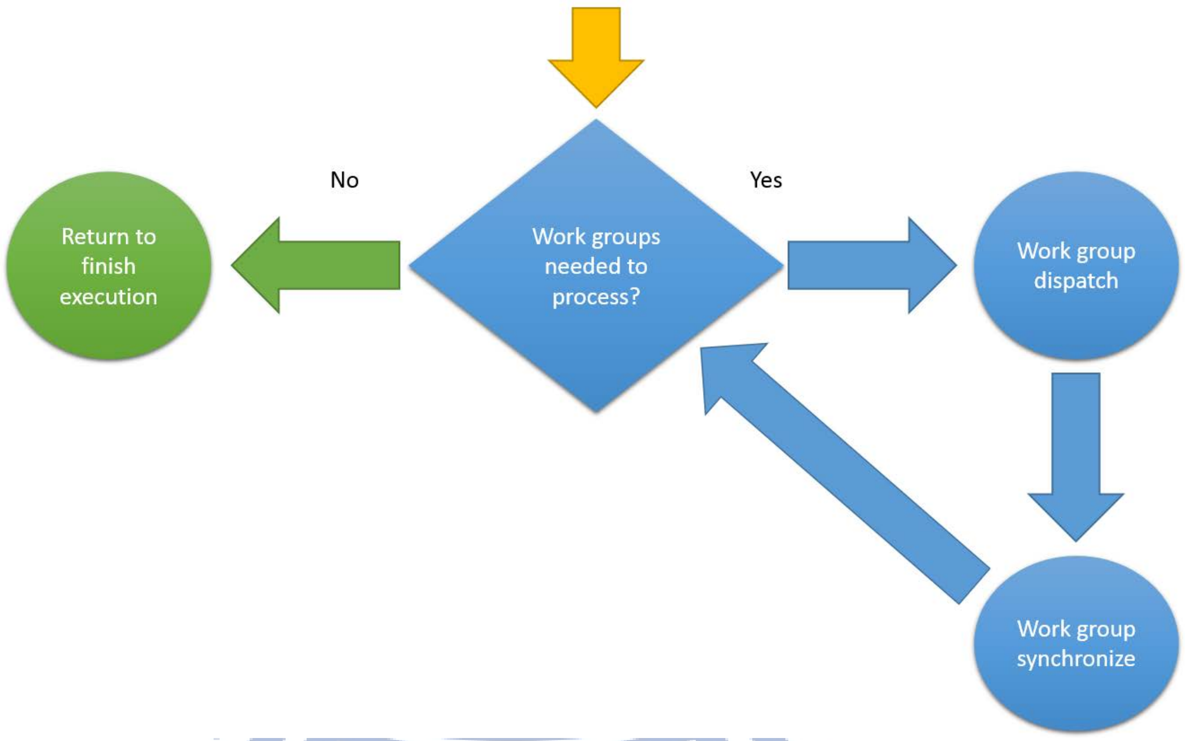

Figure 4.2. The work group dispatching loop.

The loop of dispatching work groups are done when the number of processed work groups has reached the number of work groups calculated by dividing the global size with the local size. Both the global size and the local size are stated by the programmer in the agent code. When all the threads dispatched are synchronized, the GPU guest machine triggers the CPU guest machine to collect the output data from the shared virtual memory. After that, the GPU threads are said to be idled and wait until the triggering from the CPU guest machine.

After the kernel execution, the AQL packet of the executed kernel is freed and the next packet is on the role. Consequently, after all the packets in the queue are processed, the queue is freed and the core of the program is said to be finished. The system model simulator will remain idle before the next program execution. All the threads will be destroyed and recreated in the next program execution.

5. IMPLEMENTATION

The implementation includes the BRIG generator and the HSA Translator. As the HSA link-loader is closely related to our achievement, a brief explanation is provided as well.

5.1 BRIG Generator

The BRIG generator is used to translate the HSAIL from text format to BRIG format, which is the binary format of the HSAIL. The BRIG generator consists of the HSAIL Lexical analyzer and the HSAIL rule parser. This implementation is based on the LEX/YACC structure. Both these two components conform to the HSAIL 1.0 version.

The HSAIL Lexical analyzer takes in the HSAIL text code and passes the tokens to the HSAIL rule parser for rule mapping. Tokens include the identifiers, the constants, the operation codes and the modifiers etc. Rules written in the HSAIL parser is implemented according to the EBNF (Extended Backus-Naur Form) grammar rules in the HSAIL specification document.

The BRIG format stores the components into five sections. They are the string section, the directive section, the code section, the operand section and the debug section. Each section has a corresponding section header consists of a 32-bit integer indicating the size of the section. Zeros are padded to the strings to make every string writing in the file a multiple of four. Every IR structure size is also in multiple of four. Therefore, the size of a BRIG file must be in multiple of four.

In addition, the HSAIL version 1.0 and the version before 1.0 have several differences,

1. The writing format of the BRIG – the version before 1.0 has 5 offsets

indicating the sizes of the five sections appear at the beginning of the BRIG. Instead, the sizes of the sections are located at the beginning of each section.

2. The format of the encoding of the string section – the version before 1.0 deduces each string in the string section with the null character. Such mechanisms makes the decoding to have a slower response due to the sizes of the strings are unknown until the null characters are encountered. In the version 1.0, each string has the corresponding size before the string. In addition, the sizes of the strings are padded to the multiple of four with zeros.
3. More native library functions – such as the floor and ceiling floating point operations are unavailable in the version before 1.0.

According to the specification of the HSAIL, we need to preserve the order of the sections in a BRIG file. For example, the string section is the first section and the operand section is the fourth section. When we parse a register, the identifier of the register should be in the string section and the register IR structure should be in the operand section. In order to write the components to corresponding sections, the IR structures and the strings are stored in the corresponding section buffers within the generation instead of directly writing to the file. When the parsing of the HSAIL text code is finished, buffers are written into the corresponding sections of the binary file. The BRIG is used as the input of the HSA Translator in the GPU guest machine for the HSAIL to native code translation.

This part is included in the scope of this paper.

5.2 HSA Translator

The execution of the GPGPU program with the HSA conformed simulation is shown in figure 5.1. The BRIG binary and the ARM agent binary are placed in the memory of the ARM guest machine. Users execute the agent code in in the ARM guest machine. When the kernel call in the agent code is being executed, the BRIG is copied to the QEMU physical memory. By reading the information in the AQL packet, the GPU simulating target finds the BRIG file and calls the HSA Translator via a library function call. The memory pointer pointing to the BRIG is the parameter of the HSA Translator.

The BRIG format has a section header before every section. The section header indicates the size of the section. The HSA Translator reads in the BRIG into five buffers. The sizes of the buffers are allocated according to the size specified by the section headers. Reading in the BRIG is necessary because we need to process between sections frequently. Using file pointer to traverse the file may cause severe overhead due to the frequent use of system call. Thus we sacrifice the space of storing the BRIG file into several memory buffers instead of such system calls.

A BRIG IR structure always starts with a 32-bit integer indicating the size of the structure. Following is a 32-bit integer indicating the kind of the structure. Making use of this information and knowing which section we are reading, we can distinguish all the structures. Decoding process starts with the decoding of the directive section. Code section is accessed if the directive has a code body, such as the kernel and function. Likewise, operand section is accessed if the operation in the code section needs any operands.

(a) HSAIL vector addition kernel code:

```
version 1:0:$large;

kernel &__OpenCL_vec_add_kernel(kernarg_u32 %arg_val0,
  kernarg_u32 arg_val1, kernarg_u32 %arg_val2, kernarg_u32 %arg_val3) {
  @__OpenCL_vec_add_kernel_entry:
        ld_kernarg_u32 $s0, [%arg_val3];
        workitemabsid_u32 $s1, 0;
        cmp_lt_b1_u32 $c0, $s1, $s0;
        ld_kernarg_u32 $s0, [%arg_val2];
        ld_kernarg_u32 $s2, [%arg_val1];
        ld_kernarg_u32 $s3, [%arg_val0];
        cbr $c0, @BB0_2;
        brn @BB0_1;
  @BB0_1:
        ret;
  @BB0_2:
        shl_u32 $s1, $s1, 2;
        add_u32 $s2, $s2, $s1;
        ld_global_f32 $s2, [$s2];
        add_u32 $s3, $s3, $s1;
        ld_global_f32 $s3, [$s3];
        add_f32 $s2, $s3, $s2;
        add_u32 $s0, $s0, $s1;
        st_global_f32 $s2, [$s0];
        brn @BB0_1;
};
```

(b) LLVM Instructions:

```
; ModuleID = 'hsail'


define void @__OpenCL_vec_add_kernel(i32 addrspace(1)* %arg_val0, i32
addrspace(1)* %arg_val1, i32 addrspace(1)* %arg_val2, i32 addrspace(1)*
%arg_val3) nounwind {
entry:
  %addr = ptrtoint i32 addrspace(1)* %arg_val3 to i32
  %0 = alloca i32
  store i32 0, i32* %0
  store i32 %addr, i32* %0
  %1 = alloca i32
  store i32 0, i32* %1
  %2 = call i32 @helper_hsa_get_global_id(i32 0)
  store i32 %2, i32* %1
  %3 = alloca i1
  store i1 false, i1* %3
  %4 = load i32* %1
  %5 = load i32* %0
  %icmp = icmp ult i32 %4, %5
  store i1 %icmp, i1* %3
  %addr1 = ptrtoint i32 addrspace(1)* %arg_val2 to i32
  store i32 %addr1, i32* %0
  %addr2 = ptrtoint i32 addrspace(1)* %arg_val1 to i32
  %6 = alloca i32
  store i32 0, i32* %6
  store i32 %addr2, i32* %6
  %addr3 = ptrtoint i32 addrspace(1)* %arg_val0 to i32
  %7 = alloca i32
  store i32 0, i32* %7
  store i32 %addr3, i32* %7
  %8 = load i1* %3
  br i1 %8, label %"@BB0_2", label %"@BB0_1"

"@BB0_1":                                         ; preds = %"@BB0_2",
%entry
  ret void

"@BB0_2":                                         ; preds = %entry
  %9 = load i32* %1
  %shl = shl i32 %9, 2
  store i32 %shl, i32* %1
  %10 = load i32* %6
  %11 = load i32* %1
  %add4 = add i32 %10, %11
  store i32 %add4, i32* %6
  %12 = load i32* %6
```

```
%13 = alloca float

%callLoad32 = call i32 @_load_32(i32 %12)

%Fcast = bitcast i32 %callLoad32 to float

store float %Fcast, float* %13

%14 = load i32* %7

%15 = load i32* %1

%add5 = add i32 %14, %15

store i32 %add5, i32* %7

%16 = load i32* %7

%17 = alloca float

%callLoad326 = call i32 @_load_32(i32 %16)

%Fcast7 = bitcast i32 %callLoad326 to float

store float %Fcast7, float* %17

%18 = load float* %17

%19 = load float* %13

%fadd = fadd float %18, %19

store float %fadd, float* %13

%20 = load i32* %0

%21 = load i32* %1

%add8 = add i32 %20, %21

store i32 %add8, i32* %0

%22 = load i32* %0

%23 = load float* %13

%UIcast = bitcast float %23 to i32
```

```
    call void @_store_32(i32 %22, i32 %UIcast)

    br label %"@BB0_1"

}



define void @Kernel_Entry(i32 addrspace(1)* %parmaList) nounwind {

entry:

    %0 = ptrtoint i32 addrspace(1)* %parmaList to i64

    %1 = add nuw i64 %0, 0

    %2 = inttoptr i64 %1 to i32 addrspace(1)*

    %3 = load i32 addrspace(1)* %2

    %4 = inttoptr i32 %3 to i32 addrspace(1)*

    %5 = add nuw i64 %0, 4

    %6 = inttoptr i64 %5 to i32 addrspace(1)*

    %7 = load i32 addrspace(1)* %6

    %8 = inttoptr i32 %7 to i32 addrspace(1)*

    %9 = add nuw i64 %0, 8

    %10 = inttoptr i64 %9 to i32 addrspace(1)*

    %11 = load i32 addrspace(1)* %10

    %12 = inttoptr i32 %11 to i32 addrspace(1)*

    %13 = add nuw i64 %0, 12

    %14 = inttoptr i64 %13 to i32 addrspace(1)*

    %15 = load i32 addrspace(1)* %14

    %16 = inttoptr i32 %15 to i32 addrspace(1)*

    call void @__OpenCL_vec_add_kernel(i32 addrspace(1)* %4, i32 addrspace(1)*
%8, i32 addrspace(1)* %12, i32 addrspace(1)* %16)

  ret void

}



declare i32 @_load_32(i32) nounwind

declare i32 @helper_hsa_get_global_id(i32) nounwind

declare void @_store_32(i32, i32) nounwind
```

Figure 5.1. The mapping of the HSAIL operations and the corresponding LLVM operations

Most of the Arithmetic operations can be mapped to the LLVM operations using one-to-one mapping. Some of them are mapped to combinations of LLVM operations,

1. The 24-bit operations – 24-bit operations is absent in the LLVM operations. We need to perform a bit mask of 24-bit after such operations.
2. The bit string, copy and multimedia operations – Whilst the LLVM operations does not provide such operations, we can achieve such functions with a combinations of the arithmetic operations.
3. The segment checking and converting operations – These two kinds of operations need the support from the environment. The current version of HSA Simulator has no work group memory implementation. Therefore, the segment checking and converting operations are left to the future work.
4. Mathematic operations are defined by the host library. Thus helper functions are implemented for the mathematic operations.

HSAIL operations	Mathematic helper functions
*floor	helper_Ffloor
*ceil	helper_Fceil
sqrt	helper_Fsqrt
fract	helper_Fract
fma	helper_Fma
cos	helper_Fcos
sin	helper_Fsin
log2	helper_Flog2
exp2	helper_Fexp2
rsqrt	helper_Frsqrt

Table 5.2. The mathematic operations are implemented by the helper functions. The operations with "" is introduced after HSAIL version 1.0.*

The memory operations class consists of the load, store and the atomic operations. The load and store operations are translated to LLVM function call with different function prototypes indicating the load/store performing to different bit-width of data. In addition, due to the lack of support in work group memory, all the memory accesses are treated as the memory accesses to the flat global memory. In addition, each HSAIL memory operations load/store the data with the help of the memory address parameters which are the memory addresses in the QEMU memory space. Thus the memory helper functions take the memory addresses provided in the HSAIL to compute the addresses of the data in the QEMU memory. The load operations have the data bit-width data type and the store operations are in void type. When loading a floating point data, an LLVM bit-casting operation is needed to change the data type of the data from integer to floating point. Likewise, the store operations need a bit-casting operation to change the data type of the data from floating-point to integer type in order to perform the storing. Whilst the operation is called bit-casting, the bit-width of the data is still unchanged and the only difference is the operating data type in the LLVM IR stage. Furthermore, the fine-grained syntax in the HSAIL for the memory operations are not supported in the current version of the HSA Simulator. Every memory operation existing in the kernel function is treated as the memory access that should be performed by every work item. Atomic operations are not supported in the current version of the simulator.

HSAIL operations		Memory helper functions
ld	8	uint8_t load_8(u32 addr)
	16	uint16_t load_16(u32 addr)
	32	uint32_t load_32(u32 addr)
	64	uint64_t load_64(u32 addr)
st	8	void store_8(u32 addr, uint8_t val)
	16	void store_16(u32 addr, uint16_t val)
	32	void store_32(u32 addr, uint32_t val)
	64	void store_64(u32 addr, uint64_t val)

Table 5.2. The mapping table of HSAIL operations to the QEMU memory helper functions.

Branch operations except the fine-grained modifier function are mapped to the LLVM branch operations. The HSAIL function body operations are present in basic blocks. Therefore, the branch operations in the HSAIL can be mapped to the LLVM IR as the LLVM also uses the basic block as the basic component for the container of the operations. However, the compare and branch operation in the HSAIL has to be mapped to one compare and one conditional branch in the LLVM IR as there is no such operation in the LLVM IR.

Synchronization operations such as the barriers are implemented by the helper functions. Thus we translate the barriers to helper function call. In addition, the current version of the HSA Simulator only provides the work group level thread synchronization. The fine-grained barrier is considered as the future work.

The special functions such as the work group id queuing and the work item absolute id queuing are implemented by the helper functions. Thereby we translate such operations to helper function call.

HSAIL operations	Special helper functions
workitemid	helper_workItemId
workitemabsid	helper_workItemAid
workgroupSize	helper_workGroupSize
gridsize	helper_workGridSize
gridgroups	helper_wordGridGroups
laneid	helper_laneId
maxwaveid	helper_maxDynWaveId
maxcuid	helper_maxCuId
dispatched	helper_dispatchId
dim	helper_workDim
workitemabsflatid	helper_workItemaidFlat
workitemflatid	helper_workItemidFlat

Table 5.3. Operation mapping of the HSAIL special operation to the helper functions.

According to the specification of the HSAIL, all operation destinations must be registers. In order to maintain the values in the registers, we do a register table through a set of memory pointers. In the LLVM IR, the operand types and the destination allocated type must be the same, such as floating-point operations must have floating-point destination registers and floating-point source registers. On the contrast, the HSAIL has no specific floating-point registers. Therefore, we do the bit casting in order to meet such restriction. The bit casting is done when we need a floating-point value from an integer value with the same bit width, such as 32-bit floating-point value to 32-bit integer value, or vice versa. Whenever a value is going to be written into a register, we check the type of the value with the type of the allocation of the register. If they are not the same, we use the LLVM allocation instruction to have a new allocation with the type of the being written value for the register. This new memory pointer replaces the memory pointer of such register in the register table. The truncations and the extensions of each type of the values are also implemented to make sure the values generated fit the bit width of the destination registers. The native registers manipulation is left to the LLVM Infrastructure.

Vector types are supported in the HSAIL. We translate these types to the LLVM vector types. The LLVM can generate the native SIMD instructions easily by the explicit use of the LLVM vector types and vector type operation.

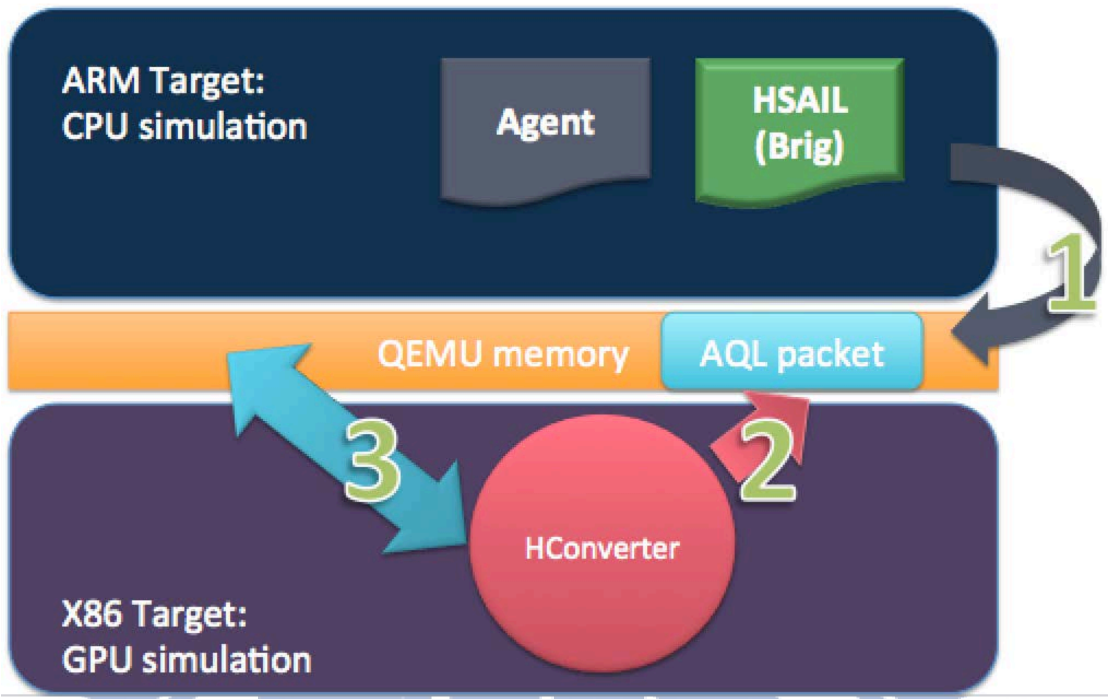

Figure 5.2. First, the BRIG is copied to the QEMU physical memory. Second, the X86 target obtains the QEMU physical memory address of the BRIG through the reading the AQL packet. Third, the memory pointer pointing to the BRIG is passed to the HSA Translator. After the translation, a re-locatable object code is outputted to the QEMU physical memory.

A wrapper function called kernel entry in the generated LLVM IR for the pass of the memory addresses. When the kernel code is being executed, the wrapper is called first and the actual kernel function will be called by the wrapper. The wrapper is added because,

a. Parameters passing to the kernel function are in pointer types as the operations in the kernel code perform direct reading and writing from/to certain memory addresses. The number of the parameters passing to the kernel function are unable to be determined in before runtime stating. Thus we need a wrapper for the parameter passing to the actual kernel function.

b. The kernel function prototype and identifier is runtime stated. The QEMU cannot call a function with undetermined pointer name. Therefore, we provide a fixed wrapper function pointer for calling the kernel function.

The LLVM infrastructure does not have the linker support. Thus the HSA Translator generates these helper function calls and leaves them unlinked. Finally a re-locatable native object code with helper functions unlinked is outputted to the QEMU physical memory.

This part is included in the scope of this paper.

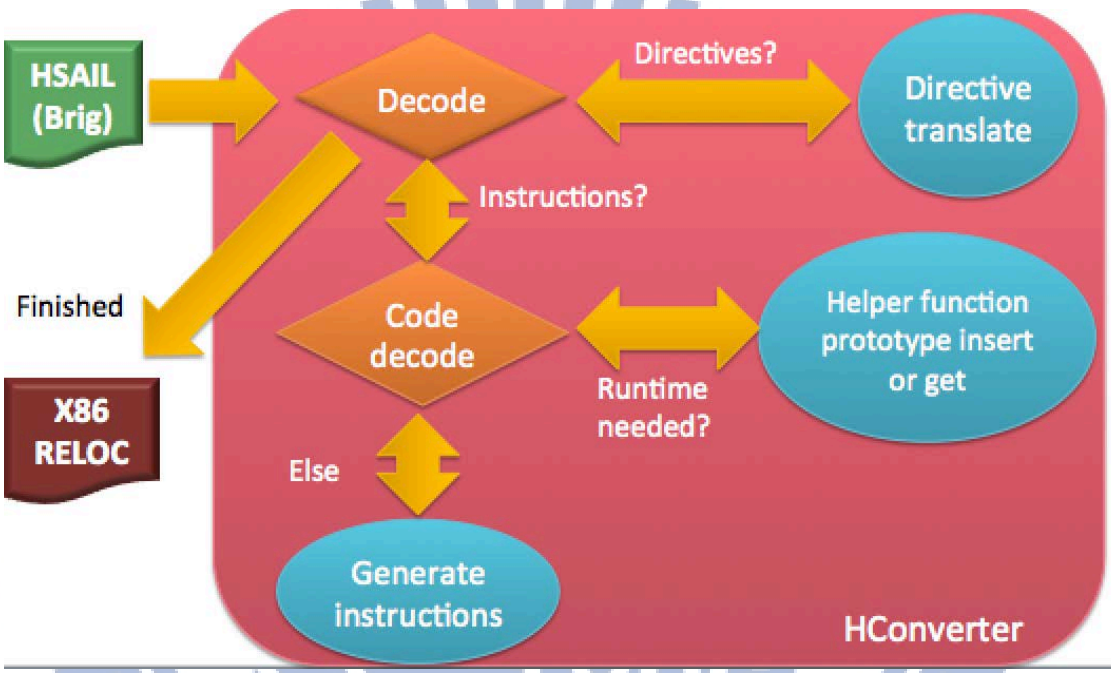

Figure 5.3. The running flow of the HSA Translator starts with the decoding of the directive section. Any instructions in the directive is available code section decoding is invoked. Necessary operands will be accessed through the section pointer in the operations. Finally the re-locatable object is generated after the translation. The current use ISA is X86 because the simulator uses an X86 target for GPU simulation.

5.3 HSA Link-loader

The generated re-locatable object code is placed in the QEMU physical memory. The design of the LLVM infrastructure is for re-targetable use, thus no linking process is provided. In order to have a linked kernel function binary, we implemented a link-loader called HSA link-loader in the QEMU.

The link-loader does the linking of the helper functions. To resolve the addresses of the helper functions, linker scans the symbol table of the re-locatable object code and fills the addresses of the helper functions in the object code. The addresses are found by searching the QEMU binary easily as the link-loader is run within the same QEMU runtime. Addresses remain unchanged in the same runtime.

The helper functions are not preferred to use the in-lining approach because the code sizes of the helper functions are huge. In addition, the functions have to be translated to the LLVM IR first before any linking process. Such translation causes loss information of the global variables and global structures in the QEMU. These reasons give rises to the approach of implementing a link-loader in the QEMU for the HSA simulation.

Because the addresses are not going to be changed during the same runtime, no second linking process should be done to the same object code. Leveraging such linking strategy, the linked object code is stored in the code cache for later use. The linked native kernel function binary is ready to be executed at this moment.

This part is excluded the scope of this paper. On the contrast, we explain the HSA link-loader for giving a more thorough understanding of the translation to the readers.

6. EXPERIMENTS AND DISCUSSION

This section presents the results from the evaluation of the HSA Simulator with the HSA Translator compiled with LLVM version 3.2. Evaluation is performed on two servers both running Ubuntu 12.04.1 LTS. One is powered by four AMD Opteron CPU totally twenty-four cores. Another one is powered by Intel Xeon E5-2620 CPU totally six cores with hyper-threading technology support.

We develop the benchmarks ourselves due to no existing compiler supports the HSAIL backend. The current benchmarks are developed based on some of the benchmarks from the Rodinia Benchmark Suite[10] and the samples in the AMD OpenCL SDK[11]. The benchmarks are modified to use the features of the HSA provided, such as shared virtual memory space.

Benchmark	Base	Domains
KMEANS	Rodinia	Data Mining
Nearest neighbor	Rodinia	Data Mining
Fast Fourier Transform	AMD Samples	Bio-Informatics
Prefix Sum	AMD Samples	Counting
Reduction Sum	AMD Samples	Mathematics
SAXPY	-	Mathematics

Table 6.2 List of the benchmarks.

The elapsed time of programs is less important for measuring as the I/O acquires most of the time. In addition, our simulation is done in system mode. Applications are run above the virtual machine. I/O time varies from time to time with the operating system scheduling and interrupts handling. Thereby, the time listed in the graphs below is the kernel execution time.

Every program time listed below is the average of five execution times. Sizes of the work groups and wave fronts are the same as the original benchmarks. We divide these benchmarks into three cases,

A. Kernels with multi-kernel and no barriers.

B. Kernels with single kernel and no barriers.

C. Kernels with barriers.

6.1 Case A

This case includes the KMEANS. This benchmark has two kernels demonstrating the communication between the CPU and the GPU. Table 6.1 shows the number of operations and the number of helper functions used in the kernels.

Kernel name	Number of operations	Number of helper functions
KMEANS_Swap	32	3
KMEANS_Core	70	4

Table 6.1. Number of operations presents each kernel in KMEANS.

The KEMANS is a multi-kernel program. The agent calls the swap kernel to have parallel data copying. After the swap kernel returns, the agent calls the core kernel several times to perform the computation. The core kernel is called three times in this experiment. The input data size is 116MB.

The result illustrates that the simulator reflects the actual relationship between the number of threads and cores. The performance increases proportionally with the increasing number of threads. Saturation occurs at 24 threads and 8 threads for the 24-core AMD machine and 6-core Intel machine with hyper-threading respectively.

The Intel CPU runs approximately double performance of the AMD CPU. In the circumstances of the number of threads are lesser than twelve, which is the number of the Intel CPU hyper-threading can support, the performance is always double of the AMD CPU. The corner of this situation happens when the number of threads greater than 8. The AMD CPU performance keeps increasing until the number of threads is 24, which is the number of the physical cores.

In addition, the multi-kernel GPGPU program with each kernel has a significant execution time favors the machine with more cores. Whilst the Intel CPU has a stronger computation power, the AMD CPU has more cores for each kernel computation having a higher performance gain in return. Furthermore, favoring the number of cores only exists when the computation time is significant. Furthermore, the data size is critical issue. Data size affects the number of work items needed in the kernel execution directly. The kernel execution time is calculated by the sum of the execution time of each work item plus the synchronization time before returning to the agent. In another words, if the number of the work items is small, the time of the synchronization acquires an inevitable percentage in the kernel execution. In case the synchronization time enjoys no percentage gain from the increasing number of threads, the kernel execution time saturates earlier in return. The KMEANS case demonstrates the percentage of the execution in the work items acquires most of the percentage of the kernel execution time.

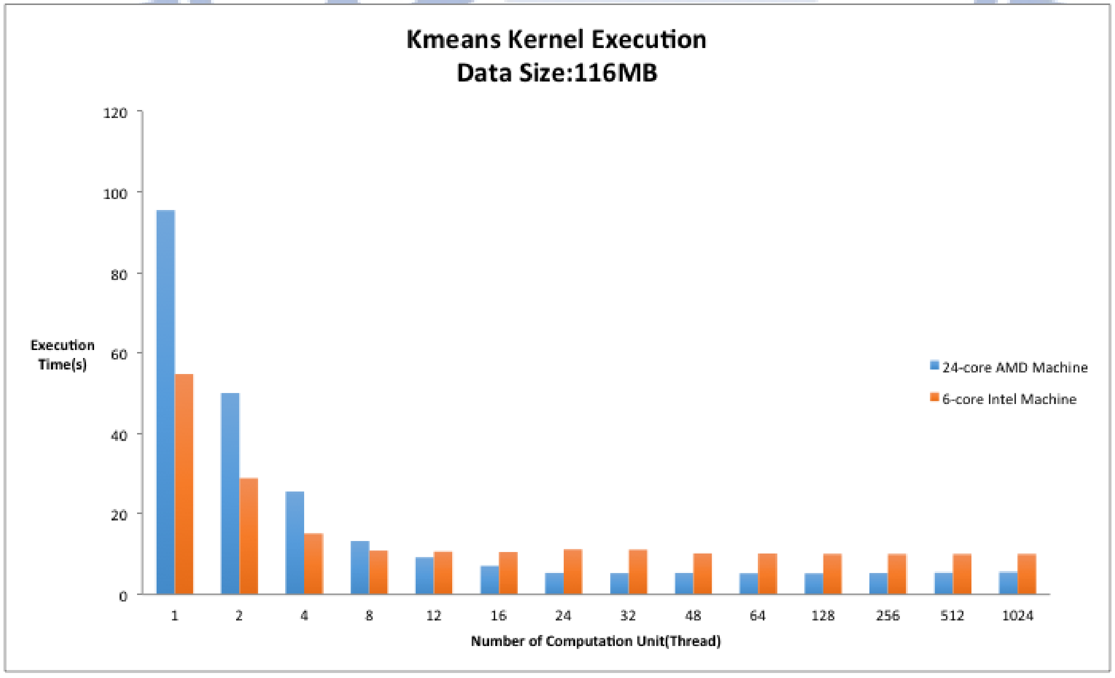

Figure 6.1. The KMEANS running on 24-core AMD machine versus 6-core Intel machine.

6.2 Case B

This case includes the Nearest Neighbor, the Fast Fourier Transform and the SAXPY. All these three programs have one kernel only. The information of the programs is listed in table 6.2.

Kernel name	Number of operations	Number of helper functions
Nearest Neighbor	52	5
Fast Fourier Transform	20	5
SAXPY	22	5

Table 6.2. Number of operations presents in Nearest Neighbor, Fast Fourier Transform and SAXPY.

The Nearest Neighbor kernel computes the distances between nodes and stores them into an output array. After kernel returns, agent computes the shortest in between. The Fast Fourier Transform performs a mathematical computation of Fourier Transformation. The SAXPY computes the multiplication and addition of two long vectors. The input data sizes are 768KB, 256MB and 256MB respectively.

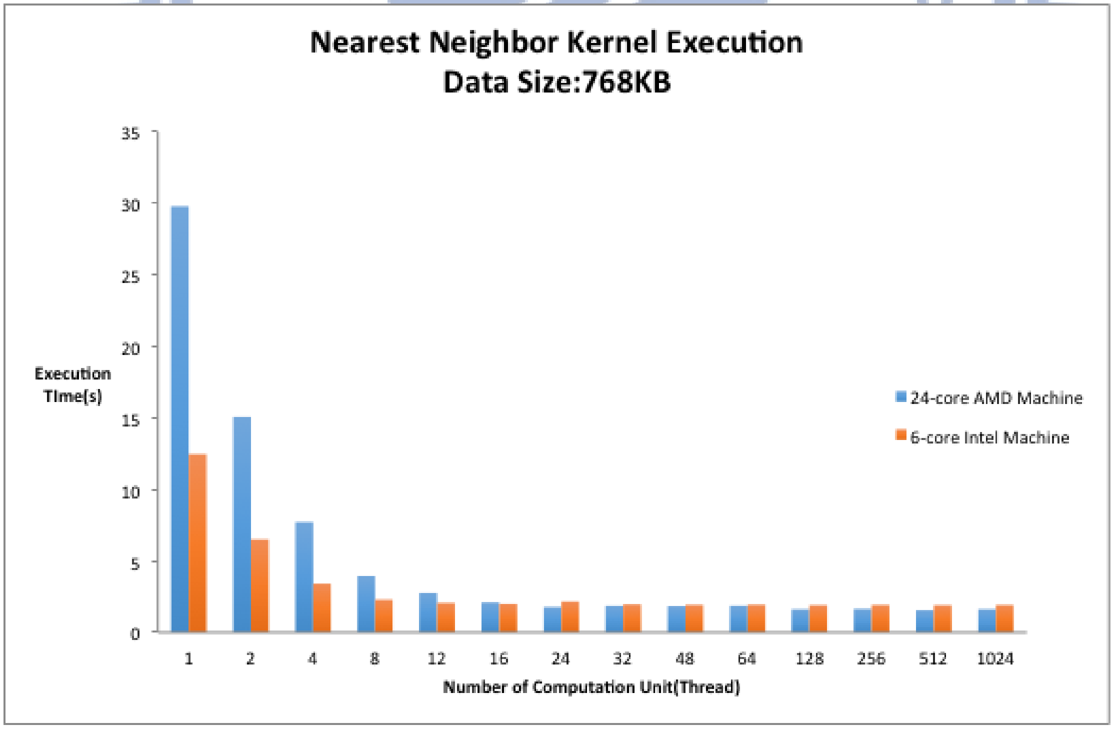

Figure 6.2. The Nearest Neighbor running on 24-core AMD machine versus 6-core Intel machine.

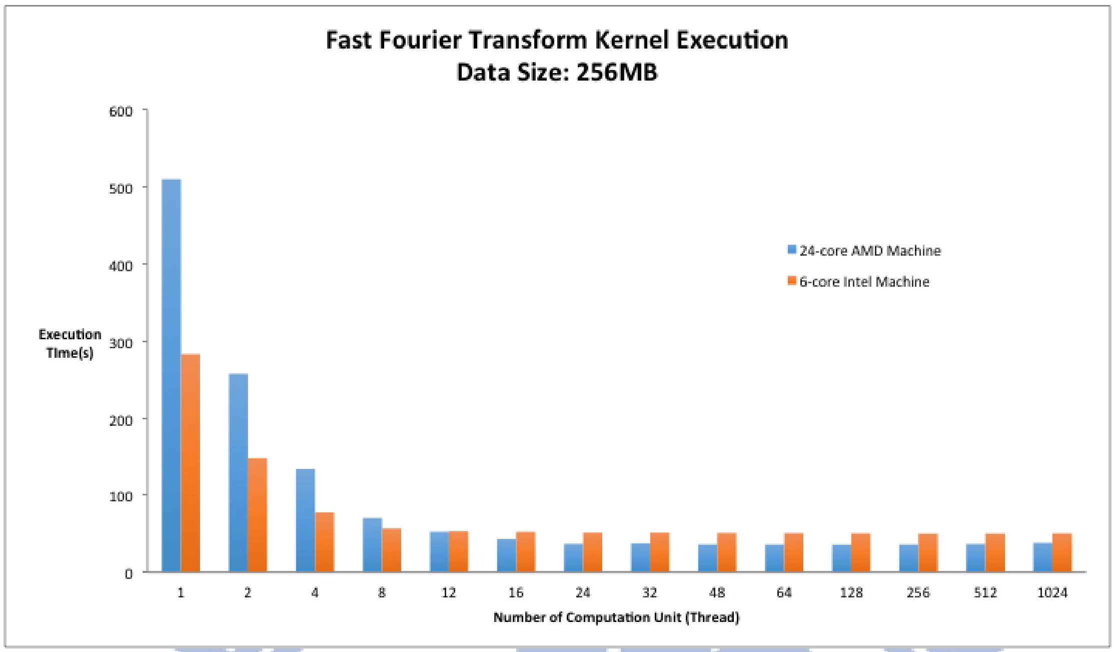

Figure 6.3. The Fast Fourier Transform running on 24-core AMD machine versus 6-core Intel machine.

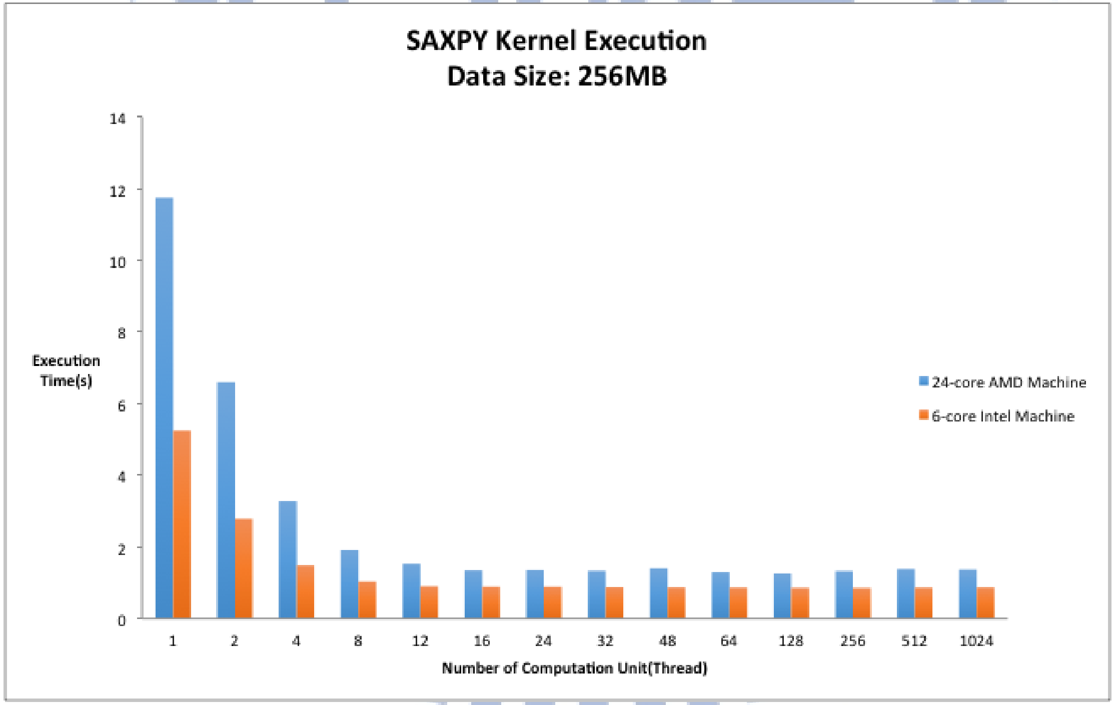

Figure 6.4. The SAXPY running on 24-core AMD machine versus 6-core Intel machine.

The performance of the Nearest Neighbor and the Fast Fourier Transform are the same as the KMEANS. The Intel machine has a better computation power that makes the performance beating the AMD machine before the hyper-threading saturates. With more threads can support, even though the AMD machine has less powerful computation unit, the large data size leads to a large number of work items which leverage the increasing number of threads. Whilst the cores are less powerful, the individual work item execution time is improved having a overall improvement of the Intel machine.

Comparing to the result of the SAXPY, the SAXPY illustrates a difference. The data sizes of the SAXPY and the Fast Fourier Transform are the same. Instead, the performance gain from the 24-core is less than the 6-core. The reasons of this phenomenon are,

1. The computation of the SAXPY is simpler than the Fast Fourier Transform. Though there are more operations in SAXPY, the cycle cost computations such as the arithmetic and memory operations own 72.7%. The percentage of the Fast Fourier Transform is 90%. There are more chances for the AMD machine to speed up in each work item in the Fast Fourier Transform, as the computation operations are much more.
2. The multiplication and division operations are more cycle costing than the addition and the subtraction. The percentage of the multiplication and the division operations are 4.5% and 20% in the SAXPY and the Fast Fourier Transform respectively. Such percentage cost much more cycles with more threads parallel computing the work items, much more overhead due to the limited resource of the CPU computation unit can be hidden.

6.3 Case C

This section demonstrates result of the kernels with barriers. The benchmarks are the Reduction Sum and the Prefix Sum.

The Reduction Sum performs a Summation of a vector with a parallel reduction in each work group adding a part of data. Summation in each work group needs to wait for the duplication of the data to a buffer as well as the synchronization of all the work groups before adding up all the result. Thereby, there are two barriers present in the Reduction Sum kernel.

The Prefix Sum performs a Summation of the data in the current index and the previous index. When summation preforming in index (i), the summation of the index (i-1) must be ready. According to the parallel Prefix Sum algorithm, there are two barriers existing.

Kernel name	Number of operations	Number of helper functions
Reduction Sum	109	37
Prefix Sum	108	21

Table 6.3. Number of operations presents in Reduction Sum and Prefix Sum.

The input data sizes of the Reduction Sum and the Prefix Sum are 256KB and 128KB respectively.

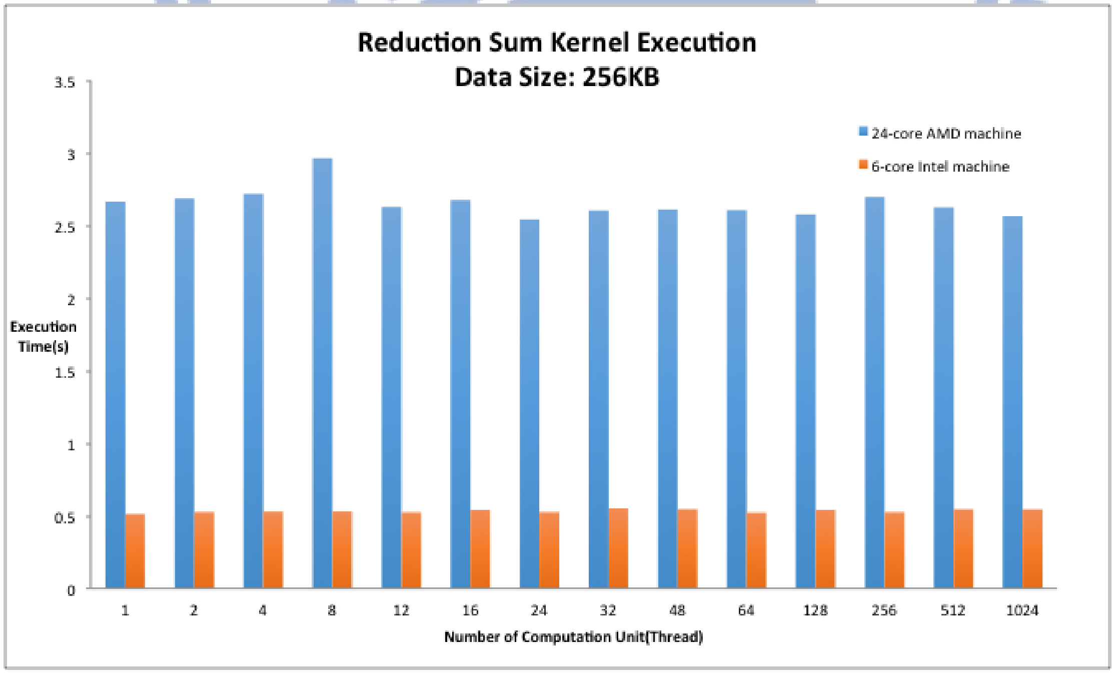

Figure 6.5. The Reduction Sum running on 24-core AMD machine versus 6-core Intel machine.

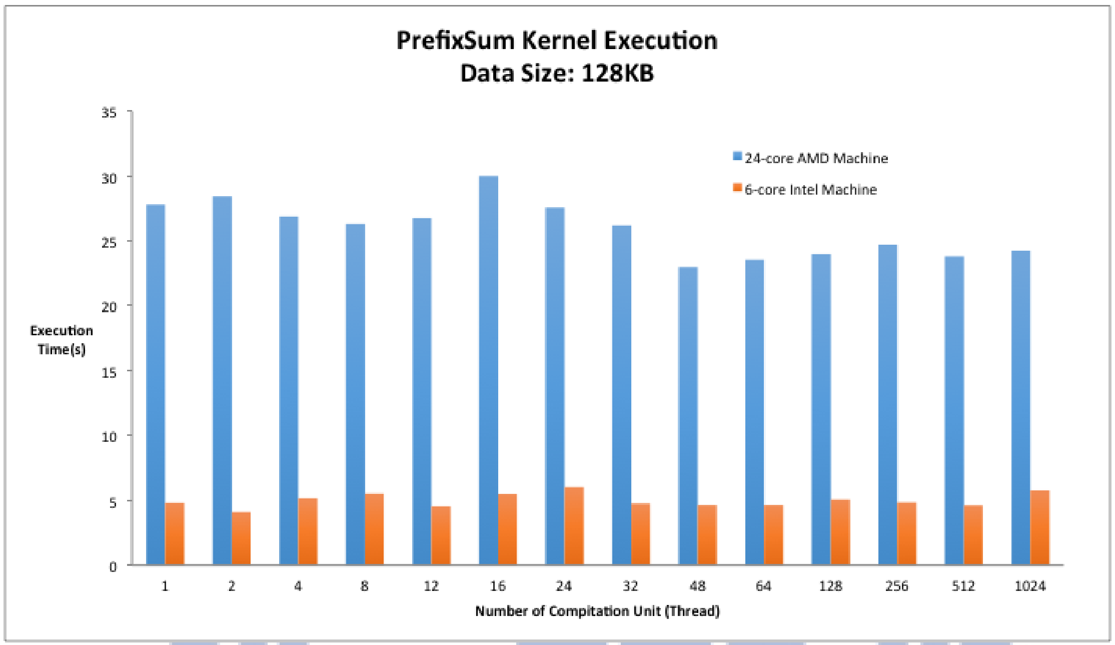

Figure 6.6. The Prefix Sum running on 24-core AMD machine versus 6-core Intel machine.

The Reduction Sum benchmark evaluates the work group barrier synchronization of the HSA Simulator. There is no performance gain with the increasing number of threads due to the two barriers in the work group level. We leverage the multi-threads from parallel execution of the work items in the Case A and B. On the contrast, in current case we are hard to benefit from the threads because the barriers always stop the work group from parallel processing until all the work groups have processed to the synchronization point. Whilst the threads within a work group process concurrently before the barriers, only one work group is allowed to process in each work group dispatching loop instance. Theoretically, the execution time with different number of threads in this case should be no change. However, due to the unlikely dispatching and synchronization of the pthreads in the operating system every time, the graph shows like a zigzag in return.

Unlike the Reduction Sum, the Prefix Sum illustrates an extreme case of the use of barriers, which helps to evaluate the work item level barrier of the HSA Simulator. The Reduction Sum needs to synchronize due to the problem of the race condition, which is necessary to prevent two or more threads in rewriting the same data. However, the Prefix Sum is the problem of the dependency within the work items. Every work item must be done after the previous work item finishing the data write in. Thereby, such extreme case of synchronization demonstrates as the graph shows. The zigzag like graph can apply the explanation from the Reduction Sum.

Conclusions

This paper presents a translator of a re-targetable IR HSAIL to the native code called the HSA Translator for the fast simulation using in the HSA Simulator. The HSA is the software development solution carried out from the AMD aimed to ease the GPGPU programming. With the shared memory model and the vector instructions, the HSA provides an easier way to the GPGPU programmers. The HSA Translator is implemented based on the LLVM Infrastructure and performs a runtime translation for the HSAIL to the native re-locatable object. In addition, the output of the HSA Translator is a re-locatable object due to the absent of linker support in the LLVM. We cover this linker problem with the HSA Link-loader which is not the achievement of this paper. However, no existing compiler can generate the HSAIL code. Thereby, another achievement of this paper is to implement a Brig generator for the generation of the HSAIL binary format called the Brig from the HSAIL text format. The text format HSAIL is written manually and five of the six programs are one of the achievement of this paper. The HSA Translator works as a finalizer in the HSA Simulator. Experimental result shows the evaluation of the HSA Translator. Two different servers are used to evaluate the HSA Translator. Two servers with different number of cores show the scalability. With the memory bandwidth difference, the memory scalability is illustrated to have performance difference.

Reference

[1] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung, "PQEMU: A Parallel System Emulator Based on QEMU," icpads, pp.276-283, 2011 IEEE 17th Internatio-nal Conference on Parallel and Distributed Systems, 2011.

[2] A. Bakhoda, G. L. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software, April 2009.

[3] Bor-Yeh Shen, Pei-Shiang Hung, Jyun-Yan You, Wei Chung Hsu, Wuu Yang, National Chiao Tung University, Taiwan, LLBT: and LLVM-based static binary translator, Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems.

[4] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Princeton University, Brown University, Automatic CPU-GPU Communication Management and Optimization,

[5] Chris Lattner, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, UIUC, CGO'04.

[6] Sylvain Collange, Marc Daumas, David Defour, David Parello. Barra: A Parallel Functional Simulator for GPGPU. 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2010.

[7] Gregory Diamos, Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems, Georgia Tech, PACT 2010.

[8] Vitaly Zakharenko, FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems, Toroto MS Thesis, 2012.

[9] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, Wei-Chung Hsu, LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends.

[10] https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page

[11] http://developer.amd.com/tools-and-sdks/heterogeneous-computing/ amd-accelerated-parallel-processing-app-sdk