University of Alberta

Jit4OpenCL: A Compiler from Python to OpenCL

by

Xunhao Li

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2010
Edmonton, Alberta

Examining Committee

José Nelson Amaral, Computing Science

Duane Szafron, Computing Science

Marek Reformat, Electrical and Computer Engineering

To my beloved family.

Abstract

Heterogeneous computing platforms that use GPUs and CPUs in tandem for computation have become an important choice to build low-cost high-performance computing platforms. The computing ability of modern GPUs surpasses that of CPUs can offer for certain classes of applications. GPUs can deliver several Tera-Flops in peak performance. However, programmers must adopt a more complicated and more difficult new programming paradigm.

To alleviate the burden of programming for heterogeneous systems, Garg and Amaral developed a Python compiling framework that combines an ahead-of-time compiler called unPython with a just-in-time compiler called jit4GPU. This compilation framework generates code for systems with AMD GPUs. We extend the framework to retarget it to generate OpenCL code, an industry standard that is implemented for most GPUs. Therefore, by generating OpenCL code, this new compiler, called jit4OpenCL, enables the execution of the same program in a wider selection of heterogeneous platforms. To further improve the target-code performance on nVidia GPUs, we developed an array-access analysis tool that helps to exploit the data reusability by utilizing the shared (local) memory space hierarchy in OpenCL.

The thesis presents an experimental performance evaluation indicating that, in comparison with jit4GPU, jit4OpenCL has performance degradation because of the current performance of implementations of OpenCL, and also because of the extra time needed for the additional just-in-time compilation. However, the portable code generated by jit4OpenCL still have performance gains in some applications compared to highly optimized CPU code.

Acknowledgements

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor José Nelson Amaral. Without your constant guidance, inspiration, supervision, support in researching and thesis writing, this thesis would not have been completed. You not only taught me the knowledge and methodology but also encouraged me to be a person of persistence, industriousness and integrity. I would have lost myself without your generous help.

I am also grateful to Rahul Garg. Thanks to the unPython + jit4GPU framework you constructed, I did not need to start from scratch. Your brillant idea makes a solid foundation to our work, I am heartily thankful to you.

I would also like to thank Professor Duane Szafron from Computing Science and Professor Marek Reformat from Electrical Engineering for helping me examine my work. Your suggestions and advices are of great help to me.

I especially want to thank my mentor, Professor Yu Zhang in USTC, for your great help in both my technical and personal development. She opened my mind and told me to pursue whatever I love, and offered advice and suggestions whenever I need them. I treasure those time we worked and shared together.

Last but not least, I wish to thank my parents, for bearing me, raising me and sheltering me. Nothing can substitute the love and comfort they give me. I love you both. To you I dedicate this thesis.

Contents

List of Figures

List of acronyms and symbols

API	Application Programming Interface
CAL	Compute Abstract Layer language
CPU	Central Processing Unit
CSLMAD	Constant Stride Linear Memory Access Descriptor
DRAM	Dynamic Random Access Memory
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
JIT	Just-in-time
OpenCL	Open Computing Language
RAM	Random Access Memory
LMAD	Linear Memory Access Descriptor
CSLMAD	Constant-Stride Linear Memory Access Descriptor
RCSLMAD	Restricted Constant-Stride Linear Memory Access Descriptor
SIMD	Single Instruction Multiple Data
SM	Stream Multiprocessor
SP	Stream Processor
SPMD	Single Program Multiple Data
STMD	Single Thread Multiple Data
TPC	Thread Processor Cluster

Chapter 1

Introduction

Scientific computing is a field that focuses on the execution of numerical computation to solve scientific and engineering mathematical problems. Ever since computers were used to model and simulate scientific problems, scientists and engineers were constantly demanding greater computing power.

Comutation on computers involves a software platform and a hardware platform.[1] On the software platform side, high-productivity scientific programming languages, such as FORTRAN and MATLAB, are designed to enable people to implement high-performance software with a reasonable amount of effort. The constant development of compiler optimization delivers higher performance by reducing the amount of time required to execute computer programs.

Thanks to abundant support and extensions, the Python language has become one of the major language choices for scientific computing. Python features a plethora of library support, simple syntax, a flexible programming model, lightweight execution, agile development, and cross-platform portability. Using Python as the programming language has some advantages over competing programming languages and platforms for scientific computing. With the help of an extension library called NumPy, Python can support flexible multi-dimensional array abstractions that are essential to scientific computing. Drawbacks of using Python for scientific computing include the interpretation overhead that limits the execution speed and the lack of support for parallel execution. To eliminate interpretation overhead for scientific computation, numerical libraries, such as NumPy, are written in the C programming language and are linked to interpreted Python programs.

On the hardware platform side, conventional hardware development, by increasing CPU clock rate in order to gain higher computing ability, is approaching the possible limit of clock rate, forcing micro-chip engineers and computing scientists to adopt multi-core designs to improve performance. As the number of cores constantly increases, a revolution of

[1] "Computing on computers" sounds redundant today, but there was a time when a "computer" was a person, and the "computing" was done with pencil and paper. Such computing was not only for accounting but also for scientific purposes.

computing is emerging. Programmers need to adapt to this programming paradigm shift in order to exploit the potential of hardware.

An aggressive example of parallel execution occurred in graphic processing units (GPU). GPUs are designed to be processors dedicated to graphic processing. Its architecture is drastically different from conventional CPU. Recently GPUs have evolved from a fixed-function graphical processing hardware pipeline to a programmable general-purpose graphic processing unit (GPGPU), allowing programmers to implement software code that can make use of the computing ability of the GPU. However, the huge architecture difference demands that programmers adopt a stream processing paradigm, which greatly increases the difficulty of programming.

A computing platform consisting of CPUs and GPUs is a heterogeneous computing system. Thanks to the many cores in a GPU chip, GPU excels in numerical computing that is highly parallelizable. In modern heterogeneous computing, GPUs undertake the majority of the computation tasks. However, programming for heterogeneous computing systems requires fully understanding of the application and the programming paradigms involved. In the case of GPGPU the random data access of the CPU and the streaming data of the GPU need to be integrated. Stream processing can handle a subset of scientific applications efficiently, but programming an stream-oriented hardware efficiently is more difficult than programming a random-access hardware. This programming-paradigm shift is a major obstacle for beginning GPGPU programming.

Some researchers are working to eliminate the gap of shifting to a new programming paradigm when programming for GPGPUs [35][38][15]. They adapt mainstream programming models to GPGPU programming to reduce the efforts required and the complexity of programming for GPGPUs. An example is the unPython and jit4GPU compilation framework developed by Garg and Amaral [12]. UnPython compiles a subset of annotated Python program into C++ and OpenMP code. Jit4GPU interprets the program at runtime and handles program execution on AMD GPU. Although unPython and jit4GPU can deliver significant performance improvements, it comes with the cost of less portability, limiting its target platform to a system containing AMD GPUs. This thesis presents an extension to jit4GPU, named jit4OpenCL, that interprets the program into an industry-standard language for heterogeneous computing: OpenCL. This change will make jit4OpenCL executable on platforms that support OpenCL.

1.1 Contributions

This thesis presents the following contributions:

- an extension to the existing RCSLMAD array access analysis usable in SIMD program

generation and optimization.

- a new just-in-time compiling framework for generating OpenCL for annotated Python program on the fly.
- an experimental evaluation showing that, with the GPU assistance, annotated Python programs can result in performance gain up to 91 times compared to hand-written CPU code in some applications.

1.2 Thesis Organization

The first two of the following chapters serve as a review of background material. Chapter 2 briefly introduces GPGPU computing, including concepts, the nVidia Tesla architecture, CUDA programming model and OpenCL platform. Chapter 3 describes the Python language, NumPy scientific computing package and unPython compiling framework. Chapter 4 is a formal statement of the array access analysis problem and the solution. Chapter 5 states the methods and algorithms used by jit4OpenCL to generate OpenCL code. Chapter 6 evaluates and analyzes the experimental result. Chapter 7 briefly examines related work. Chapter 8 concludes this thesis.

Chapter 2

Background on CUDA and OpenCL

This chapter presents a brief introduction of computing on mainstream nVidia GPGPU (General Purposed GPU), including an anatomy of GPGPU hardware architecture, two GPGPU programming frameworks (CUDA and OpenCL), along with examples as supplementary materials for reader's reference. Throughout the chapter is the explanation of the difference of programming paradigms between the conventional CPU programs and the GPGPU programs for those readers that are familiar with CPU programming. Our compiling framework's backend targets the code generation on OpenCL; thus understanding the programming platform of GPGPUs is fundamental to our analysis.

2.1 Programming for GPUs

2.1.1 GPU Characteristics

GPU is the place for rendering process. Rendering is the computing process that generates digital image/video from an object model describing the objects shape, appearance, geometry, texture and lighting. Rendering may also be refereed to as the process of generating digital graphic effects. It is a very common computing task in computer-aided simulation and design, video gaming, movie creation, and others.

Commodity graphics hardware, such as nVidia and AMD GPUs, have a graphic pipeline to perform rasterization-based rendering. For a graphic pipeline hardware, the input of the rendering process to GPU is a set of vertices, indicating the shape and size of the object. Those vertices then must undergo linear transformation and per-vertex lighting in vertex-shader units to geometrically shape the polygon of the object and make the object lit according to the lighting atmosphere. The object is then clipped if necessary, then rasterized on Render Output Units into fragments. Those fragments may undergo texturing on Texture Mapping Units to add textures to the fragments. Finally, those fragments are

stored in a buffer frame for display. Figure 2.1 shows the process.

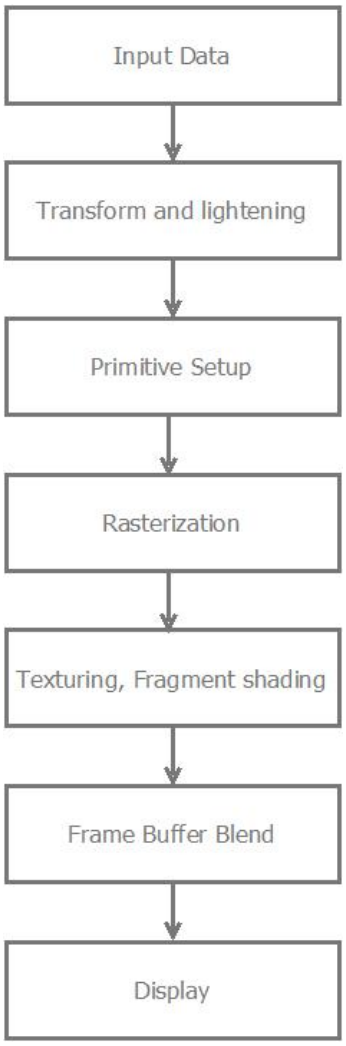

Figure 2.1: Graphic Rendering Pipeline

Because most of the computing described above is based on individual vertex, and each of the vertices shares a same processing pipeline, the pipeline can exploit the benefits of the **stream processing**[1] to accelerate the computing by parallelizing the operations. GPU designers are fully aware of this situation and commodity GPUs are designed to be a parallel stream processing computing environment that are Single Instruction Multiple Data (SIMD) capable.

[1]stream processing is a restricted parallel programming paradigm that let a stream of data goes through a processing pipeline consists of a sequence of operatons. There are no dependencies between two different data element in the stream, which makes the stream processing fully parallizable.

2.2 nVidia GPGPU Hardware

A General-Purpose GPU (GPGPU) is programmable to acclerate a wide range of applications besides graphic processing. NVidia's Tesla series GPU is the first GPU series from nVidia that supports general-purpose computing. Different from conventional CPUs, the Tesla GPU is a large-scale parallel processing unit with hundreds of computing cores that can perform computation at the same time.

2.2.1 The Tesla Architecture

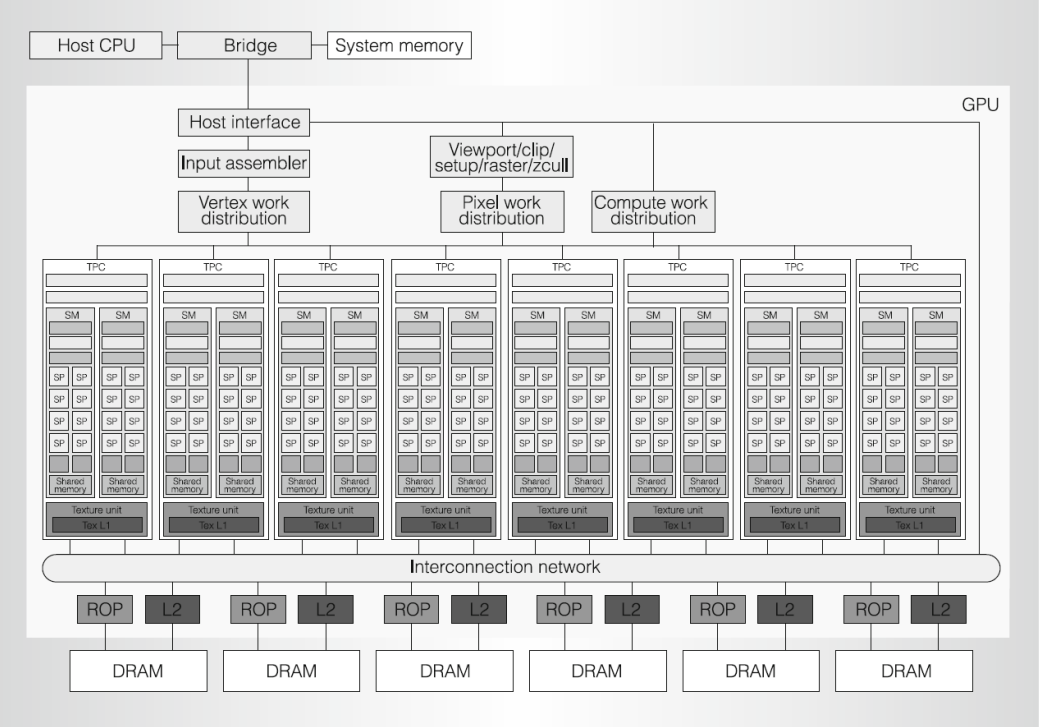

Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

Figure 2.2: Tesla Architecture (from [16])

Figure 2.2 shows a block diagram that describes the internal structure of a Tesla GPU. Basically, in a typical Tesla GPU there is an array of Thread Processor Clusters(TPC), which is shown in the middle of the figure. Within each TPC there are 2 or 3 Stream Multiprocessors (SM). About 8 to 12 Stream Processors (SP) are encapsulated in each SM, along with at least two Super Function Units (SFUs), Stream Processors and Super Function Units are small cores where computing actually happens. There is also an instruction fetch/dispatch scheduler in a Thread Processing Cluster that reads, decodes, and schedules instruction execution for each Stream Processor inside. There are also register file space

for storing registers, local memory space for storing spilled registers when the register file cannot hold all the registers. In order to serve as an intermediate software-managed cache hierarchy for fast memory access, there is an on-chip, small memory storage space ranging between 16KB to 32KB in each Stream Multiprocessor called shared-memory space. Finally, the GPU is connected with an Interconnection network to the off-chip large-capacity DRAM memory, which is used to load/store processing data.

2.2.2 Execution

Tesla is designed to support parallel threading; its hundreds of Stream Processors enables a Tesla GPU to have more than ten thousand threads issued concurrently and thousands of threads running in parallel at a same time. To reduce the complexity of hardware for handling thousands of threads, a simplified scheduling schema is used: Instead of handling each thread individually, the scheduler handles one "warp" at a time, which consists of a group of threads. Scheduling by warps reduces the hardware scheduling overhead, thus reducing the complexity of the hardware design. A thread warp has the following characteristics:

- consists of 32 threads.
- threads in a warp share a single program counter, thus they progress together. For example, if a warp is chosen to be executed for the next cycle in an 8-core SM, then it will take 4 consecutive cycles to let all the threads proceed one instruction.
- there can be several warps executing concurrently on the same SM. When a warp is executed, the other warps will wait for their turn.

A thread running on GPUs can only access the memory space in off-chip DRAM on the same graphic card and the shared-memory space . Threads that are executing on the same SM share a limited-sized on-chip register file. Using too many registers may cause threads to spill the values from the register file to a local memory.

2.3 CUDA GPU Computing Framework

Introduction to CUDA Architecture

As a stream processor, the architecture of the Tesla GPU is drastically different from the architecture of conventional computer. As a result, traditional programming models are not suitable for application implementation on Tesla GPUs. The Compute Unified Device Architecture (CUDA) was developed to help programmers implement software code on nVidia's Tesla GPUs.

CUDA implements a Single-Instruction-Multiple-Thread (SIMT) programming model to fit with the parallel GPU architecture. SIMT is an abstraction of the Single-Instruction-Multiple-Data (SIMD) programming model. The difference is that, in SIMD the CPU can

issue vector instructions, but in SIMT vectors of threads are issued rather than vectors of instructions. In general, SIMD is not scalable but SIMT is. When programming in an SIMD model, programmers need to specify the instruction width (the vector width), or the width is an characteristic of the hardware. In SIMT, programmers only describe the behaviours of the threads, CUDA will issue as many threads as the width of the vector, which can be decided at runtime. This dynamic-thread issuing technique makes a compiled program compatible with new hardware models released in the future, as long as the architecture remains the same. Figure 2.3 shows the processing flow of the CUDA platform.

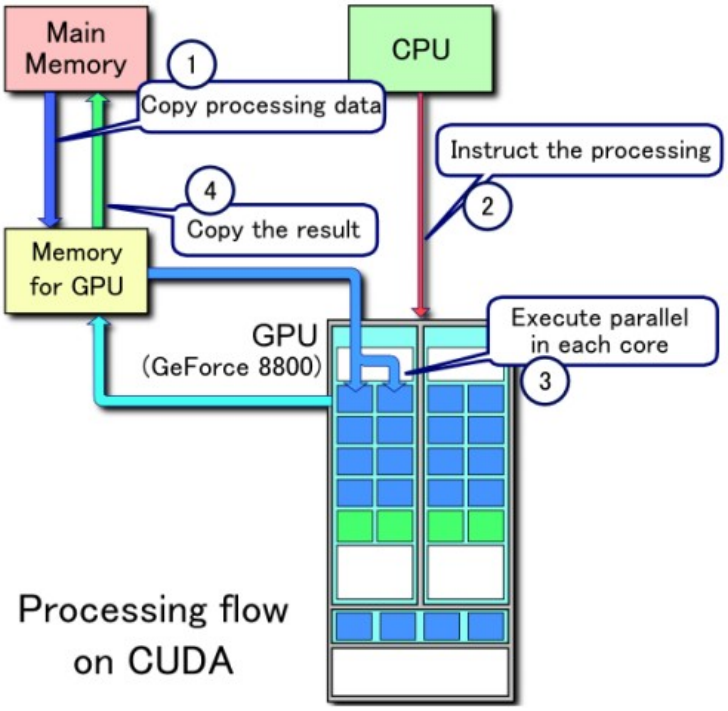

Figure 2.3: CUDA Processing Flow, (extracted from `http://en.wikipedia.org/wiki/CUDA`, Mar 11th, 2010)

In Figure 2.3, the actual computing stage (steps 2 and 3) requires the overhead of copying processing data to GPU memory and copying the result back to main memory (CPU memory) (step 1 and step 4). However, often the time used on this extra work can be compensated by the fast processing speed of GPU core execution.

CUDA Programming Model

CUDA models a heterogeneous architecture that consists of a *host* and one or several *devices*. A *host* is a conventional CPU, and a *device* is a General Purpose GPU (GPGPU). The *host* and the *devices* have to collaborate in order to achieve successful CUDA program execution. Due to the hardware design, the CUDA programming model on Tesla architecture is only suitable for computing tasks that adopt the stream-processing programming paradigm.

Tasks that cannot be parallelized in stream processing are not suitable for execution on this platform.

A complete CUDA program consists of one or more phases. Each phase is implemented either in serial *host* code or parallel *device* code. Phases implemented in *device* code execute on one or several *devices*, while those in *host* code only execute on the *host*. It is a programmer's duty to specify whether a computing phase exhibits data parallelism. A data-parallel phase must be implemented in *device* code rather than *host* code. Such *device* code is called a "kernel function" or just "kernel".

The relationship between a CUDA host and a CUDA device is a master/slave relationship. Device code cannot be invoked outside of host code. Therefore, a CUDA program always starts serially from the host. Before invoking a kernel, the host prepares the execution on GPU by transferring processing data to the GPU memory, setting up execution parameters and then invoking the device code asynchronously. After the device code execution terminates, the host takes over the execution again until the next kernel is invoked. This processing flow is shown in Figure 2.3.

CUDA Kernel Execution Organization

The launching of *device* code in CUDA requires the launching of many kernel threads. In a typical CUDA program, when the Step 1 in Figure 2.3 is finished, the host launches kernel threads for computing. These threads are organized into a 1, 2 or 3-dimensional array. The array is further divided into a grid (1, 2, or 3-dimensional), which is comprised of many thread blocks as shown in Figure 2.4.

CUDA's programming model is tailored to be capable of efficient execution on Tesla Architecture. The execution follows these rules:

- All the threads from a thread block are executed on a single Stream Multiprocessor (SM) in a Thread Processor Cluster (TPC).
- The GPU scheduler divides the task into a queue of thread blocks. An SM accepts one thread block at a time. Once all the threads in a thread block finish their execution, the scheduler immediately assigns a new thread block to the SM from the queue, until the queue is empty. The sequential execution of thread blocks in an SM guarantees that each thread block has exclusive access to shared-memory space while executing.
- The execution of threads follows the rules described in Section 2.2.2.

Threads in a thread block can communicate by:

- synchronizing execution to avoid hazardous memory accesses.
- accessing the on-chip low-latency shared memory space.

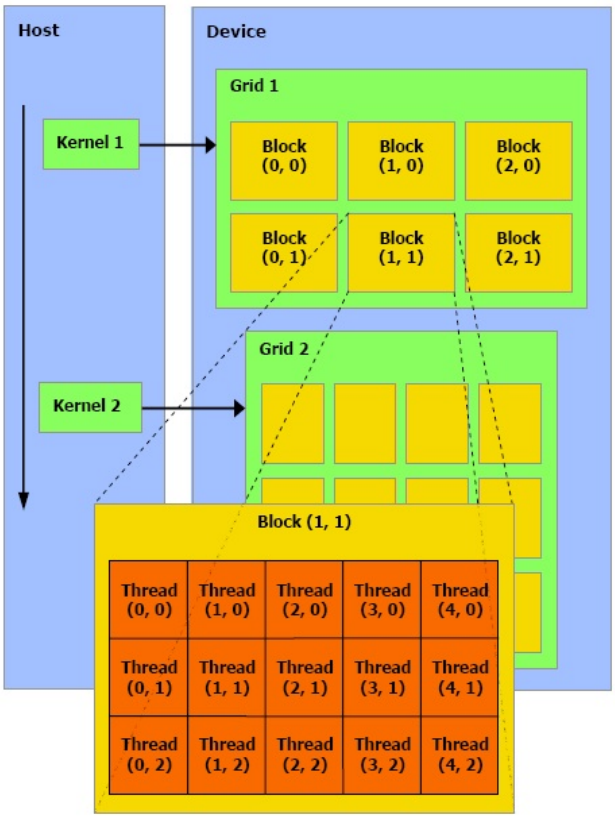

Figure 2.4: The Organization of CUDA threads (from [34])

- accessing the off-chip high-latency global memory space.

The only means of communication available for threads in different thread blocks is the off-chip global memory.

Threads and thread blocks have their own IDs. The index representing the position of a thread in the array is called *global ID*. The index representing the position in the thread block is called *local ID*. The index of a block in the grid is called *block ID*. Those IDs may be 1, 2 or 3-dimensional, depending on the thread grid configuration. Figure 2.4 shows an example where the launched kernel 1 has a 2-dimensional thread grid, which is divided into 2×3 blocks. The figure also shows the inside of a block with block ID (1,1) (the indexes of the first dimension and of the second dimension are both 1), which consists of 5×3 threads. Their local IDs are also 2-dimensional, indicating their position in the block.

2.4 OpenCL Framework

One significant drawback of the CUDA architecture is that it is developed exclusively for the nVidia Tesla architecture. Programs written in CUDA are not portable to any other heterogeneous computing architectures, such as platforms that have AMD GPUs. Due to

the lack of portability, programmers have to implement several versions of each computing kernel if to implement a same program for different computing platforms.

In 2009, the industry announced a new open, cross-platform, royalty-free computing framework called OpenCL. OpenCL was proposed by Apple, and developed in collaboration with nVidia, AMD, Intel, and IBM, *etc.* Now OpenCL is under the management of the non-profit organization Khronos Group, and implementations on different platforms are emerging quickly since its specification 1.0 announcement in 2009.

Mac OSX 10.6 Snow Leopard comes with native support for OpenCL 1.0, which is the first operating system that supports OpenCL. nVidia has released its OpenCL implementation to registered GPU programmers starting from mid-2009, and AMD published its OpenCL support for its CPUs later in 2009 and for its GPUs in early 2010. Thus, OpenCL is now available for tens of millions of personal computers.

2.4.1 OpenCL Portability

Generally speaking, the OpenCL Architecture is a framework for parallel programming, including a C99-like language, a set of OpenCL API, a set of libraries and a runtime support system. OpenCL represents programs at a higher level of abstraction when compared with a program written in CUDA.

As a general, standard framework designed for heterogeneous computing, OpenCL creates a general parallel computing framework used for CPU-GPU computing system, regardless of the 3D graphics API, such as OpenGL or DirectX. Still, OpenCL programs can execute on homogeneous computing systems containing only GPUs. OpenCL supports parallel computing natively. Programs implemented in OpenCL are extremely portable. OpenCL programs can be, without any modification, compiled to and executed on such hardware platforms as mobile devices, ordinary desktop computers or dedicated personal supercomputers, as long as the platform has OpenCL support.

2.4.2 Similarities with CUDA

OpenCL and CUDA share the following features.

- They are both heterogeneous computing platforms. They both have a *host*, and the *host* takes control of a set of *devices*.
- They are both parallel computing platforms adapting SIMT paradigm. In OpenCL, a program issues many thread instances for execution; these thread instances, called *work-items*, and *work-items*, are grouped into *work-groups*. These concepts correspond to *threads* and *thread-groups* in CUDA.

- In OpenCL, all work-items in a work-group have access to dedicated fast local-memory space. In CUDA, all threads in a thread group have access to dedicated shared memory.
- Both OpenCL and CUDA have an index space for identifying work-items/threads. Both index spaces are divided into a grid. Each element in the grid is a work-group (OpenCL) or a thread-group (CUDA).
- Their kernel instances have the same restrictions:
  - Can only have access to GPU memory space (shared memory space and global memory space).
  - Can only be declared to have a void return type.
  - Cannot have recursive structure.
  - Cannot have static variables.
  - Cannot have variable number of arguments.

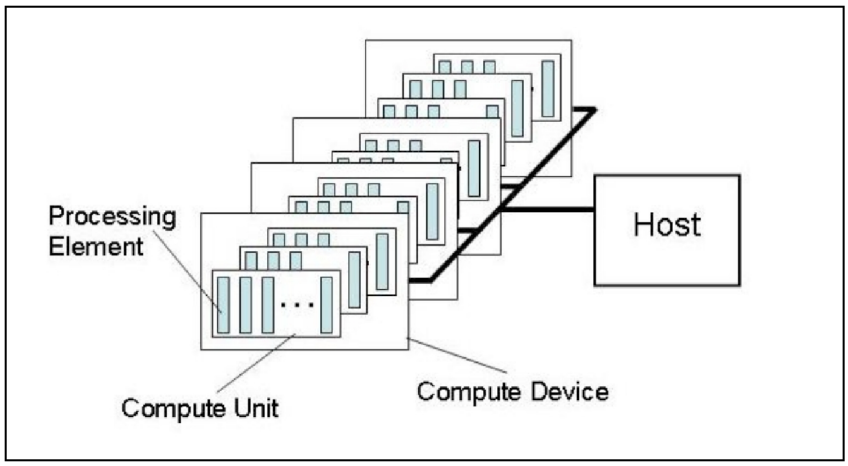

Figure 2.5: OpenCL Platform Model (from [18])

2.5 OpenCL in Examples

This section uses a program example to show how to implement an OpenCL Program. Consider the computation of the matrix multiplication ($C = A \times A$). For an element in C, say, (i, j), the following computation must be performed:

$$C(i,j) = \sum_{k=1}^{N} A(i,k) \times A(k,j)$$

The host code is straightforward. First the host program allocates two memory buffer on device — a memory buffer refers to a contiguous memory space in global memory. In this

Code 2.1: OpenCL Matrix Multiplication

```
1 __kernel void multiply(__global float *a, __global float *c,
    int N){
2     int row = get_global_id(0);
3     int col = get_global_id(1);
4     float sum = 0.0f;
5     for (int i = 0; i < N; i++){
6       sum += a[row*N + i]*a[i*N+col];
7     }
8     c[row*N + col] = sum;
9 }
```

example there will be one memory buffer for array A and another for array C. Then, the CPU executes instructions to copy array A from the host memory to device memory so that the computing units can have access to A. After the transfer, the CPU sends instructions to computing units to start the execution of a kernel function. The computing units calculate every element of array C and store the result into device memory. Once all the computing is done, the result stored in the device memory is transferred back to host memory, and the computing on the device terminates while the host can continue to execute. Figure 2.6 illustrates the decomposition of the whole process.

Clearly for any two different elements in C, $C(a, b)$, $C(c, d)$, there is no dependency between the two, thus the computation follows the stream processing paradigm. Therefore, we can create $N \times N$ threads, each thread computes a single element in C. Assuming the array is row major, Code 2.1 shows the kernel function for computing C.

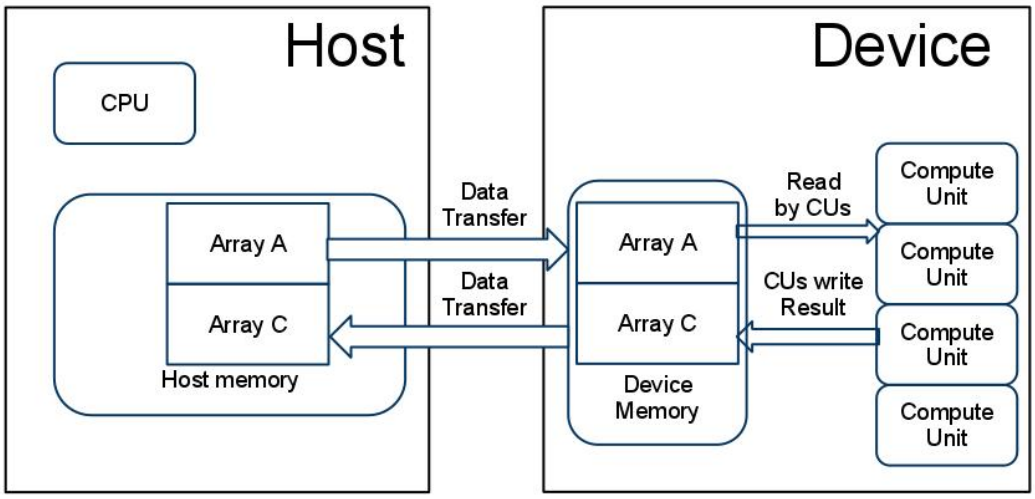

Figure 2.6: Example OpenCL Program Flowchart

In the kernel function shown in Code 2.1, lines 2 and 3 get the thread's first and second

dimension IDs, respectively. Lines 5-7 accumulate the sum. Line 8 stores the result to global memory. In line 6, a[row*N +i] and a[col+i*N] are array accesses. The subscript uses the local variables row and col to calculate the memory location offset. The performance of this OpenCL program is low because it frequently accesses the high-latency global memory (the memory space storing array A). The bandwidth usage to transfer from off-chip DRAM to the chip is $2 \times N^3$, which is the performance bottleneck for this program.

2.5.1 An optimized Kernel

The implementation in the last example is not optimized because it takes up a great amount of bandwidth to transfer between DRAM and the GPU. The latency for each global-memory access is often as high as 300 to 500 GPU cycles. Moreover, the global-memory access of each column is strided, which causes banking conflicts because of restrictions in the hardware architecture, resulting in even longer access latency. In the Code 2.1, most of the kernel execution time is wasted waiting for the retrieval of data, rather than computing sums.

An analysis of the source code quickly reveals that Code 2.1 have redundant global-memory accesses. Although the size of the array stored in global memory is N^2, the kernel consumes $2 \times N^3$ global-memory bandwidth. A better strategy is to utilize the on-chip shared memory space because accessing shared memory is much cheaper than accessing global memory. The replacement of global-memory accesses with shared-memory accesses can greatly reduce both bandwidth consumption and access latency. In OpenCL, shared-memory space is a limited-sized cache that is software-managed by threads. Therefore the optimized kernel must include code that issues loads and stores to the shared memory.

Different threads may need to read the same element in the array. For instance, threads with the same value of either row or col have to access the same strip of row or col of the array. The bandwidth demand can be reduced by moving array elements into shared memory by tiles, and then letting the program have tiled access to the array in the shared memory. The bigger the tile is, the less the bandwidth demand will be.

It is not possible to move all the processing data to shared memory at once because shared memory is a scarce resource. The traditional solution is to divide the data into small sections, and to reuse the shared memory space by moving one small section of data to it at a time. This strategy requires the programmer to *stripmine* the innermost loop-i in order to allow the program to perform tiled array accesses.

Definition 1. *Loop stripmining is a compiler optimization that splits a single loop a into nested loops a' and a''. Assume that a' is the outer loop and a'' is the inner loop. The loop a'' only iterates over a smaller section of the loop a, and the loop a' executes a'' multiple times to make a'''s iterations cover all the iterations of a.*

Loop stripmining refers to the transformation of a single loop. In some cases when more

Code 2.2: OpenCL Matrix Multiplication

```
1  __kernel void multiply(__global float *a, __global float *c,
       int N){
2      int row = get_global_id(0);
3      int col = get_global_id(1);
4      float sum = 0.0f;
5      for (int tile_i = 0; tile_i < N; i+=tile_size){
6        for (int i = 0; i < tile_size; i++){
7          sum += a[row*N + (tile_i+i)]*a[col + (tile_i+i)*N];
8       }
9      }
10     c[row*N + col] = sum;
11 }
```

than one loop is involved, loop tiling is used to enable the program to have tiled array accesses.

Definition 2. *Loop tiling is a compiler optimization that breaks a loop's iteration space into smaller blocks (each block is also called a "tile"), and let the loop iterate over tiles of data.*

Loop stripmining and loop tiling are common compiler optimization techniques used to ensure that the block size fits in the cache line size, preventing cache-line spilling. In this example loop stripmining is used to divide the processing-data sections. Code 2.2 presents the kernel code with loop `i` stripmined.

From the view of thread groups, once the loop i is stripmined in the kernel, the array access is tiled. Assume the tile size is 10, Figure 2.7 shows the array accesses of a 10×10 rectangle thread group with `row` value ranging from 10 to 19 and `col` from 20 to 29. In the figure, grey areas are the memory sections that the threads in the thread group have to access in order to complete computing the sum. The access direction is also shown in the figure. Code 2.3 shows the kernel that have tiled access for shared memory utilization.

In Code 2.3, two shared (local) memory array spaces, `aTile1` and `aTile2`, are declared as parameters for intermediate fast storage space. In the kernel body, threads interleave the computation with shared-memory loading operations. The outer loop `tiled_i` holds the code that loads the global-memory tile to shared-memory space and the synchronization that maintains data consistency. The barrier statements immediately around the inner loop `i` (lines 17 and 21) ensure that all threads in a thread group finish computing their tile before the computaion of the next set of tiles starts. These two barriers guarantees the inner loop has access to the desired elements in shared memory space. With the data loading and barriers together, most global memory accesses are eliminated.

A thread loads two elements — one for `aTile1` and one for `aTile2` — to shared memory in each tile, but accesses $2 \times TILE\_SIZE$ elements in shared memory space, thus eliminating

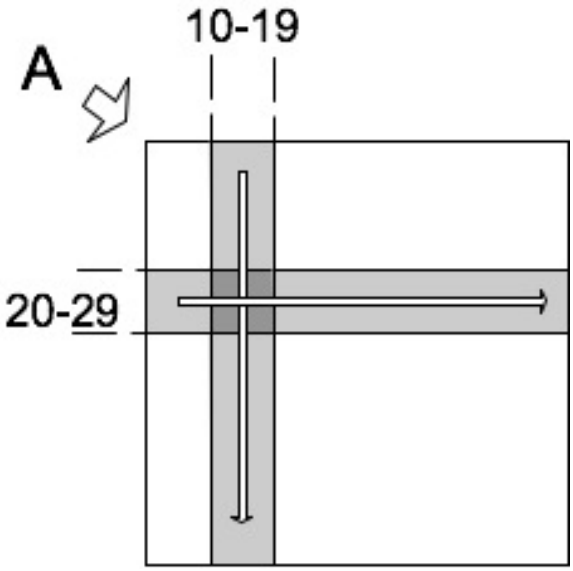

Figure 2.7: Tiled Memory Access Example

Code 2.3: OpenCL Matrix Multiplication

```
1  //TILE_SIZE is a constant
2  __kernel void multiply(__global float *a, _global float *c,
3                  int N,
4                  __local float aTile1[TILE_SIZE*TILE_SIZE],
5                  __local float aTile2[TILE_SIZE*TILE_SIZE]){
6   float sum = 0.0f;
7   int x = get_local_id(0);
8   int y = get_local_id(1);
9   int gidx = get_group_id(0);
10  int gidy = get_group_id(1);
11  int row = gidx*TILE_SIZE + x;
12  int col = gidy*TILE_SIZE + y;
13
14  for (int tiled_i = 0; tiled_i < N; tile_i+=TILE_SIZE){
15    aTile1[y*TILE_SIZE+x] = a[(gidy*TILE_SIZE*N+tiled_i)+y*N+x];
16    aTile2[y*TILE_SIZE+x] = a[(tiled_i*N+gidx*TILE_SIZE)+y*N+x];
17    barrier(LOCAL_MEM_FENCE);
18    for (int i = tiled_i; i < tiled_i +TILE_SIZE; i++){
19      sum += aTile1[y*TILE_SIZE+i] *aTile2[i*TILE_SIZE+x];
20   }
21   barrier(LOCAL_MEM_FENCE);
22  }
23  c[(gidy*TILE_SIZE+y)*N + gidx*TILE_SIZE+x] = sum;
24 }
```

global memory accesses. Assume that both the tile size and the thread group size are also $TILE\_SIZE \times TILE\_SIZE$ and the matrix size is $N \times N$. The total bandwidth consumption is only $\frac{1}{TILE\_SIZE}$ of the non-optimized kernel, which results in around 10x speedup in performance.

Another difference between Code 2.2 and Code 2.3 is that Code 2.3 uses thread local ID together with group ID to compute the array offset. The results computed by these two method are the same.

2.6 Conclusion Remarks

This chapter presents the idea of GPGPU heterogeneous computing, along with a short primer on GPGPU hardware and its programming model. GPU is a massive parallel computing equipment that has many cores on-chip with a restricted execution model. A GPU is most suitable for stream computing. For the sake of exploiting the GPU architecture, CUDA and OpenCL programming framework are designed to be the programming environment for GPGPUs. Common GPGPU applictions show strong computing ability compared to CPU applications; however they require programmers a paradigm shift to stream programming. To alleviate the burden of such paradigm shift is a main goal of our compiling framework.

Chapter 3

Background on Python, NumPy, and unPython

Chapter 2 introduces the nVidia GPGPU hardware and programming framework that are used in our analysis and compiling framework. This chapter continues to supply the readers with background knowledge of the compiling framework's source language and front-end framework. Section 3.1 discusses relevant features of the Python language; Section 3.2 shows one of Python's extension, NumPy, on which our framework is based; Finally the compiling framework's frontend and backend, unPython and jit4GPU are explained in Section 3.3 and Section 3.4 respectively.

3.1 The Python Language

Python is an object-oriented, full-fledged but flexible scripting language developed by a Dutch programmer Guido van Rossum in 1989. Ever since its creation, Python has always been a fast, light-weight, concise language with extreme flexibility, portability; it provides high productivity to the programmers. Because of these features it affords, after 20 years of development and establishment, Python is now widely used by millions of people in the fields of industry, scientific research and entertainment (game scripting).

There are several other benefits of using Python language:

- It's simple, with concise language syntax.
- It's easy to learn, easy to understand.
- It's free and open.
- It's high level.
- It supports multiple programming paradigms, including functional, imperative, object-oriented, reflective, etc.

- It's extensible and embeddable.
- It has strong library support.

3.1.1 Features

The most significant feature of Python is that it is a dynamic, scripting language. As a scripting language, Python is written in plain text format and is interpreted by Python virtual machine at runtime. Using a virtual machine, Python enables itself to provide such features as dynamic memory management, name resolution (late-binding) and dynamic typing.

Python has its own dynamic typing system. Typing system are used in every programming language to ensure the correct interpretation and memory allocation of variables. Static typing system enforces a variable to have a type and it performs type checking at compile time as many static languages do, while dynamic typing system enforces a value to have a type rather than enforcing a variable to have a type, which leads to postponing the type-checking until runtime. A dynamic typing system provides a more flexible programming environment than a static one, but it also makes the runtime check sophisticated and intepreter design restricted by some additional constraints. Many scripting languages cannot perform static typing check ahead of time due to the language features they offer, so dynamic typing system is dominant among those languages.

3.1.2 Performance Limitations

Although Python has brought so many exciting new features, the biggest limitation of this language is the performance. Because of its runtime interpretation and optimization, Python programs is often more than 10 times slower than C/C++ implementation that does the same job. This is a common problem in dynamic scripting languages, which hinders their expansions.

Although itself very suitable for GUI, text processing, controlling scripting, etc, pure Python programs are indeed not capable for computing-intensive core programs that requires high performance. However, developers of Python did not seek solutions to this by aggressively optimizing Python running speed overnight(which was, of course, impossible at the moment), but to provide internal APIs to Python programmers in order to allow them to have internal control over Python objects. By implementing custom Python objects in other languages like C by hand, programmers not only can have grasp on those functions Python does not ship with originally, but also achieve better performance because customized Python modules that are written using those APIs will be compiled statically and do not require runtime interpretation.

3.1.3 Python/C API

One of the reason Python is popular is that it is extremely extensible. Python gives programmers the privileges of using other programming languages in working with Python modules for Python so that programmers can have implementations of extension modules in languages other than Python (extending Python), and use Python-written modules as components in a larger application implemented in other languages (embedding Python).

Python/C API is Python's gateway for C/C++ programmers for extending Python. It allows C/C++ programmers to create Python modules in C/C++. The reasons to utilize Python/C APIs may include:

- Calling functions in existing libraries written for C/C++.
- Optimizing cores for performance.
- Adding new built-in types for Python.

Python/C API enables Python developers to do what exceeds Python's capability, like creating distributed computing platforms not supported natively. For details please refer to official Python/C API documentation [1].

3.2 Scientific Computing Tools for Python – NumPy

Many popular Python extensions are created using Python/C API; one of the important product essential to our work is the NumPy library module. NumPy is a Python extension module created specifically for granting Python the scientific computing ability [20]. NumPy comes as a bundle that includes:

- a powerful N-dimensional array object which is not supported natively by Python
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and FORTRAN code
- useful linear algebra, Fourier transform, and random number capabilities.
- additional NumPy C APIs that allow programmers to write high performance C extensions.

In the following context we will focus on the array object, as it is connected to the "unPython+jit4GPU" framework.

3.2.1 The Array Object

Functions

The core of NumPy is the flexible multi-dimensional `array` object implemented using Python/C API. a NumPy `array` object is an object that represents a multi-dimensional array in the memory, it comes with its member methods for setting or getting the objects attributes and values, like initialization, sub-matrix slicing, dimension change, etc.

Code 3.1: Example of NumPy array

```
1 from numpy import array
2 a = array([1,2,3,4,5,6])
3 print a
4 b = a.reshape(3,2)
5 print b
```

Code 3.1 and Code 3.2 together illustrates the programmer's view of NumPy. Code 3.1 shows an example of manipulating a NumPy array object, and Code 3.2 shows the output of the example in terminal. In Code 3.1, line 2 creates an array object from a list, which contains 6 elements. Line 1 in Code 3.2 shows the structure of the object pointed to by `a`, it is a 1-dimensional array with 6 elements. Line 4 in Code 3.1 assigns a "reshaped" view of the same object to variable `b` and , line 5 prints it. The printing result is shown in Code 3.2, which is a 2-dimensional array consists of the same values.

Code 3.2: Output of Code 3.1

```
1 >>>[1 2 3 4 5 6]
2 >>>[[1 2]
3     [3 4]
4     [5 6]]
```

Object Structure

An array object has a member variable pointer pointing to an allocated raw array buffer in the memory pool, and a list of strides that defines the the layout of the array. Given a variable a refers to an n dimensional object, a set of indices of each dimension (the i^{th} dimension's index denoted as d_i), the memory pointer ptr that points to the raw array buffer, and the list of strides s, then the memory location of $\texttt{a}[d_0,d_1,\ldots,d_{n-1}]$, denoted as m, can be calculated as:

$$m = ptr + \sum_{i=0}^{n-1} s[i] \times d_i$$

Given the same ptr, same d_i where $i = 0, \ldots, n-1$, but with different strides s, the shape of the array is different. The line 4 of Code 3.1 creates another array object from the object `a` points to. `b` and `a` share a same raw array buffer area but with different lists of

strides. Array shape manipulation operations are all based on the manipulation of strides. This is different from other array types in C or C++. In the view of a C programmer, the strides in multi-dimensional addressing is statically defined.

NumPy array object also provides public access to its raw data information for programmers to take full control or build extensions for NumPy [21]. This is essential to the analysis in unPython. In the following section we are going to take a look at unPython and its analysis.

3.3 unPython – unwrapping python to C

unPython is a Python library compiling framework created by Rahul Garg [11], which unwraps Python programs into C programs. Newer version of unPython translates an annotated Python program into an OpenMP program and an AMD CAL program(which requires the runtime support from jit4GPU). The translated program can be executed on CPUs (using generated OpenMP code) or on AMD GPUs (using generated AMD CAL code), depending on the machine hardware availability. The choosing of execution hardware is one of the jobs of jit4GPU, which will be introduced later.

Language Model Extension

The language model of unPython is an extension to the conventional Python language model. In unPython, annotations that describe parameter type information, and new object types that serve as compiler directives for declaring parallel loops are introduced in the language model. The extension is designed in such a ways that it is still compliant with Python interpreter (adding a line starting with `@` immediately before the definition of a function). To be specific, decorator `type` declares the parameter types in its own parameter list, an iteration over a `gpurange` is declared to be a parallel iteration. The extended Python program can be compiled successfully by a Python interpreter without altering the program. However, in the view of unPython compiler, the injected compiler directives (annotations and new types) provide additional information and instruction for generating parallel code. Code 3.3 shows a function that is written in unPython format.

Code 3.3: Example of unPython Program Extension

```
1 from unpython import gpurange
2 @type('ndarray[int32 1]','ndarray[int32 1]' ,'ndarray[int32 1]'
    ,'int32','int32')
3 def f(A,B,C,n):
4     sum = 0
5     for i in gpurange(N):
6         C[i] = A[i] + B[i]
7         sum += C[i]
8     return sum
```

The code states that the first three parameters from the left, `A`, `B` and `C` are 1 dimensional arrays with element data type `int32`; the last parameter is a variable of `int32`; the return type is also `int32` (indicated by the last element in the annotation list). In the function there is a parallel iteration that calculates array `C` and sums up its elements.

Benefits and Constraints

Standard Python language does not ship with support for parallel programming. unPython introduces parallelism that works as an alternative way of developing Python programs. What is more, the extended programming model is fully compatible with standard Python interpreters. Programs written for unPython can execute as a standard Python program without modification, this would benefit programmers in debugging programs and porting to new software and hardware environments.

However, unPython is not omni-potent. Due to the dynamic typing system in Python, unPython requires additional parameter typing information declared in annotation to generate efficient target code. Also, at present unPython only compiles a subset of Python program features for efficient compiling. unPython cannot support features like runtime code execution, higher order functions, generators, meta-classes or special methods [11].

3.4 jit4GPU – Just In Time Compiling for GPU

Jit4GPU is the just-in-time support component of Garg's unPython+jit4GPU Python compiling framework. The main job of jit4GPU is to dynamically translate the program into AMD CAL code, and to deal with data allocation, data transferring.

Figure 3.1 shows jit4GPU GPU execution workflow, what does not show in the figure is that whenever jit4GPU encounters an program pattern it cannot handle, it will exit immediately to fallback to CPU execution.

Jit4GPU accepts as parameters the program abstract syntax tree (AST) in stream format, variable type information and array access information. Jit4GPU first rebuild a new AST from the stream, then it does the following jobs:

Computing the Memory Region for Data Transfer

Jit4GPU computes the processing array element memory regions from array access information (the discussion of array access information is in Chapter 4). For each array access reference in the program, it calculates a rectangular memory region that covers all its memory locations that the reference points to. If two or more rectangular regions overlap, a bigger rectangular region is calculated to cover them all. The computing result is a set of disjoint rectangular memory regions, which will be transferred to device memory.

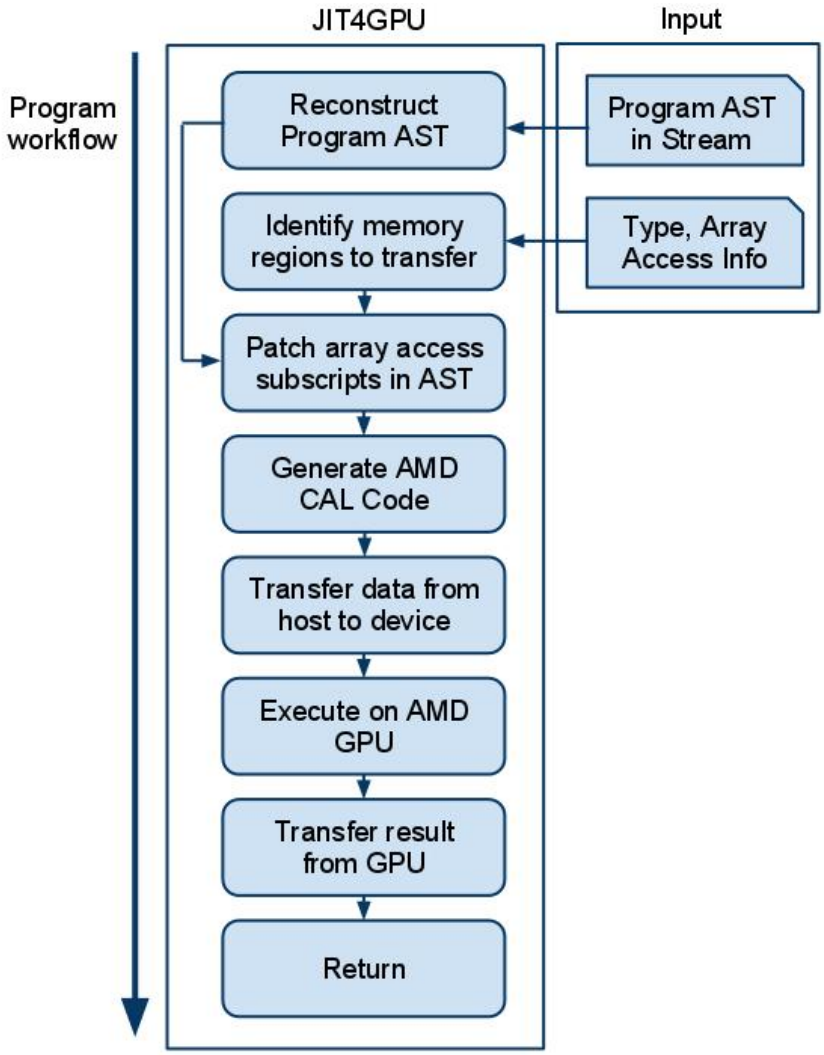

Figure 3.1: jit4GPU Workflow

Transferring data between host memory and device memory costs time due to the relatively small bandwidth between the two. Transferring rectangular regions rather than the whole bunch of memory locations helps to reduce the required bandwidth.

Code Patch, GPU Code Generation, Device Memory Allocation and Data Transfer

The array access references in the program must be altered due to the change of memory place and layout. Before actually generating GPU code, jit4GPU patches all the array access references on the AST tree, re-directing array header pointers to new place, and changing array access subscripts.

Besides code generation, jit4GPU also handles device memory operations and data transfer. Jit4GPU does the following jobs:

1. generate AMD CAL code by traversing the patched AST tree
2. allocate device memory objects, initialize with size
3. before GPU code execution, copy rectangular memory regions of processing arrays to device memory space
4. handle GPU execution
5. after GPU code execution, copy results from device memory space back to host memory space

After fetching results from device memory, jit4GPU exits and hands in the execution to its caller function.

3.5 Chapter Conclusion

As a swift and flexible programming language, Python is a very popular widely used language. With the extension module NumPy, Python now supports various scientific operations and functions. NumPy also provides its own C APIs to allow programmers to build further customized extensions. These APIs provide a good opportunity for those who wants to expand Python's computing ability by directly taking control over NumPy objects.

To extend the Python language to allow execution on parallel machines such as heterogeneous computing platforms consisting of CPUs and GPUs, the unPython+jit4GPU framework translates annotated Python program with NumPy array objects as main data structure for computing into CPU and AMD GPU code. Our jit4OpenCL framework aims at replacing jit4GPU in the unPython+jit4GPU framework in order to expand its portability.

Chapter 4

Research Problem Formulation

One of the great difference between CPU and GPU programming is the control of their memory hierarchy. NVidia GPUs have a software-managed cache space called shared-memory space (discussed in Chapter 2) that is used to reduce data-movement requirements between off-chip DRAM space and the GPU itself. Automatically generating code, by a compiler, which can utilize the additional memory space is the key for performance optimization. This chapter introduces the array-access analysis based on RCSLMAD used in jit4OpenCL. This analysis allows jit4OpenCL to exploit the potential data reuse for array accesses. Specifically, Section 4.1 gives an overview of the goal of the analysis; Section 4.2 presents a problem statement that the analysis must solve; Section 4.3 formalizes the problem solution; Section 4.4 concludes this chapter.

4.1 The Goal

The goal of this research is to let one of the existing array access analysis methods, Linear Memory Address Descriptors (LMADs), which is deployed in unPython and jit4GPU , be applied on jit4OpenCL compiler for successful OpenCL code analysis, optimization and code generation.This chapter describes the problem statement and the modifications to this analysis that were necessary for our own purposes.

Jit4OpenCL has to restructure loops, including loop strip mining, loop domain dividing, *etc*, in order to make the data fit into the limited memory resources in the GPUs. Therefore the analysis must be able to identify memory locations that are accessed within a parallel loop nest as well as the memory locations accessed by any single iteration of the loop. Most importantly, the analysis must be suitable for stream processing that can be used to generate SIMD-style programs.

4.2 Problem Statement

Given a set of array access references S in a parallel loop nest P: use S to identify the tiled memory accesses in P', which is a transformation of P. The transformation may include parallelization, strip mining and tiling.

4.3 Refined Array Access Analysis

This section first starts by introducing the LMAD-based analysis, and then discusses the adaptation of this analysis to jit4OpenCL.

4.3.1 Identifying Array Access Elements

In array access analysis, a memory location (noted as l) indicates a position in the memory space. Let M be the set of memory locations referenced by an array access throughout the program's execution.

Given a loop nest of depth d, the memory references within the loop nest may be represented by a function f, whose input is a d-component vector $\vec{i} = (i_1, i_2, i_3, ..., i_d)$, where i_k is the k^{th} loop index in the loop nest. The set of legal loop indices $\vec{i}$ forms the d-dimensional iteration domain D. D is represented in the following format: $D = (l_1..u_1, l_2..u_2, \ldots, l_d..u_d)$, where for any $0 < k \leq d$, l_k and u_k are the lower and upper bounds of loop k, respectively, and $l_k \leq u_k$. For normalized loops, the lower bound of each loop counter is 0. We further constrain D by requiring the upper bound to be an affine function of the outer loop indices. Such constraint does limit the analysis on applications that has certain memory accessing patterns, but a large portion of scientific applications can be implemented with such constraints.

Let $f : \vec{i} \rightarrow \mathbb{N}$ be the following affine function:

$$f(\vec{i}) = b + \sum_{k=1}^{d} s_k \times i_k | \vec{i} \in D$$

where $s_k | k = 1, 2, ..., d$ are the increment strides of each loop counter, b is the base of the array access descriptor. The function $f(\vec{i})$ is an array descriptor called a Constraint-Stride LMAD (CSLMAD) [11]. Assume that the strides are sorted in decreasing order. Let s_{r_k} be the k^{th} stride in this sorted list, u_{r_k} be the upper bound, and r_k be the position in the sorted list of the loop with stride s_{r_k}. Let $s_{r_m} = \min(s_{r_k}) | s_{r_k} \neq 0$.

A CSLMAD is a Restricted Constant-Stride LMAD (RCSLMAD) if and only if for every s_{r_k} we have:

$$s_{r_k} \geq s_{r_m} - 1 + \sum_{j=k+1}^{m} u_{r_j} \times s_{r_j}$$

This restriction guarantees that in a RCSLMAD the references with small strides do not overlap with references with larger strides.

A function $f(\vec{i})$ and a loop-iteration domain D define a set of memory locations $L_f(D) = \bigcup_{\vec{j} \in D} f(\vec{j})$. The memory space between $min(f(\vec{j}))$ and $max(f(\vec{j}))$ for all $\vec{j} \in D$ is the *region* defined by f on D, and is denoted as $|D|_f$. Not all locations within this region are necessarily referenced by the loop. The locations that are referenced are called the *effective memory location*s in this analysis. Figure 4.1 shows an example of a region. Grey blocks are effective memory locations while white blocks are not.

Theorem 1. *Given $\vec{i} \in D$ and $\vec{j} \in D$, such that $f(\vec{i})$ and $f(\vec{j})$ are RCSLMADs. If $\vec{i} \neq \vec{j}$ then $f(\vec{i}) \neq f(\vec{j})$.*

Proof. For the sake of contradiction, assume that there are two vectors $\vec{i}$ and $\vec{j}$ in an iteration domain D such that $\vec{i} \neq \vec{j}$ and $f(\vec{i}) = f(\vec{j})$. Therefore:

$$f(\vec{i}) - f(\vec{j}) = 0 \tag{4.1}$$

$$\sum_{k=1}^{d} i_k \times s_k - \sum_{k=1}^{d} j_k \times s_k = 0 \tag{4.2}$$

$$\sum_{k=1}^{d} (i_k - j_k) \times s_k = 0 \tag{4.3}$$

$$\sum_{k=1, i_k \neq j_k}^{d} (i_k - j_k) \times s_k = 0 \tag{4.4}$$

In an RCSLMAD, strides are sorted in decreasing order. Let p be the dimension for which $s_p \geq s_k$ for all k such that $i_k \neq j_k$, and assume that $i_p > j_p$. Equation (4.4) can be rewritten as:

$$(i_p - j_p) \times s_p + \sum_{k>p, i_k \neq j_k}^{d} (i_k - j_k) \times s_k = 0 \tag{4.5}$$

To prove that Equation (4.5) cannot be true, we refer to the definition of a RCSLMAD. In the definition, the strides satisfy the following condition:

$$s_p \geq s_d - 1 + \sum_{j>p}^{d} u_j \times s_j \tag{4.6}$$

Thus, replacing s_p in the left hand side of Equation (4.5) with (4.6) will yield:

$$\begin{aligned} &(i_p - j_p) \times s_p + \sum_{k>p, i_k \neq j_k}^{d} (i_k - j_k) \times s_k \geq \\ &(i_p - j_p) \times (s_d - 1 + \sum_{k>p}^{d} u_k \times s_k) + \sum_{k>p, i_k \neq j_k}^{d} (i_k - j_k) \times s_k \end{aligned} \tag{4.7}$$

$i_p - j_p \geq 1$ because $i_p \neq j_p$, thus:

$$\begin{aligned}
&(i_p - j_p) \times (s_d - 1 + \sum_{k>p}^{d} u_k \times s_k) + \sum_{k>p, i_k \neq j_k}^{d} (i_k - j_k) \times s_k \\
&\geq (s_d - 1 + \sum_{k>p}^{d} u_k \times s_k) + \sum_{k>p, i_k \neq j_k}^{d} (i_k - j_k) \times s_k \\
&\geq (s_d - 1) + \sum_{k>p, i_k \neq j_k}^{d} (u_k + i_k - j_k) \times s_k \qquad (4.8)
\end{aligned}$$

Now we will show that the sum in expression 4.8 must be strictly positive. For any values of i_k and j_k, $(i_k - j_k) \geq -|i_k - j_k|$. Therefore:

$$\sum_{k>p, i_k \neq j_k}^{d} (u_k + i_k - j_k) \times s_k \geq \sum_{k>p, i_k \neq j_k}^{d} (u_k - |i_k - j_k|) \times s_k \qquad (4.9)$$

For i_k, j_k we have that : $0 \leq i_k < u_k$, $0 \leq j_k < u_k$. The difference between two numbers that are strictly smaller than u_k must be less than or equal to $(u_k - 1)$, therefore:

$$\begin{aligned}
|i_k - j_k| &\leq (u_k - 1) \\
-|i_k - j_k| &\geq (1 - u_k) \qquad (4.10)
\end{aligned}$$

Combining expressions 4.10 and 4.9 results in:

$$\begin{aligned}
\sum_{k>p, i_k \neq j_k}^{d} (u_k + i_k - j_k) \times s_k &\geq \sum_{k>p, i_k \neq j_k}^{d} (u_k + (1 - u_k)) \times s_k \\
\sum_{k>p, i_k \neq j_k}^{d} (u_k + i_k - j_k) \times s_k &\geq \sum_{k>p, i_k \neq j_k}^{d} s_k
\end{aligned}$$

s_k is always positive, therefore we can conclude that:

$$\sum_{k>p, i_k \neq j_k}^{d} (u_k + i_k - j_k) \times s_k > 0 \qquad (4.11)$$

Using expression 4.11 we conclude that the left-hand side of equation 4.5 is strictly greater than 0. That is, under the assumption that $i_p > j_p$, $f(\vec{i}) - f(\vec{j})$ cannot be 0, which means that there cannot exist two different vectors $\vec{i}$, $\vec{j}$ in domain D such that $f(\vec{i}) - f(\vec{j}) = 0$.

Thus from the contradiction it is proven that $f(\vec{i}), i \in D$ is a one to one mapping. □

Corollary 1. *Given $D_1 \subseteq D$ and $D_2 \subseteq D$ such that $D_1 \cup D_2 = D$, $D_1 \cap D_2 = \emptyset$ then $L_f(D_1) \cup L_f(D_2) = L_f(D)$, $L_f(D_1) \cap L_f(D_2) = \emptyset$.*

Proof. $L_f(D_1) \cup L_f(D_2) = L_f(D)$ is a straightfoward deduction from the fact that:

$$\left(\bigcup_{a \in D_1} \{f(a)\} \right) \bigcup \left(\bigcup_{b \in D_2} \{f(b)\} \right) = \bigcup_{c \in D_1 \cup D_2} \{f(c)\}$$

To prove that $L_f(D_1) \cap L_f(D_2) = \emptyset$, we use the fact that $\vec{i} \rightarrow f(\vec{i})$ is a one-to-one mapping from Theorem 1. If we have $\forall \vec{t} \notin D$ then $f(\vec{t}) \notin \bigcup_{\vec{p} \in D} f(\vec{p})$. Let $\vec{j} \in D_1$ and $\vec{i} \in D_2$ where $D_1 \cap D_2 = \emptyset$. Then $f(\vec{j}) \notin \bigcup_{\vec{p} \in D_2} f(\vec{p})$ and $f(\vec{i}) \notin \bigcup_{\vec{p} \in D_1} f(\vec{p})$. Therefore $\bigcup_{j \in D_1} f(\vec{j}) \bigcap \bigcup_{i \in D_2} f(\vec{i}) = \emptyset$, that is $L_f(D_1) \cap L_f(D_2) = \emptyset$. □

There may be unnecessary elements (non-effective memory locations) interleaved with effective memory locations, creating "holes" in the memory region. It is a waste to leave these holes in shared memory because transferring data between different levels of device memory is expensive. Consider the example shown in Figure 4.1. The figure represents array access of the following RCSLMAD (assuming that the starting address is 0):

$$f((i,j)) = 2 \times i + 12 \times j$$

$$0 \leq i < 3 \text{ and } 0 \leq j < 3$$

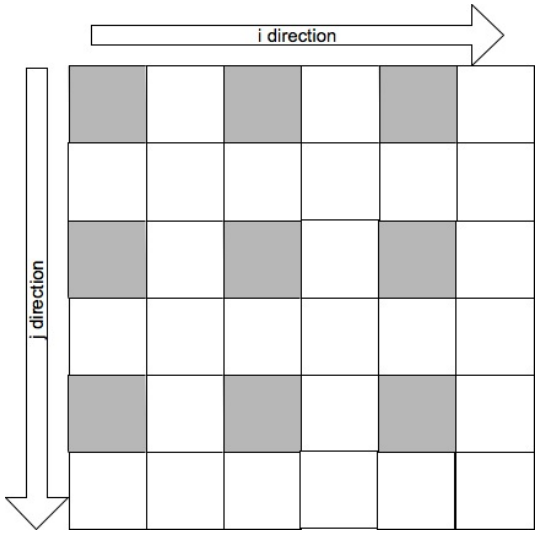

Figure 4.1: Example of legal RCSLMAD

Although the region of the RCSLMAD covers a large area, there are only 9 effective memory locations. In some cases transferring the whole RCSLMAD region to the device shared memory would be extremely expensive because the effective memory locations only occupy a small portion of the region.

The solution to this is to compress the RCSLMAD to squeeze the holes out. If $f(\vec{i}), \forall \vec{i} \in D$ is a RCSLMAD, where the j^{th} dimension's lower bound is 0, upper bound is u_j, then we can construct another RCSLMAD function $g(\vec{i})$ such that its strides are $s_{r_d} = 1, s_{r_k} = \sum_{j=k+1}^{d} u_{r_j} \times s_{r_j}$, where $r_l, l = 1, 2, ..., d$ is the dimension index sorted in stride's decreasing order. This transformation of RCSLMADs eliminates the access holes from the region defined by g on D, $|D|_g \leq |D|_f$ and $|L_g(D)| = \prod_{i=1, u_i \neq 0}^{d} u_i$. Most importantly, the memory

space occupied by the effective memory locations in $f(\vec{i})$ is $|D|_g$.

Theorem 2. *All memory locations in the region of D on g (that is, the memory space spreading from $max(L_g(D))$ to $min(L_g(D))$) are effective memory locations.*

Proof. It is obvious that:

1. $g(\vec{i}), \vec{i} \in D$ is a RCSLMAD;
2. there are $|L_g(D)| = \prod_{i=1, u_i \neq 0}^{d} u_i$ effective elements in $f(\vec{i})$, $g(\vec{i})$, and $|D|_g$, which means that every memory location in $g(\vec{i})$ is an effective memory location.

□

For instance, consider the example shown in Figure 4.1. The constructed RCSLMAD function $g(\vec{i})$ is: $g((i,j)) = 1 \times i + u_1 \times j = i + 3j$, and $|D|_g = 9$, resulting in no waste of shared memory.

In a compiler implementation, LMADs are transferred to shared memory and are compressed to reduce the usage of scarce shared memory using Theorem 2. In other words, given a RCSLMAD $f(\vec{i})$ representing processing data elements in device global memory, our compiler creates a new RCSLMAD $g(\vec{i})$ in device shared memory, and maps every legal $f(\vec{i})$ to its destination $g(\vec{i})$ before computing. After the computation $f(\vec{i})$ is updated if $g(\vec{i})$ has any change.

4.3.2 Program Transformation and RCSLMAD Decomposition

In some real-world scientific computing applications the computer needs to crunch large amount of input data. Due to the size restriction of the on-chip shared-memory architecture used in present nVidia GPGPU, programmers cannot process big RCSLMADs on device shared memory for data reuse. To generate efficient code, the compiler must decompose the RCSLMAD into smaller RCSLMADs, so that:

1. each of the smaller RCSLMADs fits to shared memory size;
2. any of the stream processor in GPU can process one decomposed RCSLMAD at a time;
3. the collective result of the access to the decomposed RCSLMADs is exactly the same as accessing RCSLMADs that are not decomposed.

In this compiler implementation, the domain for RCSLMAD is manipulated for decomposing. Given a RCSLMAD $f(\vec{i}), \vec{i} \in D$, split the domain D into disjoint sub domains D_1, $D_2 \ldots D_n$ such that $\forall i, j, D_i \cap D_j = \emptyset$, $D_1 \cup D_2 \cup \ldots \cup D_3 = D$, thus n smaller RCSLMADs are

formed. These RCSLMADs are: $f(\vec{i}), \vec{i} \in D_1$, $f(\vec{j}), \vec{j} \in D_2$, ..., $f(\vec{k}), \vec{k} \in D_n$. According to Corollary 1, $\forall i, j$ we have $L_f(D_i) \cap L_f(D_j) = \emptyset$, and $L_f(D_1) \cup L_f(D_2) \cup \ldots \cup L_f(D_n) = L_f(D)$.

D is split in such a way that every dimension of D is divided into contiguous regions. This way D is decomposed from a large cube into many smaller cubes. Each small cube represents the region of a sub domain. In this thesis this process is called RCSLMAD decomposition. RCSLMAD decomposition requires transformations to the program source code: for a parallel loop nest that contains sequential loops with array accesses in the body, the compiler makes the array accesses tiled by strip-mining the sequential loops and dividing parallel loop domains. These transformations will split D into many sub-domains. By iterating over a tiled array access, the corresponding iteration sub-domain is traversed. Figure 4.2 is an example showing a 3-dimensional domain ($100 \times 100 \times 100$) decomposed into 8 smaller contiguous sub-domains, each sub-domain consists of $\frac{1}{8}$ of the bigger domain's space (the size is $50 \times 50 \times 50$). This is done by splitting each dimension in half.

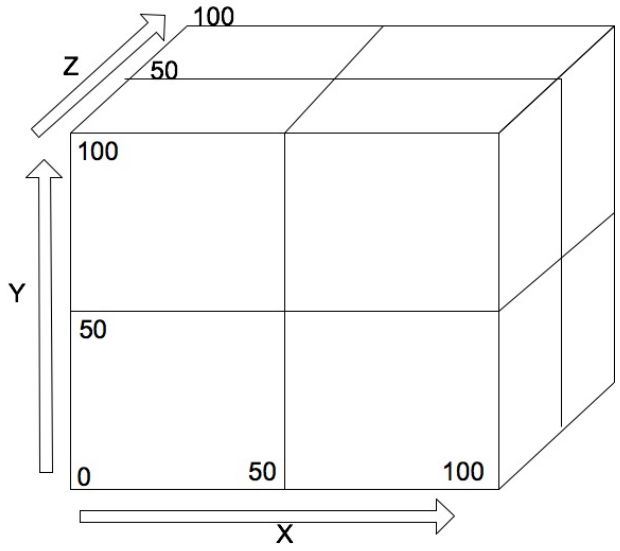

Figure 4.2: Domain Decomposition Example

4.3.3 Ordering

Sorting all the elements in a RCSLMAD enables the compiler to generate shared-memory transfer code through memory-address mapping. Threads have their own, unique IDs, the compiler must know which elements in a RCSLMAD should a thread load to shared memory. Creating an order among those memory locations pointed by a RCSLMAD will also create a correspondence relationship between threads and the elements of RCSLMADs.

Ordering Elements for Different Memory Layer Mapping

Elements in $f(\vec{i}), \vec{i} \in D$ can be ordered because a RCSLMAD is a one-to-one mapping from vector value to scalar value. To calculate this order number, we have to reuse some concepts defined before: in previous context, we used the stride list sorted in decreasing order, $S_{r_j}|j = 1, 2, \ldots, d$ to explain the idea of RCSLMAD, where the subscript $r_j|j = 1, 2, \ldots, d$ is the the domain index of the j^{th} stride in this sorted list, u_{r_j} and l_{r_j} are the upper bound and lower bound corresponding to S_{r_j} respectively, and the r_mth stride represents the smallest non-zero stride in the list. The order number of element $f(i_1, i_2, \ldots, i_d)$ in this thesis is defined as:

$$Order(\vec{i}) = \sum_{j=1}^{r_m-1} \left((i_{r_j} - l_{r_j}) \left[\prod_{k=j+1}^{r_m} (u_{r_k} - l_{r_k}) \right] \right) + (i_{r_m} - l_{r_m}) \tag{4.12}$$

Expression (4.12) constructs a unique order ID of an element using those non-zero strides. In our analysis each RCSLMAD is normalized, so $l_{r_i} = 0, i = 1, 2, \ldots, d$, (4.12) can be simplified to:

$$Order(\vec{i}) = \sum_{j=1}^{r_m-1} \left(i_{r_j} \left[\prod_{k=j+1}^{r_m} u_{r_k} \right] \right) + i_{r_m} \tag{4.13}$$

Conversely, given $Order(\vec{i})$, and D, we can calculate $\vec{i}$. Assume that we have the value of $Order(\vec{i})$, that the bounds of D are $(l_1 \ldots u_1, l_2 \ldots u_2, \ldots, l_d \ldots u_d)$, and that the strides are $(s_1, s_2, \ldots, s_d)$. Let s_{r_k} be the k^{th} largest stride, then $\vec{i} = (i_1, i_2, \ldots, i_d)$ can be found as described below. The first non-zero index, i_{r_m} is given by:

$$i_{r_m} = \begin{cases} Order(\vec{i}) & \text{if } m = 1 \\ Order(\vec{i}) \pmod{u_{r_m} - l_{r_m}} & \text{if } m \neq 1 \end{cases} \tag{4.14}$$

the remainder indices $i_{r_{m-1}}, i_{r_{m-2}, \ldots, i_1}$ can be calculated as follows:

$$i_j = \left\lfloor \frac{Order(\vec{i})}{\prod_{b=j+1}^{r_m} (u_b - l_b)} \right\rfloor \pmod{u_j - l_j} \tag{4.15}$$

where $j = r_m - 1, r_m - 2, \ldots, 1$.

For normalized loops all lower bounds (l_b and l_j in the expression above) are equal to zero. In this case, a simplified version for normalized RCSLMADs is:

$$i_j = \left\lfloor \frac{Order(\vec{i})}{\prod_{b=j+1}^{r_m} u_b} \right\rfloor \pmod{u_j} \tag{4.16}$$

where $j = r_m - 1, r_m - 2, \ldots, 1$.

We use an example to show how it works: assume there is a RCSLMAD, $f(\vec{i}) = 24 \times i_1 + 4 \times i_2 + 1 \times i_3, 0 \leq i_1 < 5, 0 \leq i_2 < 6, 0 \leq i_3 < 4$, we need to compute the order ID of the element at $(i_1, i_2, i_3) = (1, 2, 3)$, it will be:

$$Order(\vec{i}) = 1 \times (6 \times 4) + 2 \times 4 + 3 = 24 + 8 + 3 = 35$$

And to compute the $\vec{i}$ when $Order(\vec{i}) = 35$:

$$i_3 = 35 \pmod 4 = 3$$

$$i_2 = \lfloor \frac{35}{4} \rfloor \pmod 6 = 2$$

$$i_1 = \lfloor \frac{35}{6 \times 4} \rfloor \pmod 5 = 1$$

Ordering Threads

Within a thread group, every thread has its own order. This order is defined as follows: for an n-dimensional thread group, with dimension sizes of $l_i, i = 1, 2, \ldots, n$, then the thread ID in the group is:

$$ID_{thread} = \sum_{i=1}^{n} \left(d_i \times \prod_{j=i-1}^{i-1} l_j \right) \tag{4.17}$$

Where d_i is the i^{th} component of the thread local ID. The range of thread ID extends from 0 to the size of group minus one.

Assigning Shared Memory Loading/Storing Order

Each thread is responsible for loading and storing RCSLMAD elements to shared-memory space. Each i^{th} thread can load/store the i^{th} element because a sequence has already been assigned to the threads in a thread group, and to the elements in the RCSLMAD.

However, this solution may cause hazards to the program. Section 5.1 describes the potential threats and the compromises the compiler makes in order to maintain the correctness and consistency of the program.

4.3.4 RCSLMAD Decomposing Example

Consider the following RCSLMAD:

$$f(i, j, k) = 64 \times i + 8 \times j + 2 \times k$$
$$0 \leq i < 4, 0 \leq j < 8, 0 \leq k < 4 \tag{4.18}$$

The RCSLMAD described by equation 4.18 has three parameters — i, j, and k — whose strides are 64, 8 and 2, respectively. Assuming that the array is stored in row-major order,

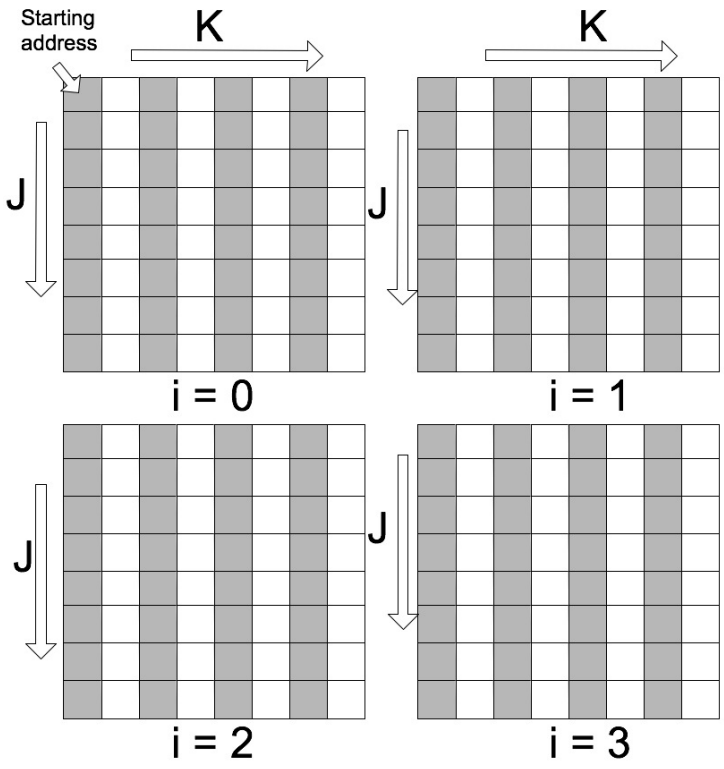

Figure 4.3: 3-Dimensional LMAD example

Figure 4.3 explicitly represents the memory space as a cube. White blocks represent memory locations that are not accessed while grey ones are those that the RCSLMAD (4.18) accesses. Consider the case in which this LMAD must be decomposed into 8 smaller adjacent but disjoint RCSLMADs so that each can be processed with smaller memory space requirement. The 8 smaller RCSLMADs can be obtained by splitting dimensions i, j and k in half as indicated by the black broken lines Figure 4.4.

Now assume that the compiler needs the RCSLMAD representation of the memory locations in Group 7. The RCSLMAD function $f(i, j, k)$, can be adapted to this new iteration domain. The i, j and k dimensions are split in half. The i dimension of group 7 extends from 2 to 3, the j dimension extends from 4 to 7, and the k dimension from 0 to 1. Therefore group 7 can be represented by:

$$\begin{aligned} f(i,j,k) &= 64 \times i + 8 \times j + 2 \times k \\ 2 \leq i < 4, &5 \leq j < 8, 0 \leq k < 2 \end{aligned} \tag{4.19}$$

This RCSLMAD can be normalized to make every dimension index variable starts from

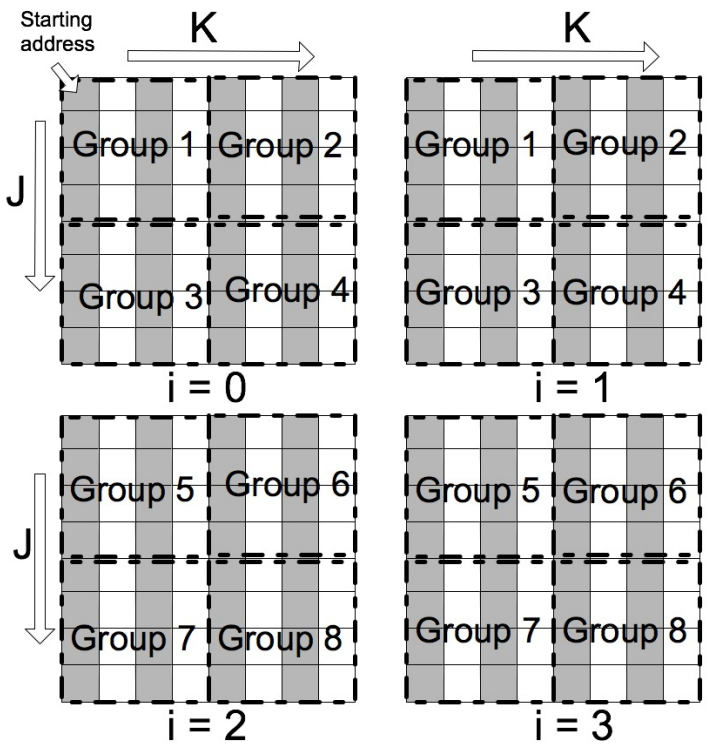

Figure 4.4: Decomposing LMAD into small Groups

0. The normalization requires the following transformation to Equation 4.19:

$$f(i,j,k) = 64 \times (2+i) + 8 \times (4+j) + 2 \times k$$
$$0 \le i < 2, 0 \le j < 4, 0 \le k < 2 \tag{4.20}$$
$$\rightarrow \quad f(i,j,k) = 160 + 64 \times i + 8 \times j + 2 \times k$$
$$0 \le i < 2, 0 \le j < 4, 0 \le k < 2 \tag{4.21}$$

Equation 4.21 is the normalized RCSLMAD for Group 7, the other three RCSLMADs can be obtained by exactly the same analysis. The analysis is most suitable for stream-processing programs.

4.4 Discussion on LMAD-based Analysis for GPGPU

LMAD is a simple and straightforward representation for memory accesses that has been developed to assist in the formulation of compiler memory behavior analysis. This chapter presented an adaptation of the LMAD analysis to decompose loops in a way that is especially suitable for stream processing. Given an RCSLMAD function and its parameters, the compiler can identify on the fly the memory locations that each stream processor should process. This analysis enables automatic parallel-program generation for stream proces-

sors such as GPGPUs because the analysis incurs in only a small amount of performance overhead.

Chapter 5

Generation of OpenCL code from Python

This chapter describes the generation of OpenCL code from the code generated by unPython for programs written in Python/NumPy. The approach adopted in this thesis is to reuse as much of the infrastructure developed by Rahul Garg as possible. The jit4OpenCL analysis flowchart is shown in Figure 5.1. At a high level, the modifications listed below were necessary to transform jit4GPU from a compiler that generates AMD CAL code to a compiler that generates OpenCL code.

- Remove AMD CAL program generation.
- Insert OpenCL runtime support and OpenCL kernel generation routine.
- Insert a grid-configuration analysis that is specific for OpenCL.

5.1 Dealing with Overlapping LMADs

The memory regions referenced by multiple LMADs may overlap. In Rahul Garg's implementation, several LMADs are grouped if they overlap, and the compiler schedules group transfers for each LMAD group.

That analysis is not working perfectly with LMADs whose regions overlap. Consider the following situation: assume there are two LMADs $f(\vec{i})|\vec{i} \in D_1$ and $g(\vec{j})|\vec{j} \in D_2$, that we have $\vec{i'} \in D_1$, $\vec{j'} \in D_2$ where $f(\vec{i'}) = g(\vec{j'})$. If the program writes to both LMADs in two different threads and the two LMADs are loaded to shared memory at different times, the program may result in a Write-After-Write (WAW) hazard because the same element is written twice but the two writes are not ordered.

To avoid this problem and preserve data consistency, jit4OpenCL makes the following decisions regarding the write and read status of the array references if there is an overlap between two or more LMADs.

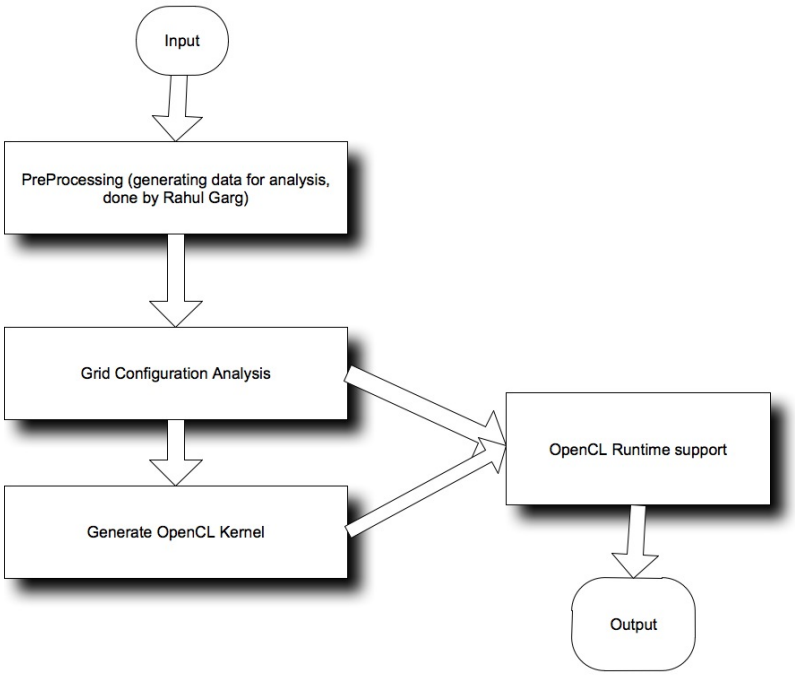

Figure 5.1: Modified jit4GPU Analysis Flowchart

- If all the references in the overlapping LMADs are read references, then there are no hazards. The compiler generates code that moves these LMADs to on-chip shared memory to reduce the cost of data transfer.
- if the overlapping LMADs contain at least one array write references, then the compiler does not generate shared-memory code for all of the overlapping LMADs. Instead, global memory access code is generated.

5.2 OpenCL Program Generation

The following problem description is similar to a problem description presented by Rahul Garg[11].

Given a Python function P, a table of array accesses T_a that occurs within a loop in P and a table of the loop bounds T_b, compute a heuristic solution to the following problem:

- generate a parallel OpenCL program P_{OpenCL}, including grid configuration and kernel, such that for all inputs $s \in S$ that are legal for P, the results produced by P and P_{OpenCL} are identical, that is $P(s) = P_{\text{OpenCL}}(s) | s \in S$.
- compute a new table T'_a for array accesses representing the memory addresses in the shared memory on the GPU, and a sequence of OpenCL code that loads T_a from the GPU's global memory into shared memory. Modify P_{OpenCL} to make array accesses in P_{OpenCL} refer to T'_a rather than T_a.

5.2.1 Host Code Generation

The host code generation mainly consists of transferring LMADs to global memory on the device and setting up arguments for the kernel execution.

Copying from Host Memory to Device Memory

There are significant differences in device resource allocation between CAL and OpenCL. In CAL, the programmer may create 1D, 2D or 3D area storage space in the device, and the device memory address is mapped to a host memory address. Thus, when CAL API is used to allocate memory space in the device memory, the API returns a global pointer pointing to a host memory address. The programmer can use this global pointer to transfer data from and to device memory space in the same fashion as for a transfer between two host memory addresses.

OpenCL uses a different kind of API format to manipulate device memory. OpenCL requires the programmer to initialize device memory objects with their sizes, then the programmer must use OpenCL APIs to request transfers that are handled by OpenCL itself.

To reduce the data transfer bandwidth from host memory to device memory, Garg's jit4GPU determines the smallest rectangle shape from source memory space that covers all the effective elements in an LMAD group in the memory. Then jit4GPU creates a rectangular memory space of the same size on the device memory for storage of the group. Such an approach works for CAL but is not suitable for OpenCL.

Rather than calling CAL API to get a host memory pointer, the compiler for OpenCL must create a host memory buffer area. The data in the source rectangle is copied to this buffer and then the content of the buffer is transferred to device memory. This change enables jit4OpenCL to use Garg's analysis to reduce data transfer between host and device but has the shortcoming of additional host memory to accommodate memory copy, and additional instructions to be executed in the host.

jit4OpenCL now adapts OpenCL 1.1 specification and tested all the experiments using OpenCL 1.1 implementations. The compiler first test if the input RCSLMAD is a contiguous memory region, if the answer returns yes, it will trasfer the memory region using two OpenCL APIs: `clEnqueueReadBuffer(...)` or `clEnqueueWriteBuffer(...)`. Otherwise the compiler invokes `clEnqueueReadBufferRect(...)` or `clEnqueueWriteBufferRect(...)` for copying rectangular memory regions.

5.2.2 Grid Configuration Generation

Grid configuration controls the parameters for a kernel issue. A grid configuration must determine the number, the geometry, and the size of each thread group as a parameter prior to kernel execution.

The size and geometry of a thread group may affect performance. Groups that have few threads may fail to utilize shared-memory space effectively, while groups that are too big would probably overflow the register file or would not execute because of insufficient shared-memory space to hold all the data. To optimize shared-memory utilization the grid configuration must be determined according to the program's array access pattern.

The current implementation of jit4OpenCL creates a grid configuration such that:

- the total number of threads in a thread group is a multiple of 32.
- the thread group has the geometrical shape of a rectangle.
- threads in a group collectively do not require more shared memory space than the hardware can offer.
- the group is big enough to efficiently use shared-memory space.

The grid is configured by a heuristic search method. The result depends on the hardware availability information that is retrieved via the OpenCL API, and on the program shared memory usage.

5.2.3 Kernel Code Generation

The kernel code generation transforms the information passed by unPython into an OpenCL kernel, with additional optimizations.

While invoking jit4OpenCL, unPython passes the following arguments to jit4OpenCL:

- a serialized abstract syntax tree (AST) of the source code in the form of a string.
- a set of LMADs describing the array-access references in the source code.

Jit4OpenCL reconstructs an abstract syntax tree from the string and traverses the tree to generate target OpenCL kernel code. The following content describes the kernel code generation approach.

5.2.4 Transforming Loops

Translating Perfect Parallel Loop Nests According to Banerjee[3],"a sequence of loops form a nest, if each loop(except the first) is totally included within the previous loop", such a sequence of loops form a perfect loop nest. The number of loops is said to be the "depth of the loop nest". The loop that has $i-1$ outer loops is called "the i^{th} level loop of the loop nest".

We expand the definition of perfect loop nest into the parallel domain:

Definition 3. *If different iterations of a loop can be performed simultaneously or out-of-order, then the loop is a parallel loop. The entire loop finishes when all its iterations are finished.*

Definition 4. *A sequence of parallel loops form a perfect parallel loop nest, if each loop (except for the first) is totally included within the previous parallel loop, and all the non-looping instructions are inside the deepest loop.*

Assume that there are p levels of parallel loops in the perfect loop nest, counting from the outermost loop, we denote those loop indexes as $l_1, l_2, \ldots l_p$ respectively, this sequence of loop indices makes up a vector. We also denote their index domains as $d_1, d_2, \ldots d_p$ respectively. Jit4OpenCL generates a p-dimensional thread grid configuration (in reality, OpenCL and CUDA only support dimensions not exceeding 3), and the i^{th} dimension's iteration domain starts from 0 to $d_i - 1$.

Code 5.1: Python Stencil Computing Code

```
1 for y in gpurange(n-2):
2         for x in gpurange(n-2):
3                 w = float32(0.2)
4                 B[y,x] = w*A[y+1,x+1] + w*A[y+2,x+1]
5                     + w*A[y,x+1] + w*A[y+1,x] + w*A[y+1,x+2]
```

Code 5.2 is an example frequently used in graphic processing, it represents the operation of a low-pass filter. We will use this simple stencil computation as an example to demonstrate how jit4OpenCL generates OpenCL grid configuration: Jit4OpenCL first recognizes that the depth of perfect parallel loop is 2, the values of d_1 and d_2 are both $n-2$. Thus, the compiler generates a 2-dimensional grid, with the domain of the first dimension (the y dimension) and the second dimension (the x dimension) both stretching from 0 to $n-3$, as shown in Figure 5.2, where a small square represents an issued thread. In this example, $(n-2)\times(n-2)$ threads are issued, and each thread has its own loop index. For example, the thread at the upper-left is assigned with its $x = 0$ and $y = 0$, thus, that thread computes `B[0,0]`.

Translating Serial Loops

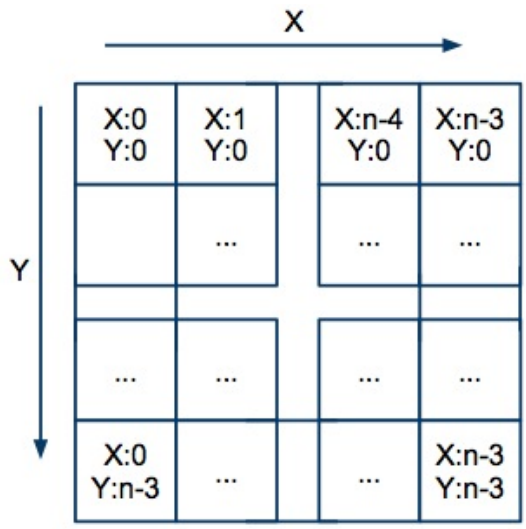

Figure 5.2: Stencil Program Thread Grid Example

Definition 5. *A serial loop is a common loop in which the execution of the* x^{th} *iteration depends on one or more of the predecessor iterations. The predecessors of iteration* x *are the iterations* $1, 2 \ldots (x-1)$.

For serial loops that are in a loop nest body, jit4OpenCL performs the following transformations:

- strip mines serial loops according to the RCSLMAD analysis for tiled array accesses;
- inserts code to transfer the tiled array block that is going to be accessed in the stripmined loop from off-chip global memory onto on-chip shared memory — this code is inserted immediately before the execution of the stripmined loop body;
- replaces the global array accesses in the loop body with corresponding shared-memory accesses;
- inserts code to transfer modified array tiles from shared memory back to global memory (this code executes immediately after the stripmined loop body);
- inserts synchronization barriers right after the data transfer between shared memory and global memory for data consistency.

The later part of this chapter describes the process of creating data transfer code and of redirecting array access references. Section 5.2.5 illustrates the translation of a serial loop.

Dealing with Imperfect Loop Nests Not all applications contain perfect loop nests that can be easily optimized for performance by a compiler, some applications execute

imperfect parallel loop nests. For instance, some instructions may appear between different levels of loops instead of appearing inside the deepest loop. Dealing with imperfect loops enables the compiler to handle a larger set of applications.

Jit4OpenCL's approach is to break down the imperfect parallel loop nest into two or more smaller perfect parallel loop nests so that the compiler can then handle each of them, but still keeping the data/instruction dependency in the program. Given the Abstract Syntax Tree(AST) of the imperfect loop nest, jit4OpenCL splits an imperfect parallel loop nest into two or more smaller perfect parallel loop nests, while preserving dependencies.

Because jit4OpenCL does not perform instruction scheduling, the transformation guarantees that before the execution of an instruction, all its predecessor instructions are executed, to ensure the correctness of the program. Algorithm 5.2.4 describes the approach. The compiler invokes SplitAST method to split the input AST.

Shared Memory Loading/Storing Code Generation

Given a set of RCSLMADs R that will be used in the loop body, the thread's global and local IDs: ID_g, ID_l respectively, the group size and the grid size in each dimension x (S_x^{group} and S_x^{grid}, respectively), the loop stripmining length L_{sm}, and the stripmined outer loop counter c, jit4OpenCL can calculate the data elements that will be accessed within the inner loop by changing the RCSLMAD's domain.

For each RCSLMAD $r \in R, r : f(\vec{i}), \vec{i} \in D$, the following address will be referred in the inner loop:

$$f(\vec{i}), \vec{i} \in D'$$

where for each dimension j in D', if j is a parallel dimension, then:

$$ID_g \times S_j^{group} \leq j < ID_g \times (S_j^{group} + 1) - 1$$

otherwise

$$L_{sm} \times c \leq j < L_{sm} \times (c + 1) - 1$$

It is obvious that $f(\vec{i}), \vec{i} \in D'$ is also a RCSLMAD. To verify that, just replace D with D', and the characteristics of RCSLMAD defined in previous chapter still stand. Moreover, D' covers the loop index domain of the inner loop for all the threads in the group. To utilize shared memory, $f(\vec{i}), \vec{i} \in D'$ is transferred to shared memory, where it gets compressed using the method stated in Theorem 2.

In the host code jit4OpenCL declares a dedicated, independent, shared-memory space for each $r \in R$ that is accessed in a serial loop in the code. The size of this space is exactly $L_f(D')$ where $r : f(\vec{i}), \vec{i} \in D$. Therefore, jit4OpenCL can generate shared-memory loading/storing code using the following method:

1. assign an unique order ID to the thread, using the method discussed in Section 4.3.3;

2. for every RCSLMAD $r : f(\vec{i}), \vec{i} \in D'$, the i^{th} thread loads/stores the i^{th} element from $f(\vec{i}), \vec{i} \in D'$, and stores it to the i^{th} position in the corresponding RCSLMAD in shared-memory space.

Adding Synchronization

Synchronization instructions are interleaved between shared-memory load/store operations and the computing instructions to ensure that data dependencies are honored and the memory accesses are correct.

Redirecting Array Access References

After the shared-memory loading stage, the array accesses references in the source code, which have shared-memory copies have to be redirected to their copies in shared-memory space. Two changes must be made:

1. change of the array head pointers.

2. change of the subscripts.

Changing the array pointer is simple, we just need to replace the array name with the one declared to point to shared-memory space. Changing of the subscripts requires jit4OpenCL to construct a new normalized RCSLMAD for the copy in shared memory.

Given a RCSLMAD $r : f(\vec{i}), \vec{i} \in D'$ referring to an array access in global memory (where D' is the tiled domain that will be accessed within the strip-mined serial loop by all the threads in the group), and the shared memory with size larger than or equal to $L_f(D)$, the compiler constructs another RCSLMAD $g(\vec{i}), \vec{i} \in D''$, where D'' is a normalized representation of D', and $g(\vec{i}), \vec{i} \in D''$ points to memory locations in the shared memory space. Normalization shifts the region of each dimension to start from 0 and leaves the region size unchanged. The compiler redirects the array access references by replacing the access to the first RCSLMAD with the new one.

5.2.5 Scenario Example of Kernel Transformation

This section uses the annotated Python code shown in Figure 5.2 as an example to illustrate how the compiler works. This code is an implementation of matrix multiplication in Python. The top two loops in the nest, loop `i` and loop `j`, are parallel; the innermost loop, loop `k`, is a serial loop because of the data dependence in the scalar variable `sum`. There are three array access references: `A[i,k]`, `B[k,j]` and `C[i,j]`.

To generate a kernel, the compiler extracts the code content within the parallel loops for kernel body generation. The information in these loops is used to generate thread grid configuration. In this example, a 2-dimensional grid $\langle$`i, j`$\rangle$ configuration is generated.

Code 5.2: Python Matrix Multiplication Code (Before Translation)

```
1 for i in gpurange(N):
2   for j in gpurange(N):
3     sum = 0
4     for k in range(N):
5       sum = sum + A[i,k]*B[k,j]
6     C[i,j] = sum
```

Code 5.3: Translated Kernel Body (intermediate result)

```
1 int sum = 0;
2 int i = gid_i*group_size_i + lidi;
3 int j = gid_j*group_size_j + lidj;
4 for (int sp_k = 0; sp_k < N; sp_k+= sp_size){
5   for(int k = 0; k < sp_size; k+=1){
6     sum = sum +A[i][sp_k+k]*B[sp_k+k][j];
7   }
8 }
9 C[i][j] = sum;
```

Now consider the content inside the parallel loops: to generate an efficient kernel that can utilize shared memory, the serial loop `k` has to be strip mined in order to allow the GPU to have tiled array access. Also, in the generated kernel the parallel-loop indexes `i` and `j` are replaced by a combination of OpenCL group IDs, group sizes, and thread local IDs, as shown in Code 5.3. In Code 5.3, `gid_i` and `gid_j` are the group ID of variables `i` and `j` respectively; `group_size_i` and `group_size_j` are their size along the `i` and `j` dimensions, respectively. Line 2, and line 3 calculate the correct value of `i` and `j`.

The next step is to generate shared-memory utilization code. The memory loading/storing code is inserted between the inner loop and the outer loop of the strip-mined serial loop. To correctly generate code, the compiler must identify the set of array accesses that is referenced in the innermost loop by all the threads in the group. This identification is done by calculating the RCSLMAD representation of the array accesses in the strip-mined loop body (line 6 in Code 5.3). The smaller iteration domain of loop index, together with the original LMAD function, identify the set of memory locations that will be accessed by the loop body.

The number of threads in a group can be calculated as the multiplication of `group_size_i` and `group_size_j`. A unique order value is assigned to each thread, which can be calculated as `lidi*group_size_i + lidj` or `lidj*group_size_j + lidi`. Next, the i^{th} thread must load the i^{th} element in a RCSLMAD, and the global-memory array reference must be replaced with a shared-memory reference. All these changes yield Code 5.4.

Code 5.4 is simplified by omitting details from the kernel instance compiled using

Code 5.4: Kernel Body with Shared memory Utilization

```
int lidi = get_local_id(0);
int lidj = get_local_id(1);
int gid_i = get_group_id(0);
int gid_j = get_group_id(1);
int order = lidi*group_size_i + lidj;
int sum = 0;
for (int sp_k = 0; sp_k < N; sp_k+= sp_size){
  A_sm[order] = A[N*(gid_i*group_size_i+ (order/group_size_i))
         + 1*(sp_k*p_size+(order % group_size_i))];
  B_sm[order] = B[N*(sp_k*sp_size + (order/sp_size))
         +1*(gid_j*group_size_j+(order/group_size_j))];
  for(int k = 0; k < sp_size; k+=1){
   sum = sum + A_sm[group_size_i*lidi+1*k]
         *B_sm[group_size_j*k+1*lidj];
  }
  //no need to generate shared memory store because
  //A and B are read only.
}
C[i][j] = sum;
```

jit4OpenCL. When executed on an nVidia Tesla Architecture this code achieves about 7× better performance than Code 5.2. This performance improvement indicates that the method that jit4OpenCL uses is effective to generate efficient OpenCL code for this type of computation on nVidia GPUs.

5.3 Chapter Conclusion

This chapter presented a high-level description of the techniques used in jit4OpenCL for just-in-time compilation and kernel generation followed by a scenario demonstration, in which it discussed in detail some of the implementation challenges that had to be overcome to use the analysis described in Chapter 4 for kernel generation. This chapter also discussed the need for additional synchronization to prevent the generation of incorrect results due to dependencies.

Algorithm 1 The Algorithm for splitting AST

SplitAST:

Require: the imperfect parallel loop nest root node r as input

initialize queue of ASTs: A_1
initialize stack s for loop information storage (each stack frame holds the information of a loops loop-id, the iteration domain)
initialize queue of ASTs: A_2
enqueue r to A_2
call RecursivelySplitAST
return A_1

RecursivelySplitAST:

extract first AST t from A_2, if A_2 is empty then exit
construct a stack frame f, record t's loop info f, push f to the stack s
if t's loop body only contains another parallel loop p **then**
  enqueue the loop body of t to A_2
else
  {loop body contains more than one parallel loop}
  split the loop body instructions using parallel loop statement as delimiter into a list L of instruction lists and parallel loops
  for each l in L **do**
    if l is a list of non-loop instructions **then**
      $p' =$ BuildLoop
      enqueue p' to A_1
    else
      {it is a parallel loop}
      call recursivelySplitAST
    end if
  end for
end if
pop s

BuildLoop:

initialize a temporary stack s_2
initialize an AST node pointer $t = l$
while s is not empty **do**
  get f by popping stack top s
  re-construct a loop node n with the information stored in f
  make n the parent node of t
  make t points to n
  push f to s_2
end while
while s_2 is not empty **do**
  get r by popping stack top s_2
  push r to s
end while
return t

Chapter 6

Experimental evaluation

This chapter explains the experimental methodology used to evaluate the performance of the unPython and jit4OpenCL framework and then presents an analysis of the experimental results. The results of the experiments reported in this chapter support the following claims:

1. The execution on a CPU-GPU heterogeneous architecture can deliver significant performance speedup in some applications. However, not all applications can achieve speed improvements.

2. The process for just-in-time compiling and data transfer incurs significant overhead. In some cases the overhead dominates the execution time.

3. jit4OpenCL is targeting an abstract architecture (OpenCL) rather than AMD GPU, the jit4OpenCL executing on an AMD GPU is slower than jit4GPU in some cases.

6.1 Methodology

An important claim for jit4OpenCL over jit4GPU is that jit4OpenCL generates OpenCL code that can be executed in all platforms that support OpenCL while jit4GPU is restricted to execution in AMD GPUs. Therefore the experimental evaluation of the performance of jit4OpenCL is conducted on two machines, one with a nVidia GPU and the other with an AMD GPU.

The machine with nVidia GPU is a desktop Intel Core 2 Duo E5200 (2.5GHz), nVidia GTX260, and 6GB DDR3 memory. The result is obtained on Ubuntu 9.10 32-bit using GCC 4.4 and nVidia Computing SDK 3.2 Beta.

The machine with an AMD GPU has a Phenom II X4 925 2.8GHz CPU, Radeon 5850 1GB GPU, 4GB DDR3. The OpenCL driver and compiler used is Catalyst 10.7 OpenCL 1.1 update with AMD Stream SDK 2.2 on Ubuntu 10.04 64-bit.

Codes on both machines are compiled using GCC flag `-O3`.

Several highly parallel applications are chosen as benchmarks in this evaluation. The Coulomb Potential (abbr. CP), N-body simulation and Mandelbrot, Blackscholes that derived from Parboil benchmark suit by Garg [12], are used, along with Matrix-Multiplication. These applications are widely used in scientific computing and represent a class of applications that can be parallelized for efficient execution. Those applications embed high stream parallelism, thus they are suitable for execution on stream processors such as GPUs.

We also included benchmark testing results of Matrix-Transpose and Stencil applications on nVidia platform for reference. These two applications show low computing/data ratio and thus are not recommended to be compiled using jit4OpenCL. The test results support this received wisdom.

In the experimental evaluation, each test is executed 40 times on each machine (execute on CPU for 20 times, another 20 executions are on GPU). Throughputs or speedups are plotted on figures with error bars indicating a 95% confidence interval. Each GPU execution is profiled to record the take down time, the JIT compiler's total execution time, including the time used for compiling OpenCL code, data transfer, data analysis and actual computing. Those histogram figures also show the comparison results against generated OpenMP code or highly optimized CPU library code.

6.2 Result Analysis

6.2.1 Results on nVidia Machine

The performance benchmark result of each application with different size of input is plotted in the figures in this section. In the figures code generated by jit4OpenCL is compared to either highly-optimized ATLAS library, hand-written C code, or OpenMP code generated by unPython, depending on the availability.

Figure 6.1 shows the execution performance of the Matrix-transpose application. The jit4OpenCL code is compared against hand-written C implementation. The benchmark shows that highly optimized CPU code outperforms the GPU code in all tests.

Figure 6.2 compares the stencil performance throughput between hand-written C code and the generated OpenCL code on nVidia GPU. This experiment shows that hand-written C code runs faster than jit4OpenCL for all input data size.

Profiling shows that in Matrix Transpose and Stencil, most of the execution time of the jit4OpenCL code is spent on overhead because of the low computation/data transfer ratio (the ratio of Stencil is $5:1$ while Matrix Transpose is $1:1$). Although the kernel execution on GPU runs in just several hundredth of a second, the data transfer and JIT compiling time overhead exceeds the total time required to execute the CPU code. This benchmark suggests that, in order to take advantage of the GPU, programmers need to carefully select the applications that they will run in such architectures.

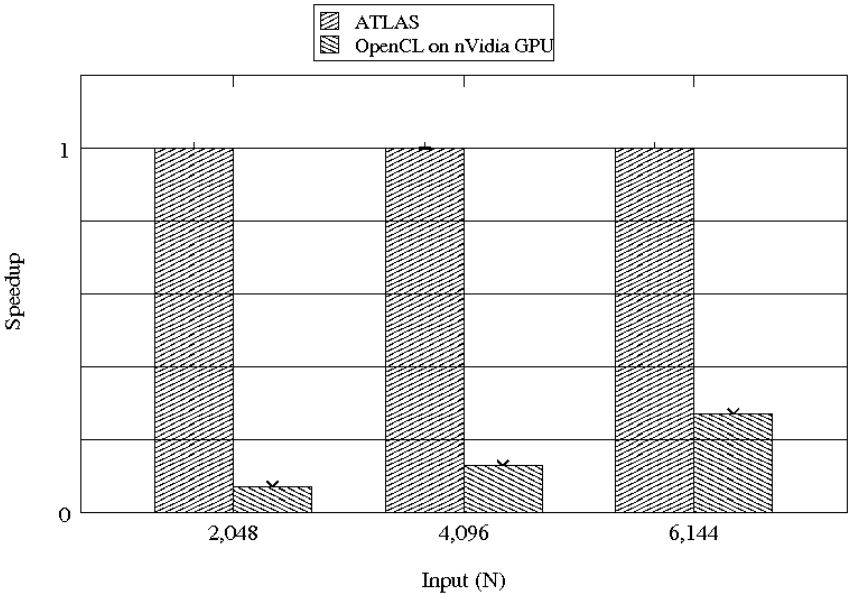

Figure 6.1: Matrix Transpose Result (nVidia)

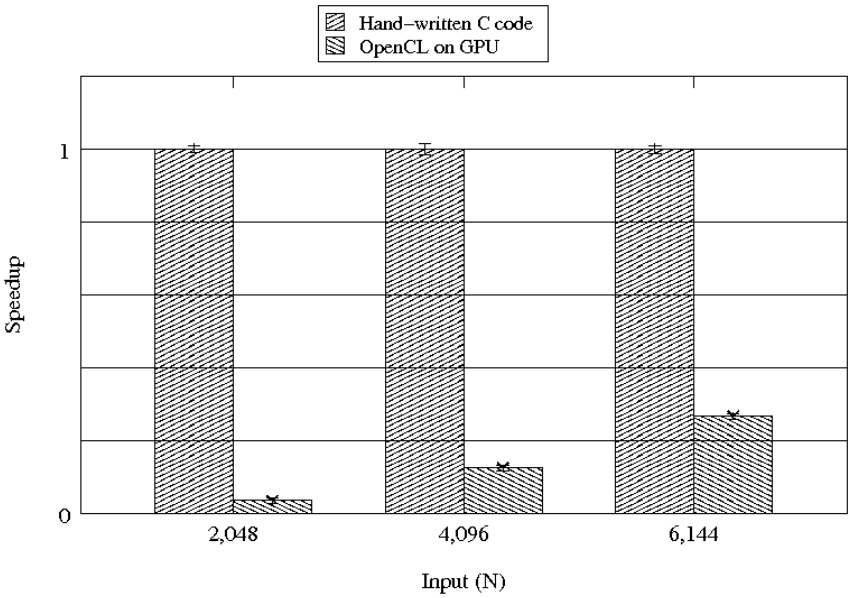

Figure 6.2: Stencil Result (nVidia)

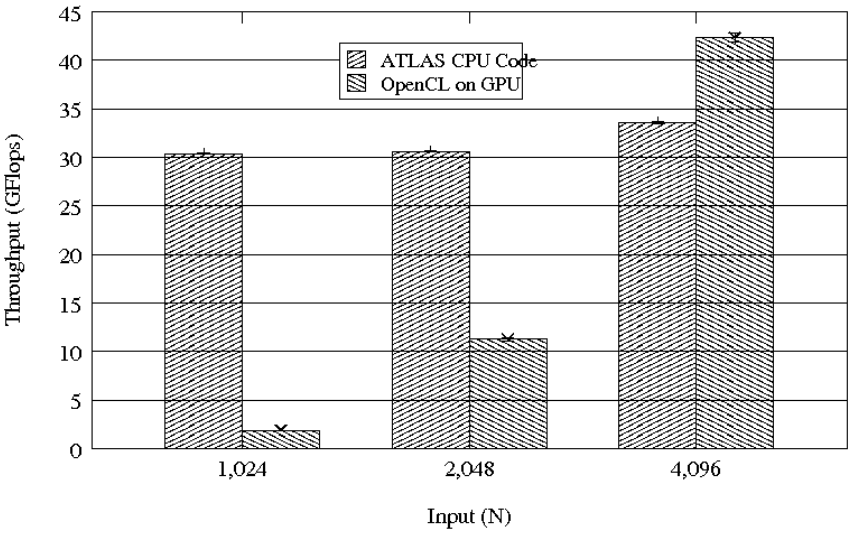

Figure 6.3: Matrix Multiplication Result (nVidia)

Figure 6.3 shows the throughput of matrix-matrix multiplication execution on the nVidia GPU measured in GFlops using code generated by jit4OpenCL. The result is compared to highly optimized code from the Automatically Tuned Linear Algebra Software (ATLAS) code. ATLAS features a state-of-the-art auto-tuning linear-algebra library toolkit for different homogeneous computing system. The performance of the ATLAS library ranges from 30 Gflops to 33 Gflops throughout the test, while jit4OpenCL can reach as high as 42 Gflops.

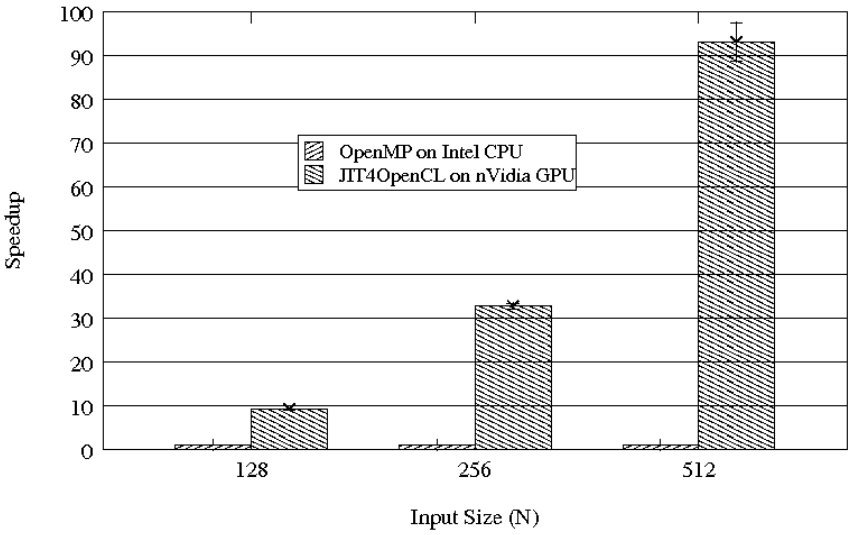

Figure 6.4: Coulomb Potential Energy Result (nVidia)

Figure 6.4 is the comparison result of the Coulomb Potential (CP) on the nVidia GPU. The figure suggests that up to 92 times speedup can be achieved when the input data size is large (N=512). CP has a high computation/data transfer ratio and therefore it requires

a small transferring data set, yielding a small data transfer overhead.

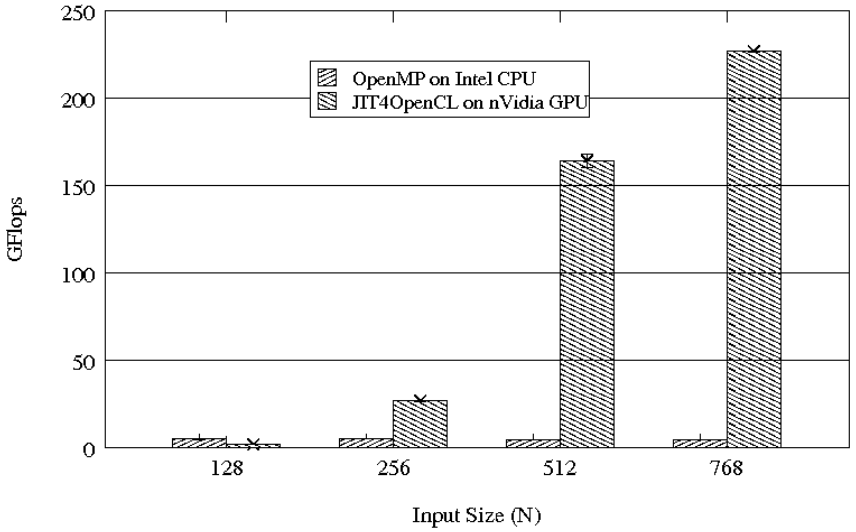

Figure 6.5: N-Body Simulation Result (nVidia)

N-Body simulation result is shown in Figure 6.5. Jit4OpenCL delivers a peak performance of 227 Gflops, while the generated OpenMP code runs at a speed of 4.7 Gflops. This is a 49x speedup achievement that can be obtained when the problem size is as large as 768. However, the performance degrades when the problem size is small. The OpenMP program performs steadily at a speed of 4.6 to 4.8 Gflops through out the test.

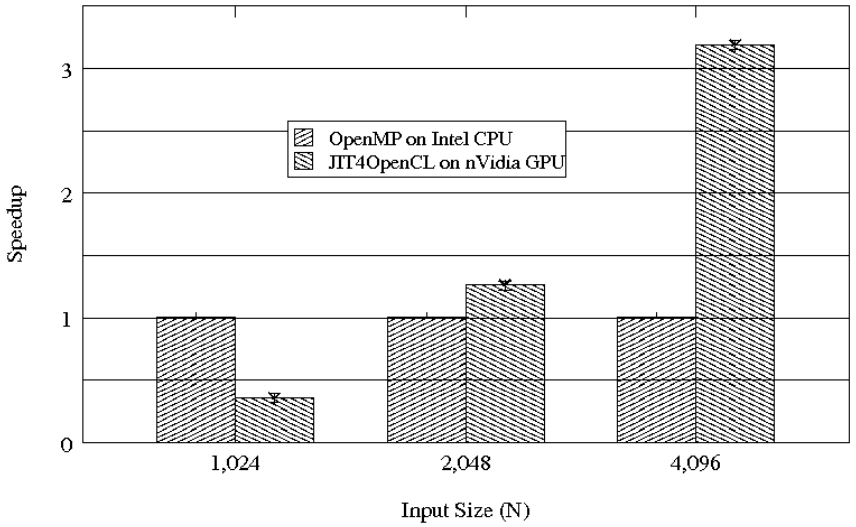

Figure 6.6: Blackscholes Filter Result (nVidia)

Figure 6.6 shows that the Blackscholes application has a performance gain of 3.3x when the problem size is as large as N=4096, while the speedup is 1.26x when N=2048 and 0.35x when N=1024.

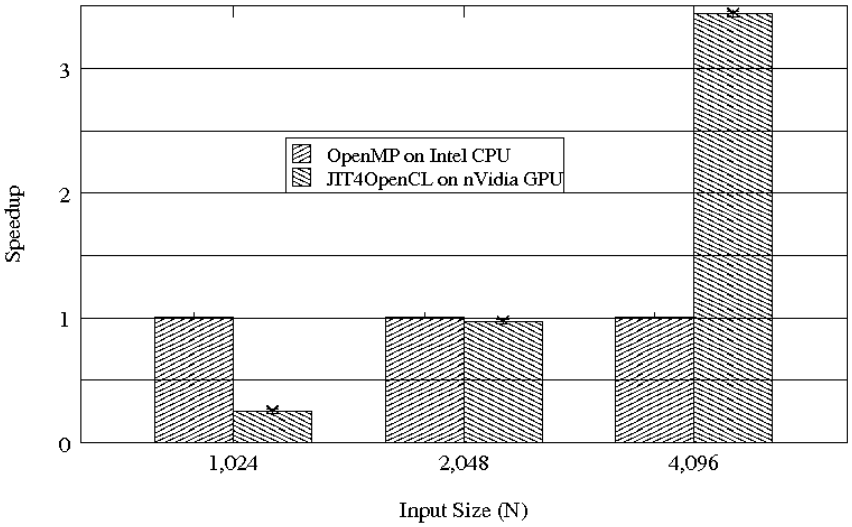

Figure 6.7: Mandelbrot Result (nVidia)

Figure 6.7 gives the result of performance for code generated by jit4OpenCL for the Mandelbrot application. It shows that Mandelbrot can have performance gain when the input problem size is large. When the input data size is equal or less than 2048, the performance actually degrades; when the input data size is 4096, we observed a 3.9x speed up.

6.2.2 Jit4OpenCL Results Compared with Jit4GPU on AMD Machine

We also obtained the performance results of Matrix-Multiplication, N-Body simulation, Coulomb Potential, Blackscholes and Mandelbrot on the AMD platform. The result is compared against jit4GPU and unPython-generated OpenMP code (except Matrix-Multiplication that is compared against code from the ATLAS library). We expect the jit4OpenCL performance on AMD architecture to be lower than that of jit4GPU — because jit4OpenCL is targeting OpenCL architecture rather than AMD architecture.

Figure 6.8 shows the execution result of jit4GPU and jit4OpenCL on the AMD GPU. From the figure we can see that jit4OpenCL cannot generate efficient GPU code on AMD platform for the Matrix-Multiplication application. The jit4OpenCL has around 12x performance degradation compared with CPU ATLAS, while the jit4GPU can reach an execution rate as high as 233GFlops. The CPU ATLAS library has a 67Gflops throughput, compared to around 3Gflops that jit4OpenCL achieves. This result suggests that jit4OpenCL is not yet capable of generating some applications for AMD platform when it is tuned for nVidia architecture.

Figure 6.9 shows the performance result of Coloumb Potential on AMD platform. In

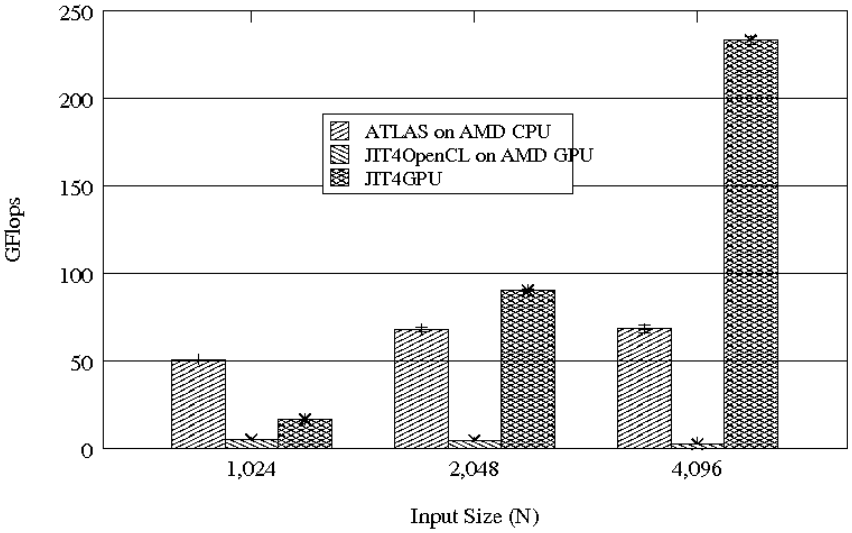

Figure 6.8: Matrix-Multiplication Result on AMD

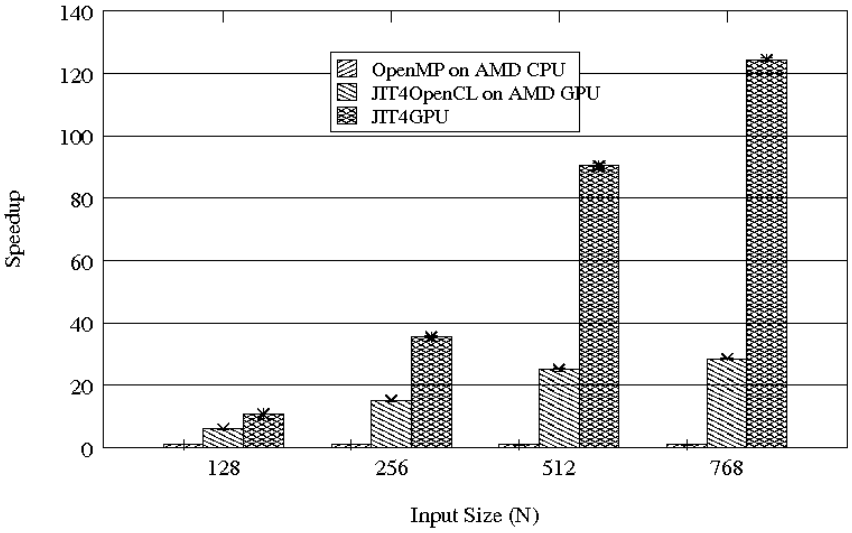

Figure 6.9: CP Result on AMD

the figure, Jit4OpenCL has a speedup factor of 29x compared to OpenMP while jit4GPU performs 124x faster than the CPU code. Compared to jit4GPU, jit4OpenCL is 4.3x slower.

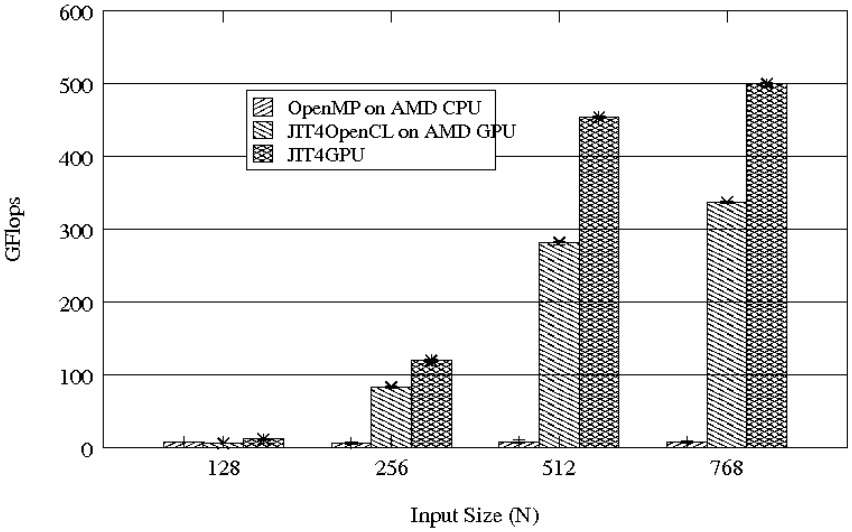

Figure 6.10: N-Body Result on AMD

Figure 6.10 shows the performance result of N-Body simulation on AMD machine. From the figure we can see that the performance of jit4OpenCL is about 71% of that of jit4GPU. Jit4GPU can reach up to 500GFlops while jit4OpenCL is 158GFlops slower.

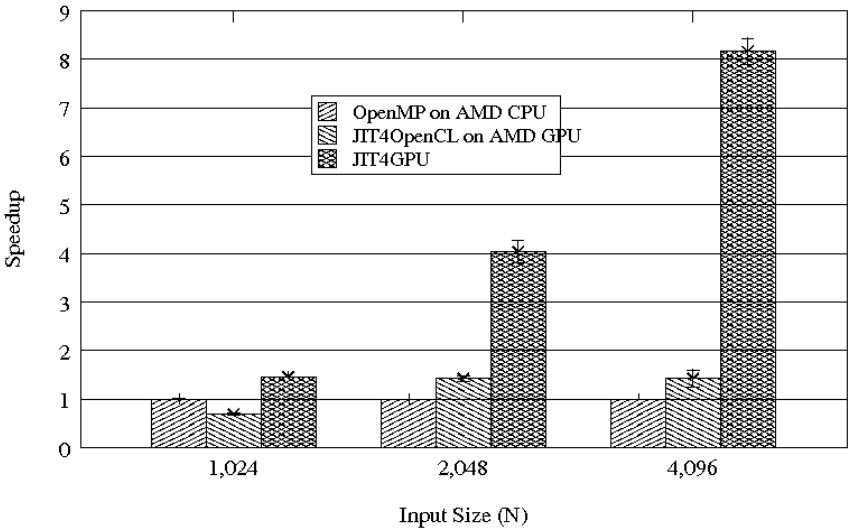

Figure 6.11: Blackscholes Result on AMD

Results for Blackscholes are shown in Figure 6.11. The figure tells that given a large problem size input (N=4096), jit4OpenCL can have roughly 1.4 times speedup compared to generated OpenMP code, while jit4GPU can perform 8 times faster than OpenMP.

The result in Figure 6.12 is interesting because the jit4OpenCL performance surpasses

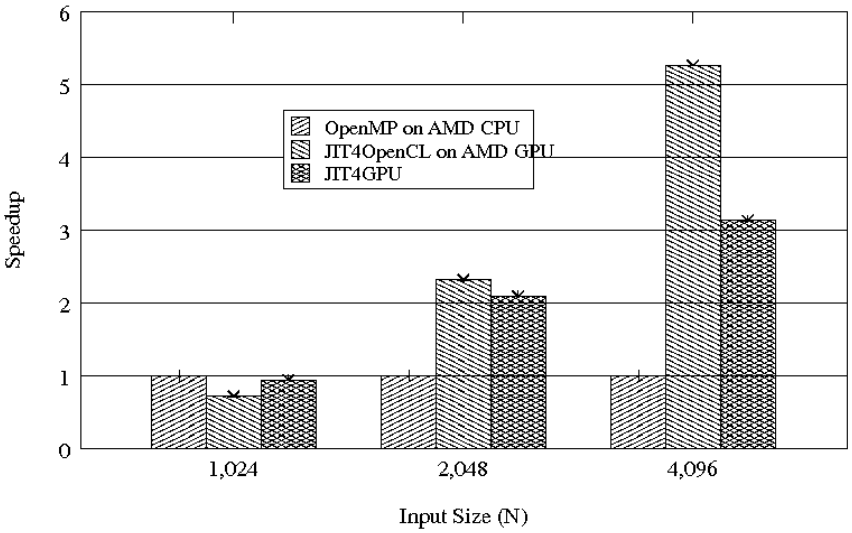

Figure 6.12: Mandelbrot Filter Result on AMD

that of jit4GPU and is 1.6x faster than jit4GPU on the same AMD machine for large input size (N = 4096). But when the input data size is small, neither jit4OpenCL nor jit4GPU can outperform generated OpenMP code.

6.2.3 Performance Analysis

Time Decomposition

The execution time of a program compiled by jit4OpenCL can be decomposed into three parts: jit4OpenCL analysis time, OpenCL JIT compiling time, Data Transfer time and Kernel computing time. A discovery is that for some benchmark applications, such as Coloumb Potential, Mandelbrot and Blackscholes, the GPU kernel computing time only takes up less than 15% of the total execution time. Most of the execution time is overhead. Figures 6.13-6.19 show the distribution of the execution time of each application on nVidia GPU; Figures 6.20-6.24 show the time distribution on AMD GPU. Each graph shows four components of the execution time:

- the time used for generating OpenCL host and device program
- the time used for generating GPU kernel binary by invoking external compiler
- the time used for copying processing data to temporary buffer
- the time used for GPU computing

Figure 6.13-6.19 show that the compilation and data transfer overheads are non-trivial. For applications that are not computing-intensive, such as Matrix-transpose, Stencil, kernel execution time only take up a small portion of the overall execution time. However, even

for the computing-intensive applications such as Matrix-multiplication, CP, the combined overhead is around 50% of the total execution time at least. Although jit4OpenCL gains speedup on applications like Blackscholes and Mandelbrot, most of the execution time is still consumed in combined overhead. In the N-Body simulation the kernel execution takes up to 12.5 seconds, marginalizing the combined overhead to 9% of the total execution time. However, the overhead is still significant.

Figures 6.20-6.24 show that on the AMD machine jit4OpenCL has a larger overhead compared to jit4GPU. In those figures, jit4OpenCL's kernel compiling time component is the time used for invoking AMD OpenCL compiler, and jit4GPU's kernel compiling time indicates how long the AMD CAL compiler takes to compile the generated CAL code.

- Figure 6.20 shows the jit4OpenCL and jit4GPU Matrix-multiplication break down time on AMD machine. The Figure shows that the kernel execution of jit4OpenCL is 55x slower than jit4GPU. A reason may be that the jit4OpenCL-generated code conflicts with the design of AMD GPU architecture, probably is connected with memory-bank conflicting. Fixing the problem is a high-priority task in the next update of jit4OpenCL.

- Figure 6.21 shows the Coulomb Potential execution breakdown time on AMD machine. The result indicates that the kernel execution time of jit4OpenCL is about 5x slower than that of jit4GPU. Also, the jit4OpenCL analysis overhead takes 2 times longer than jit4GPU; the kernel compilation takes 2.5 times longer.

- Figure 6.22 plots the N-Body simulation time breakdown in the experiment. The long kernel computing time offsets the overhead when the input problem size exceed 512. Although the time spent on overhead of jit4OpenCL is 3x of that of jit4GPU, the kernel execution time of jit4OpenCL is 1.4x longer, resulting the overall time of jit4OpenCL around 50% longer than jit4GPU.

- Figure 6.23 shows the result of Blackscholes. The figure tells that: 1, jit4OpenCL-generated kernel is around 34x slower than jit4GPU; 2, the data transfer time required by jit4OpenCL doubles; 3, the kernel compilation takes 2.5x longer. The jit4OpenCL overall performance is around 20% of jit4GPU when input problem size N is 4096.

- The last figure in this chapter, Figure 6.24 shows an unexpected result. jit4OpenCL-generated kernel performs 6.3x faster than jit4GPU-generated kernel: when N=4096, jit4OpenCL took 0.088 second to complete kernel computing, while jit4GPU needs around 0.55 second. However, jit4OpenCL demands 0.15 second for data transfer, while jit4GPU only needs 0.037 second on average; the analysis overhead time of jit4OpenCL compared with jit4GPU is 2.7:1; the kernel compiling comparison is 1.2:1.

In sum, compared to jit4GPU, Jit4OpenCL needs an additional stage of compilation that translates generated OpenCL kernel into GPU binary code, and this additional compilation leads to big losts of performance; Jit4OpenCL requires longer time to transfer a same amount of data; also, the analysis time in jit4OpenCL is longer than jit4GPU. This is possibly because jit4OpenCL is derived from an old version of jit4GPU, whose analysis takes a longer time to finish, and that in the OpenCL context the initialization process is more complicated than that of AMD Stream platform. Large overhead hinders performance speedup.

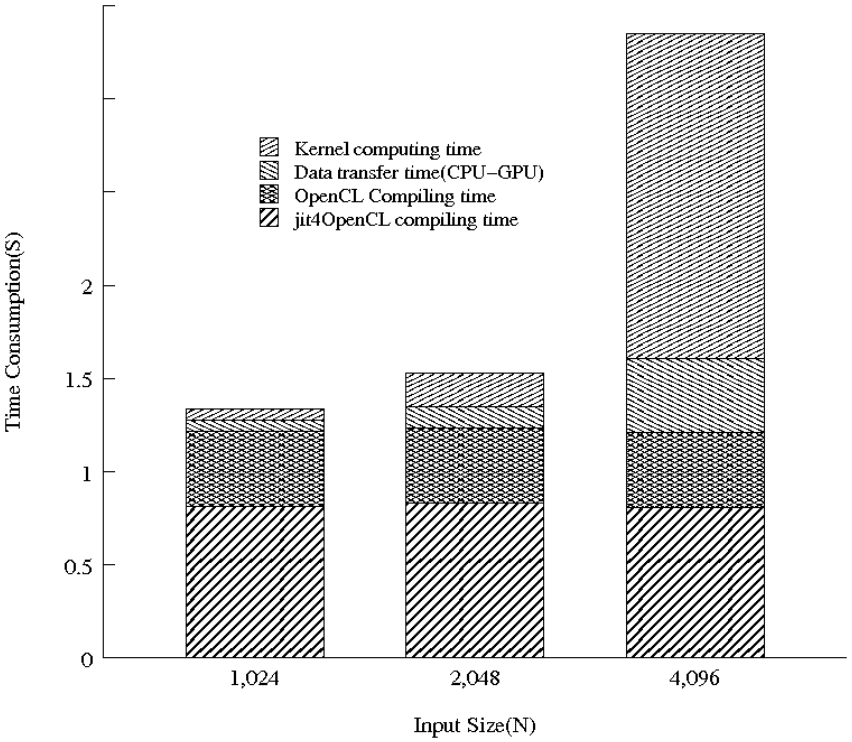

Figure 6.13: Matrix multiplication Execution Time Decomposition (nVidia)

The figures in this section show that, as expected, jit4OpenCL has a constant compilation time for a certain application, regardless of the input problem size. Such overhead is determined by the following factors:

- The input program source code implementation: jit4OpenCL selects kernel generating and data transfer strategies according to the program being compiled.
- The performance of corresponding platform OpenCL compiler: different implementations of the OpenCL compiler for different target platforms have different compilation-time requirement for the same input.

The cost of data transfer, however, is dependent on the input problem size and the application characteristics. Stencil, Matrix-Transpose and Blackscholes, have a large comput-

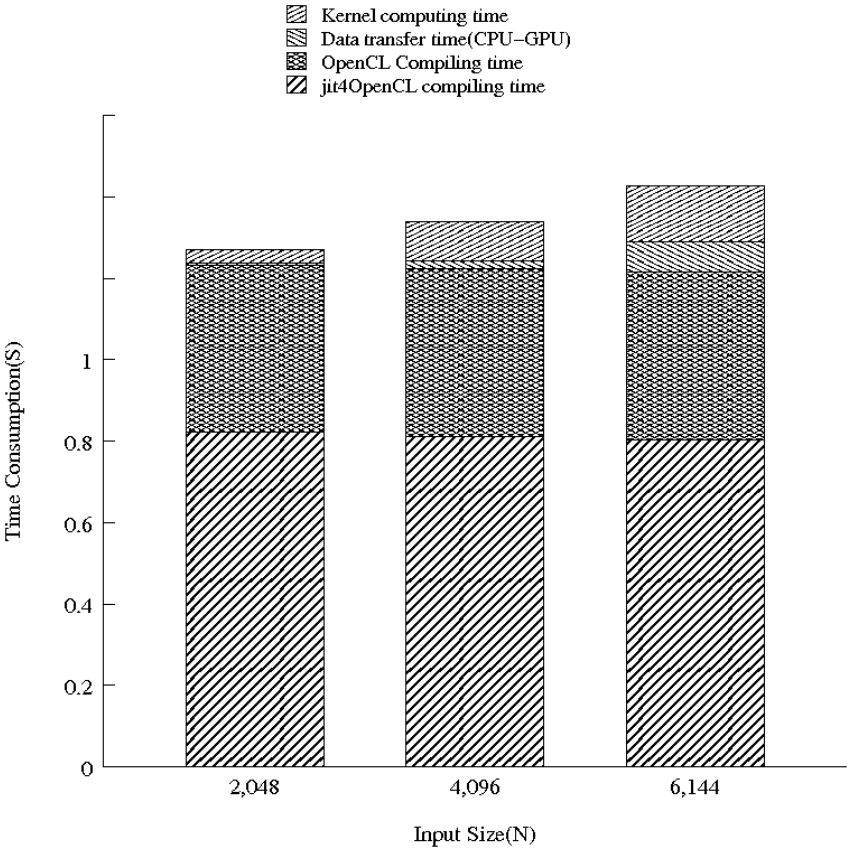

Figure 6.14: Matrix Transpose Execution Time Decomposition (nVidia)

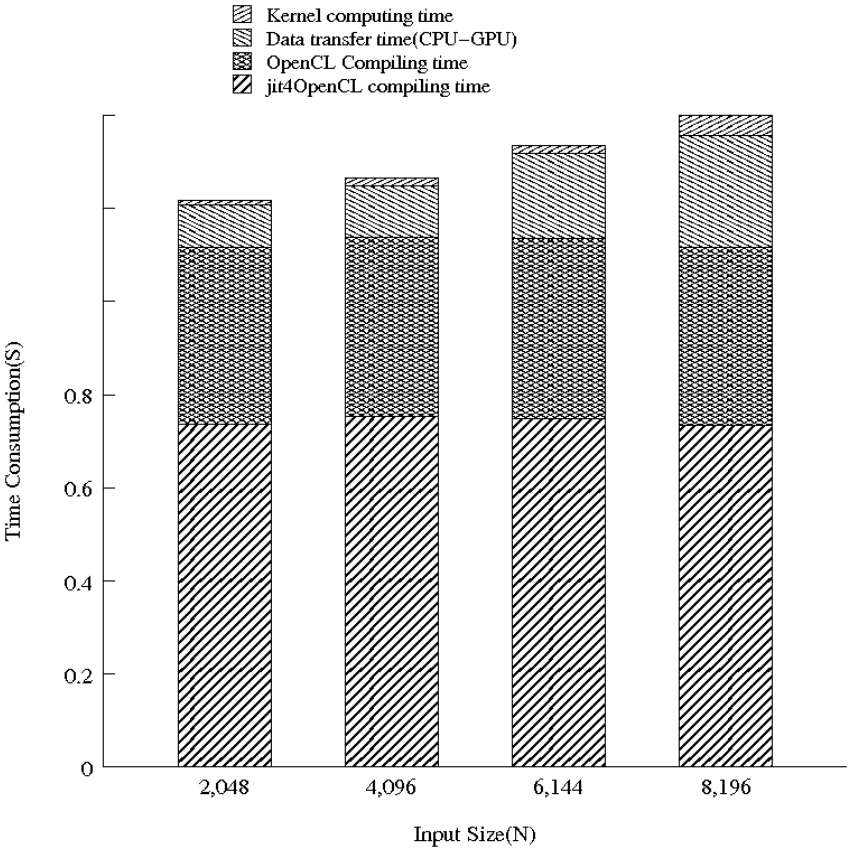

Figure 6.15: Stencil Execution Time Decomposition (nVidia)

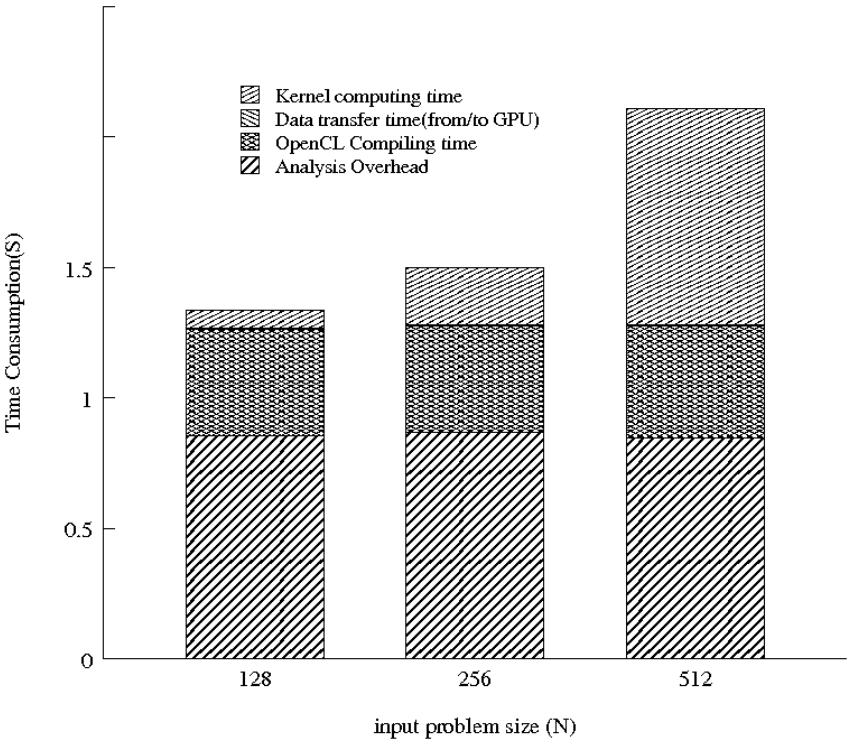

Figure 6.16: CP Execution Time Decomposition (nVidia)

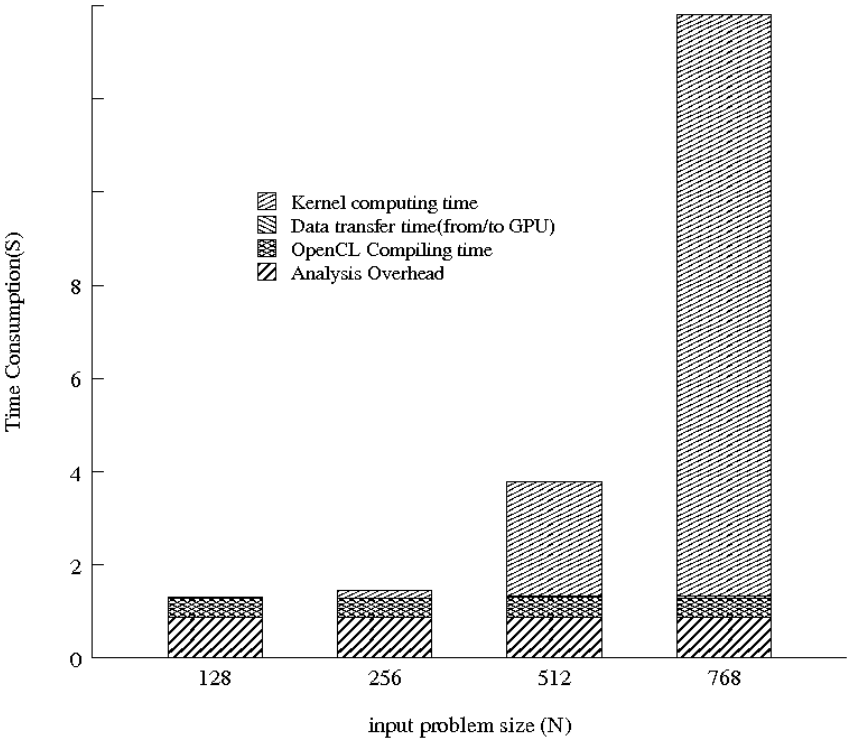

Figure 6.17: N-Body Execution Time Decomposition (nVidia)

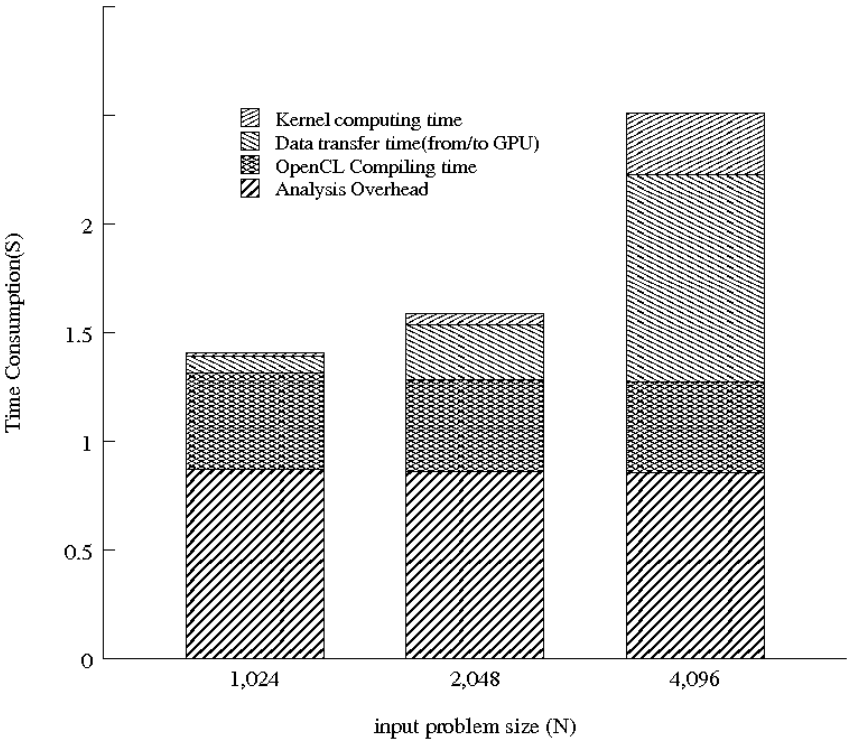

Figure 6.18: Blackscholes Execution Time Decomposition (nVidia)

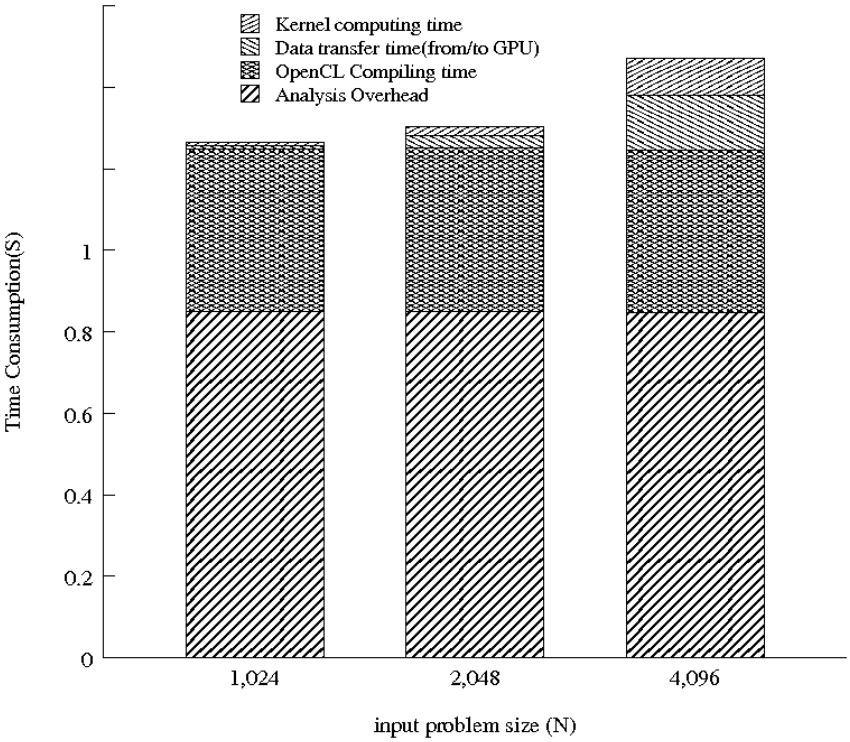

Figure 6.19: Mandelbrot Execution Time Decomposition (nVidia)

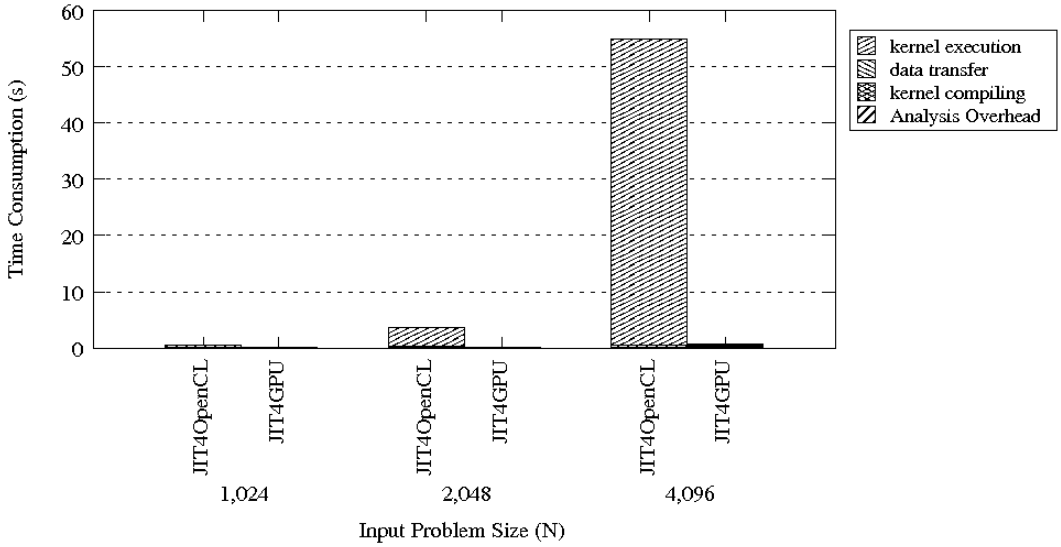

Figure 6.20: Matrix-Multiplication Execution Time Decomposition (AMD)

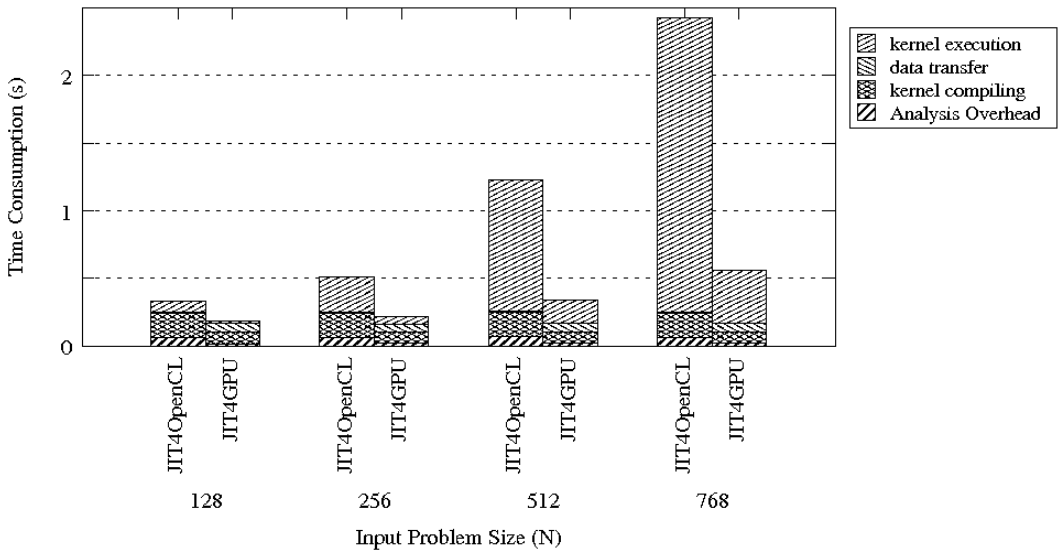

Figure 6.21: CP Execution Time Decomposition (AMD)

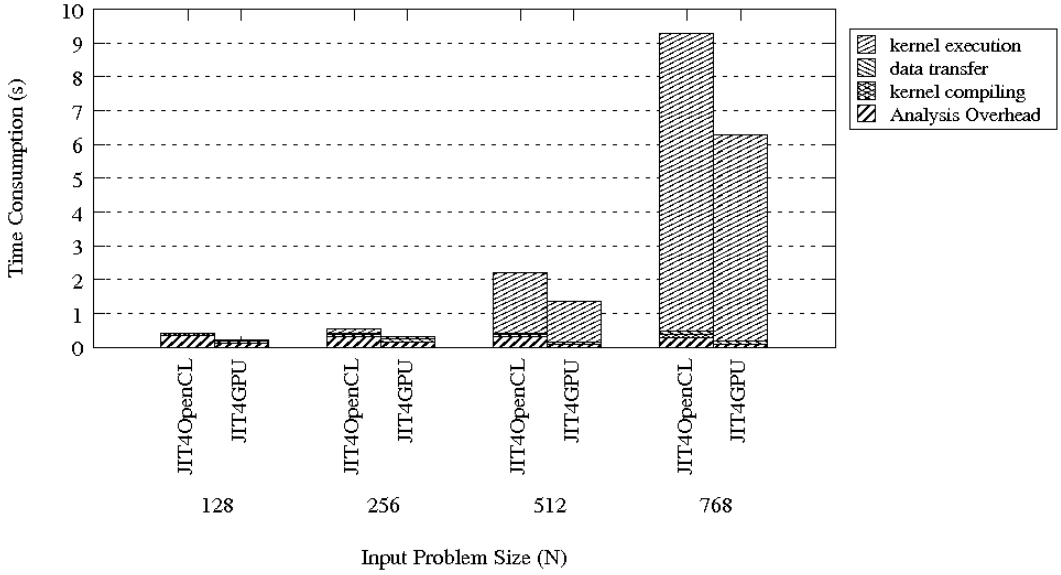

Figure 6.22: N-Body Execution Time Decomposition (AMD)

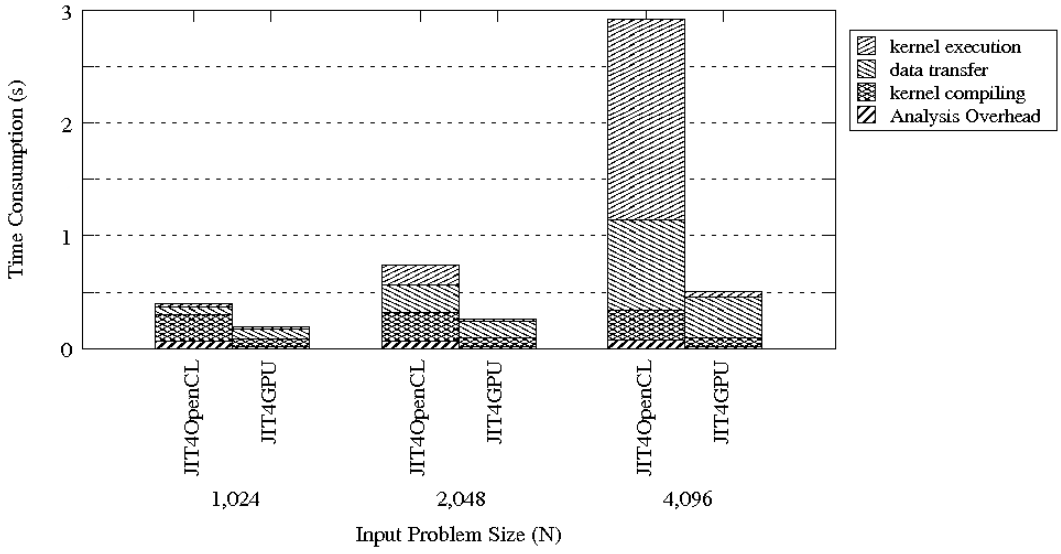

Figure 6.23: Blackscholes Execution Time Decomposition (AMD)

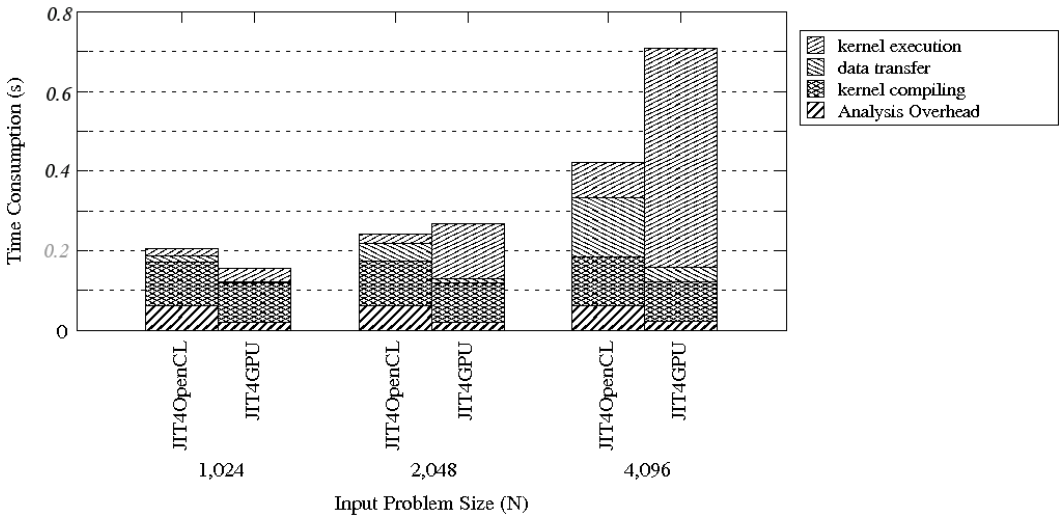

Figure 6.24: Mandelbrot Execution Time Decomposition (AMD)

ing/data transfer ratio, thus the data transfer cost occupies a large portion of the computing time; applications such as CP, N-Body and Mandelbrot, require much less data transfer.

The current version of jit4OpenCL cannot handle small inputs efficiently due to the large overhead. When the input problem size is small, most of the execution time is used for compiling and data transfers rather than actual computation. Thus, jit4OpenCL should only be used to process large inputs. Also, jit4OpenCL spends most of the time on kernel computing for a computing-intensive applications such as N-Body simulation (time complexity $O(n^4)$), and this can marginalize overhead.

Our evaluation result shows that the automatically generated kernel is efficient in some benchmark tests. On the nVidia GPU platform, most of the time is used in overhead rather than actual computation. The time consumption for CP needs a constant time (roughly 1.28 seconds) for overhead running, while the kernel execution took less than 1 second at most (Figure 6.16); Mandelbrot requires around 1.13 seconds for compiling and analysis, as a contrast, the kernel only needs less than 0.1 seconds to finish computing (Figure 6.19); Blackscholes also needs more than 1.3 seconds for analysis and compiling, adding another 1.05 seconds for data transfer, while the computing time is only 0.47 seconds at most (Figure 6.18). Those results suggest that the overhead reduction is of highest priority in the following development of jit4OpenCL.

The only exception is the N-Body simulation, which is the most computing-intensive application used in our evaluation. However it still has more than 1 second overhead (Figure 6.17). We expect that a tuning on the analysis overhead will boost jit4OpenCL overall performance.

OpenCL Kernel Compiling Time Cost

One of the greatest performance obstacle is the compilation time used for calling the external OpenCL compiler to generate GPU kernel binary code. For instance, nVidia NVCC OpenCL compiler typically takes 0.4 second to generate a binary for nVidia platform. This is a heavy delay we cannot eliminate in jit4OpenCL, as it is only related to target platform SDK implementation.

We also discovered that the time used for compiling OpenCL kernel into target platform code varies drastically on different platforms, even on the same platform, the time required still fluctuates on AMD platform. The following table shows the average time used for invoking just-in-time compiler to generate OpenCL binary kernels on nVidia and AMD platforms:

Benchmark	nVidia NVCC	AMD Stream
Matrix-Mult	0.403	0.125
CP	0.416	0.183
N-Body	0.414	0.074
Blackscholes	0.417	0.246
Mandelbrot	0.399	0.112

NVCC compiler needs around 0.4 second to compile each input kernel, which prevents jit4OpenCL from gaining further performance speedup; the AMD Stream compiler is around 3 times faster. In some of the tested benchmarks the cost of just-in-time compiling generated kernel occupies 30% of the execution time.

It is useful to point out that the time used for compiling target platform code includes the time needed for common optimizations such as dead code elimination, loop unrolling, common subexpression elimination, *etc*, which greatly enhances GPU kernel performance.

Data Transfer

In an heterogeneous computing platform, data transfer between a host and a computing device is a necessary process to prepare data for computing on the device. The cost of data transfer can be high. This cost is related to the transfer bandwidth between CPU memory and GPU memory, and to the size of input data. In general, the higher the bandwidth, and the higher the ratio of computing/transfer, the lower the overhead portion will result. For instance, in Figure 6.16 the CP application only requires a very small computing/transfer ratio, thus resulting in very low transfer overhead.

The data transfer speed is dependent on the implementation of the target platform OpenCL implementation as jit4OpenCL invokes OpenCL APIs for data transfer. Our evaluation observed that jit4OpenCL requires a longer time than jit4GPU for data transfer between CPU memory and GPU memory on AMD machine. On nVidia machine the transfer speed is around 1.6GB/s, which is 800MB/s slower than the CUDA implementation

on the same machine. The results indicates that current OpenCL implementation on both nVidia and AMD platforms provide slow communication between CPU memory and GPU memory. We expect future OpenCL implementations can diminish or eliminate the speed gap, which will lead to a performance gain in jit4OpenCL.

6.3 Concluding Remarks

We measured the performance of several benchmark applications on jit4OpenCL using two different machines. In some cases overhead accounts for most of the execution time. When it comes to the execution of computing-intensive applications, jit4OpenCL can still outperform traditional CPU code by up to 91 times. However, we did observe a serious performance degradation when compiling Matrix-Multiplication, this suggests that jit4OpenCL-compiled program performance may vary on AMD architecture. User discretion is advised.

The experimental evaluation also indicates that the jit4OpenCL execution has a heavy overhead, including fixed-cost compiling time that is determined by the program structure, the jit4OpenCL analysis requirement and the target executing platform OpenCL compiler that shipped with the corresponding SDK. In most cases, the biggest parts of the overhead are the analysis in jit4OpenCL inherited from jit4GPU, and the time consumption used in invoking OpenCL compiler for the target platform GPU binary code generation.

Jit4OpenCL cannot generate efficient code for all benchmarks, due to the restrictions on array access analysis and the GPU architecture. Only those applications that adapt stream processing programming paradigm are capable of exploiting the parallelism on GPU, and only those that have regular array access patterns can be compiled efficiently by jit4OpenCL.

Chapter 7

Related Work

7.1 Other Compilations of Scripting Languages

There has already some tool kits for compiling scripting language into either another source language or into executable. Pros and cons of different techniques used for compiling scripting languages are discussed in the rest of this section.

7.1.1 Dynamic Scripting Language Embedders

Dynamic script embedders are programs that wrap script and interpreter into executables. It is worthy to talk about embedders here because they also create executables, but generally speaking they are not compilers. Py2exe [27], Py2app [26], cx_Freeze [7], Squeeze [32] and exemaker [8] are tools for wrapping Python programs into executables for Windws(Py2exe), Mac OSX(Py2app) or cross-platform environment(cx_Freeze, Squeeze, exemaker). Those tools creates Python executables that do not require an Python interpreter at runtime. The wrapping procedure is as follows: First, conduct a dependency analysis to get the dependent script files, Python libraries and interpreters; second, archive all necessary files and libraries into a standalone executable distro. When user invokes the program, the executable calls Python interpreter to perform an JIT compile on the archived python bytecode/script. This method for translating scripting language into executable format suffers from slow execution(JIT compilation is required) and big file size. However, embedding a runtime machine is a quick and simple way for generating standalone distros for dynamic typed scripting language.

7.1.2 Dynamic Scripting Language Compilers

Dynamic scripting language compilers often follow the traditional compiler process flowchart: they typically converts the source file into abstract syntax tree, then into intermediate representation, performs simple data and type analysis for optimization. For instance, Rhino JavaScript compiler translates JavaScript file into Java bytecode [10], BaCon [2],

XBLite [37], *etc* translates Basic language into executables.

Another approach is to use code templates for code generation, and is part of the template meta-programming. For instance, BCX [5] convert BASIC scripts to C source code using code templates. Lliasov *et al.* used a special code template to build a simple but highly-portable Just-In-Time compiler [14]. They use machine-code templates to efficiently generate target code and use different templates for retargeting platforms. Their experiments showed that they brought the JIT compiler performance to a level comparable with traditional static compilers, but requires considerably more memory (often 5 to 20 times more) than other JIT compilers.

7.2 Loop Access Analysis

Paek *et al.* proposed the Linear Memory Access Descriptor(LMAD) in [22] [23]. It is a simplified linear model that describes the memory locations accessed by an array in a loop nest. LMAD is used to analyze array access behaviors. For instance, to identify coalescing accesses, interleaving accesses, contiguous accesses, and is capable of privatizing arrays for parallel shared memory machines. Garg *et al.* uses a subset of LMAD, Restricted Constant Strided Linear Memory Access Descriptor(RCSLMAD) to discover and identify memory locations of accessed array elements[12]. In our work we utilize RCSLMAD for identifying accessed memory locations of thread groups and threads inside the group to do the array privatization on on-chip shared memory.

Another powerful access analysis tool that is widely used in compiler analysis is the polyhedral model. The polyhedral model was first introduced for systolic array synthesis, and this representation is called Systems of Affine Recurrence Equations (SAREs) over polyhedral domains [28]. Feautrier later proved that exact data flow analysis on a certain types of loops can obtain an isomorphic form (dynamic single assignment) of the SARE over polyhedral domain [9]. Analyzing dynamic single assignments can lead to discovery of potential parallelism. The processing of polyhedral model is divided into three parts: static dependency analysis on input program; polyhedral transformation; loop code generation. However, the domain of transformation is often large enough to take a good amount of time of search to get the exact transformation that will lead to improved performance. Though more accurate, due to the time constraint, at present, polyhedral model cannot be used when time constraint is a main concern.

Several polyhedral model implementation has been published. Especially in GCC version 4.4.0, the polyhedral model analysis module called Graphite [25] has been included for high-level memory optimizations for loop nests. When GCC obtain the program GIMPLE, SSA and CFG representation, it then undergo the Graphite pass, which constructs a polyhedral model of the program (GPOLY). Graphite performs on the input optimizations

or analysis like cost-modeling, scheduling execution order by data access pattern. Experiments revealed that adapting Graphite may increase generated file size (because of complex domain decomposition), and lead to longer compiling time (because of model computation complexity). But the generated code shows significant speed up on benchmark.

7.3 Optimizing GPU Programs

Ryoo *et al.* discussed the possible optimization space for CUDA programs on nVidia's Tesla Architecture [30] [29] [31]. They argued that to achieve good performance using CUDA, programmers have to strike the right balance between the number of simultaneously active threads, and thread resource usage, including register usage, shared memory usage, and the global memory bandwidth usage. One of the important discovery they found in their benchmark is that without overflowing register file, increase the amount of simultaneously active threads will lead to better performance. They also studied several application cases and discussed about their portability to parallel platforms, along with in-depth analysis such as performance bottlenecks, and some general guideline for implementation. In our work, the automatic grid configuration generation module follows the guidelines described in [29], and we regard the optimization principles a good source for reference.

Unlike Ryoo *et al.*, Volkov *et al.* showed that dense linear algebra computation can achieve up to 40% of performance gain compared to Ryoo's result by an in-conventional way of implementation on some benchmarks [36]. In their implementation, they carefully analyzed the access latency to each different level of memory and the PTX code (the intermediate byte code format for CUDA programs) generated by NVCC compiler, to devise an implementation pattern that unrolls loops aggressively and interleaves memory transferring with computing. They argued that fewer active thread will lead to more register file space available for each thread, and less thread-scheduling overhead. Their tweaking is most suitable for manual optimization, however, their method is hard to be adapted to compiler design due to the involvement of delicate analysis on the cost of accessing the memory architecture, and different GPU may result in different performance.

7.4 Compiling for Hybrid Systems

Garg *et al.* created a Python compiling framework that translates annotated Python program into both OpenMP and AMD CAL language [12]. When executing on a hybrid execution environment, the compiler framework's backend jit4GPU decides whether to execute compiled OpenMP code or AMD CAL code, depending on the program behaviors and available hardware components. Compared with our OpenCL generation framework, Garg's work has less portability because the GPU acceleration is specifically designed for the AMD

architecture, but its advantage is that jit4GPU can generate optimized CAL assembly directly. Redirecting jit4GPU into generating OpenCL code is a trade off of portability and runtime performance, and it makes the real performance dependent on the performance of OpenCL JIT compiler provided by hardware manufacturers.

Baskaran *et al.* developed a C-to-CUDA code generation framework for affine programs [4]. They used PLUTO(PLUTO: A polyhedral automatic parallelizer and locality optimizer for multi-cores) [24] for polyhedral model analysis and CLooG [6] for multi-level tiled code generation. In spirit they used a very similar approach to solve on-chip/off-chip data transfer and tiling problems: by dividing the multi-dimensional loop domain into contiguous sub-domains. However, they use polyhedral model for array analysis rather than LMAD. Another difference between their system and ours is that theirs is an static compiler that involves no runtime compiling, while unPython must cope with jit4GPU to resolve runtime dynamic type checking.

Lee *et al.* developed an automatic OpenMP-to-CUDA compiler framework, resulting around 14x to 25x speedup in some systems [15]. This framework can translate irregular (unaffined) OpenMP programs into CUDA program, however, it only does straightforward language-to-language transformation and cannot utilize the on-chip shared memory to optimize data access, which results severe performance degradation compared to other CUDA generation frameworks.

Liu *et al.* came up with an optimization framework, G-ADAPT (GPU adaptive optimization framework), that takes as input an unoptimized CUDA source code and outputs a parameter-optimized one [17]. They use heuristic-based empirical search to traverse the optimization space to automatically find the (near-)optimal program configuration. In the searching, a database containing past program execution information is used to help heuristically generate next version of output. Each output is compiled and executed for immediate benchmark feedback. Their works could be used on further optimization of other ahead-of-time CUDA generation frameworks but is not suitable for a just-in-time system like ours due to time constraints.

Zhu *et al.* designed a data classification and distribution algorithm based upon LMAD representation, they used it to translate programs manually for distributed shared memory (DSM) systems [39]. In their analysis, data is categorized into three classes via LMAD analysis: private, distributed and shared. Private data consists of variables only accessed by a single processor; distributed data consists of variables that might be used for multiple processors; shared data consists of variables their accessing pattern is complex or just statically unknown. The framework divides the iteration space to assign each sub-iteration to one processor, message passing communication is used to distributed data and shared data across processors. Their works involve inter-processor communication, which is not suitable

for nVidia GPU architecture.

Multicore-CUDA (MCUDA) [33], developed by Stratton *et al.* translates CUDA programs to the conventional shared memory CPU architecture. The benefit is that the programmers can adapt CUDA programming model to write conventional systems. MCUDA translates CUDA kernel to sequential program but requires a framework to manipulate threading, transform shared memory declarations at runtime. Compared to our approach, it is an opposite one that translate GPU code back to CPU code.

CUDA-Lite [35] is a higher CUDA-like programming model developed by Ueng *et al.*. It aims at reducing the burden upon programmers and increasing the productivity by introducing an automated memory coalescing tool into the framework. With annotations in CUDA kernel, CUDA-Lite is capable of generating memory loading/storing code automatically, sometimes it is capable of generating coalesced memory accesses. The reason CUDA-Lite requires annotations is that it does not utilize analysis on the behavior of memory accesses, and our OpenCL framework also involves automatic memory access generation, but it can acquire enough information to generate memory access code via the LMAD analysis.

Another similar CUDA tweak that requires additional annotation is the hiCUDA (Hi-level directive-based language for CUDA program) developed by Han *et al.* [13]. It is a source-to-source compiler that translates hiCUDA programs into CUDA programs. In hiCUDA model, the programmer writes conventional programs with additional hiCUDA directives, then the compiler transforms it into CUDA host code and kernel code according to the program behaviors and the directives, which includes identifying data transferring region, loop partitioning, barrier fencing, etc. The experiment shows that there is no performance loss after a hiCUDA program translation. Generally speaking the model Han *et al.* created requires the programmers to have knowledge on GPGPU computing, and the programmers still need to design program in the view of GPU, it is not as adaptable as our framework that does not require programmers prerequisites on heterogeneous computing systems.

Chapter 8

Conclusions

This thesis describes jit4OpenCL, a retargetting of jit4GPU, which is a Just-in-Time compiler developed by Garg and Amaral for runtime execution on a heterogeneous architecture consisting of AMD GPUs. Jit4OpenCL generates code for the industry-standard OpenCL language, enabling the framework to execute the same source program on any platform that supports OpenCL.

In order to efficiently exploit the potential of the new memory hierarchy model in OpenCL, we proposed an adaptation of the LMAD analysis for scattered memory accesses. Our analysis tiles the program's RCSLMAD accesses by splitting the program's iteration domain, and then reusing the original RCSLMAD function to describe each tile. The analysis can break down array accesses while maintaining program consistency, making further transformations possible. This analysis was originally developed to generate optimized kernel code for the nVidia's Tesla GPU architecture, but it also works for similar SIMD platforms.

Jit4OpenCL adapts this analysis for OpenCL code generation. Jit4OpenCL blocks parallel iterations into smaller parallel iteration blocks and converts each into a thread group. Each iteration in each group is converted into a parallel thread. Jit4OpenCL uses the analysis as the essential way to identify the memory locations accessed by each thread and thread group.

Depending on the memory reference patterns, jit4OpenCL may compromise the performance to maintain program correctness by avoiding optimizations that potentially break the program's consistency. For example, when multiple array access references overlap in their regions, shared memory space is not used for the sake of data consistency.

The experimental evaluation indicates that jit4OpenCL can improve the performance significantly in some applications on either AMD or nVidia platform. The execution performance is determined by the following factors:

- the compiling overhead that is determined by the complexity of the program code.

- the data-transfer overhead that relates to the bandwidth between host and devices, and the size of data.
- the computation/data transfer ratio of the input application. High ratios often yields high performance.
- the array access pattern in input application. Irregular access pattern may cause jit4OpenCL to generate inefficient code or to fail to produce GPU code.

8.1 Future Work

There is still significant room for improving the framework presented in this thesis. The expensive overhead at runtime is a main obstacle for further performance improvement, thus further optimization should focus on improving the time used on JIT compiling.

Our experiments show that the kernel execution speed is not optimal – it is because either:

- the grid configuration is not optimal
- the auto-generated algorithm is not optimal

Current implementation of jit4OpenCL uses a fixed, square grid configuration that cannot be optimal in most cases. However, grid configuration is an important parameter that will affect the execution performance. A better solution is to adapt irregular grid configurations that better suit the input application better. This change requires more in-depth analysis we do at the JIT compiling time, and asks for more information gathering from the input application and the GPU architecture. Several tuning approaches can be used for this job, including utilizing machine-learning to reach an optimal solution, or adapt an auto-tuning framework, either method must focus on the grid dimension configuration.

Current implementation of jit4OpenCL regards each run as independent to one another. However, a library method can be called several times in its software lifespan, some of the analysis result, such as RCSLMAD analysis, kernel code generation, can be re-used in the following executions. This will greatly reduce the overhead.

A large portion of compilation overhead is the invoking of OpenCL compiler to generate target GPU binary code. For example, to generate OpenCL code for nVidia system, around 0.4 second must be consumed in order to get the kernel binary by invoking the OpenCL compiler. According to the specification of OpenCL 1.0 [19], each OpenCL platform is capable of providing APIs for saving or loading a kernel binary. However, different kernels may be generated for different input problem size of the same application, Thus, jit4OpenCL needs additional information when storing the binary kernel. Utilizing those APIs to avoid repetitively generating kernel binaries will accelerate the performance.

Jit4OpenCL is not capable of handling RCSLMADs that their memory regions overlap. When jit4OpenCL detects that the memory regions of two or more RCSLMADs overlap, and at least one of them contains a writing operation, jit4OpenCL will avoid utilizing shared memory optimization for those RCSLMAD memory accesses in order to maintain consistency. This is a coarse analysis because even when the regions of two RCSLMADs overlap, it is still possible that the two RCSLMADs do not reference the same memory element. When it happens jit4OpenCL loses an opportunity to perform memory access optimization. Also, in current implementation of jit4OpenCL we are using a one-to-one mapping rule for mapping global memory elements to shared memory space. This is sufficient in handling an application whose RCSLMADs do not overlap. However, when dealing with overlapping RCSLMADs, jit4OpenCL can no longer adapt current analysis method. A refined analysis based on the characteristics of RCSLMADs may be capable of identifying such situation.

Bibliography

[1] Python/C API Reference Manual (2009-09-30). `http://www.python.org/doc/2.5/api/api.html`.

[2] BaCon: Basic to C converter. `http://www.basic-converter.org/`.

[3] Utpal Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Boston, 1997.

[4] Muthu Manikandan Baskaran, J.(Ram) Ramanujam, and P. (Saday)Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction (CC)*, pages 244–263, Paphos, Cyprus, 2010.

[5] Bcx(basic to c converter). `http://bcx-basic.sourceforge.net/`.

[6] CLooG. `http://www.cloog.org/`.

[7] cxFreeze. `http://cx-freeze.sourceforge.net/`.

[8] exemaker. `http://effbot.org/zone/exemaker.htm`.

[9] Paul Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988.

[10] M. Foundation. Rhino: Javascript for java, 2007.

[11] Rahul Garg. A compiler for parallel execution of numerical python programs on graphic processing units. Master's thesis, University of Alberta, 2009.

[12] Rahul Garg and José Nelson Amaral. Compiling Python to a hybrid execution environment. In *Workshop on General-Purpose Computation on Graphic Processing Units(GPGPU)*, pages 19–30, 2010.

[13] Tianyi David Han and Tarek Abdelrahman. hi CUDA: a high-level directive-based language for GPU programming. In *Workshop on General-Purpose Computation on Graphic Processing Units(GPGPU)*, pages 52–61, Washington, DC, USA, 2009.

[14] Alex Iliasov. Templates-based portable just-in-time compiler. *SIGPLAN Notices*, 38(8):37–43, 2003.

[15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Principles and Practice of Parallel Programming(PPoPP)*, pages 101–110, Raleigh, NC, USA, 2009.

[16] Eric Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[17] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *International Parallel and Distributed Processing Symposium(IPDPS)*, pages 1–10, Rome, Italy, 2009.

[18] Aaftab Munshi. *The OpenCL specification version 1.0*. Khronos OpenCL Working Group, 2009.

[19] Aaftab Munshi. *The OpenCL specification version 1.1*. Khronos OpenCL Working Group, 2010.

[20] Scientific Computing Tools For Python – NumPy. `http://numpy.scipy.org/`.

[21] Travis E. Oliphant. *Guide to NumPy*. NumPy Developers, 2006.

[22] Yunheung Paek, Jay Hoeflinger, and David A. Padua. Simplification of array access patterns for compiler optimizations. In *Programming Language Design and Implementation(PLDI)*, pages 60–71, Montreal, QC, Canada, 1998.

[23] Yunheung Paek, Jay Hoeflinger, and David A. Padua. Efficient and precise array access analysis. *Transactions on Programming Languages and Systems(TOPLAS)*, 24(1):65–109, 2002.

[24] PLUTO compiler. `http://pluto-compiler.sourceforge.net/`.

[25] Sebastian Pop, Albert Cohen, Cedric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *GCC Developers' Summit*, pages 179–198, Ottawa, ON, Canada, 2006.

[26] Py2app. `http://svn.pythonmac.org/py2app/py2app/trunk/doc/index.html`.

[27] Py2exe. `http://www.py2exe.org/`.

[28] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *International Symposium on Computer Architecture(ISCA)*, pages 208–214, Ann Arbor, Mi, USA, 1984.

[29] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance

evaluation of a multithreaded GPU using CUDA. In *Principles and Practice of Parallel Programming(PPoPP)*, pages 73–82, Salt Lake City, UT, USA, 2008.

[30] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, and Wen-mei W. Hwu. Program optimization study on a 128-core GPU. In *Workshop on General-Purpose Computation on Graphic Processing Units(GPGPU)*, Boston, MA, USA, 2007.

[31] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen-mei W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10):1389–1401, 2008.

[32] Squeeze. `http://bcx-basic.sourceforge.net/`.

[33] John A. Stratton, Sam S. Stone, and Wen-mei Hwu. MCUDA: An efficient implementation of cuda kernels on multi-cores. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, Edmonton, AB, Canada, 2008.

[34] Damien Triolet. *nVidia CUDA, preview.* nVidia, 2007.

[35] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 1–15, Edmonton, AB, Canada, 2008.

[36] Vasily Volkov and James Demmel. Benchmarking GPUs to tune dense linear algebra. In *Conference on Supercomputing(SC)*, pages 1–11, Austin, TX, USA, 2008.

[37] XBLite compiler. `http://www.xblite.com/`.

[38] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par*, pages 887–899, Delft, Netherland, 2009.

[39] Jiajing Zhu, Jay Hoeflinger, and David A. Padua. Compiling for a hybrid programming model using the LMAD representation. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 321–335, Cumberland Falls, KY, USA, 2001.