Enable OpenCL Compiler with Open64 Infrastructures

Yu-Te Lin*, Shao-Chung Wang*, Wen-Li Shih*, Brian Kun-Yuan Hsieh† and Jenq-Kuen Lee*
*Department of Computer Science,
National Tsing-Hua University, Hsinchu, Taiwan
†Cloud Computing Research Center for Mobile Applications,
Industrial Technology Research Institute, Hsinchu, Taiwan

***Abstract*—As microprocessors evolve into heterogeneous architectures with multi-cores of MPUs and GPUs, programming model supports become important for programming such architectures. To address this issue, OpenCL is proposed. Currently, most of OpenCL implementations take LLVM as their infrastructures. This presents an opportunity to demonstrate whether OpenCL can be effectively implemented on other compiler infrastructures. For example, Open64, which is another open source compiler and known to generate efficient codes for microprocessors, can contribute further to performance improvements and enhancing the adoption of heterogeneous computing based on OpenCL. In this paper, we describe the flow to enable an OpenCL compiler based on Open64 infrastructures for ATI GPUs. Our work includes the extension of the front-end parser for OpenCL, the generation of high-level intermediate representations with OpenCL linguistics, performing high-level optimization, and finally applying OpenCL specific optimization for code generations. Preliminary experimental results show that our compiler based on Open64 is able to generate efficient codes for OpenCL programs.**

I. Introduction

In recent years, heterogeneous multi-core architectures are becoming mainstream of architecture designs. Equipped with many specialized processing units, heterogeneous multi-core architectures are proposed to deliver high performance on specific applications. Many processor vendors have designed their heterogeneous multi-core processors such as IBM Cell [1], [2], Intel Sandy Bridge [3], and AMD Fusion APU [4]. IBM Cell, equipped with one Processor Element (PPE) and eight Synergistic Processor Elements (SPE), is adopted by the well-known game console SONY PlayStation 3. The PPE takes charge of general purpose tasks and the eight SPEs provide high floating-point throughput with vector operations. In another example of heterogeneous multi-core processors, Intel Sandy Bridge and AMD Fusion APU consist of general purpose processors and graphic processing units to boost the performance of multimedia applications.

However, most of traditional programming languages are designed for homogeneous computing. Simply adopting these languages into heterogeneous development environment results in adding various compiler directives into programming models, which raise the difficulty of application developments. Therefore, several programming models are proposed to use these heterogeneous computing resources efficiently. Cell Broadband Engine (CBE) software development kit, released by IBM for Cell, introduces two compilers for PPE and SPE respectively and a runtime library for communication between PPE and SPE. There are also many programming models proposed for heterogeneous computing, either in personal computers or embedded environments such as Streaming RPC [5], CTM [8], BrookGPU [9], or HMPP [10]. Although these tools improve development efficiency in heterogeneous environments, application developers still have to learn various tools for different development environments. To address this issue, OpenCL was proposed as an attempt for a standard for programming across heterogeneous multi-core environments.

Currently, there are many OpenCL software developer tools available such as Apple Xcode [16], Intel OpenCL SDK [18], and AMD APP SDK [21]. Most of the OpenCL compiler implementations take LLVM [14] [15] as their compiler infrastructures, which present an interesting question if OpenCL can be effectively implemented on other compiler infrastructures. This will have the opportunity to bring more academic innovations to help the OpenCL efforts. In this paper, we describe the development of supporting OpenCL based on Open64 infrastructures. We extend the high-level intermediate representations of Open64 to have OpenCL linguistics and propose an extended two-stack method for parsing OpenCL vector operations in the front-end. In the back-end, we also extend the low-level intermediate representations of Open64 for matching the hardware properties for OpenCL. Besides, the register allocation method and the optimization for private memory load are also discussed. Preliminary experimental results show that our Open64 based OpenCL compiler has high reliability and is able to generate efficient codes for OpenCL programs.

The remainder of this paper is organized as follows. Section 2 briefly introduces OpenCL and the Open64 compiler infrastructure. Section 3 presents implementation details. Next, Section 4 gives the experimental results. Finally, Section 5 concludes this paper.

This work was supported in part by National Science Council (under grant no. NSC-100-2219-E-007-007 and NSC-100-2220-E-007-003) and Ministry of Economic Affairs (under grant no. MOEA-99-EC-17-A-01-S1-034) in Taiwan.

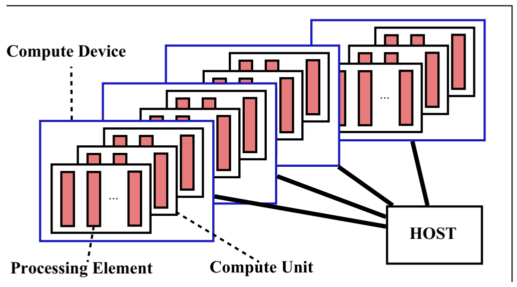


Fig. 1. OpenCL Platform Model [19]

II. Background

A. OpenCL Language

OpenCL (Open Computing Language) is a programming framework for developing applications on heterogeneous platforms consisting of CPUs, GPUs, DSPs and other processors. It defines a C99 extension programming language for kernel programming. OpenCL was initially developed by Apple Inc. After collaborating with other technical teams, like AMD and NVIDIA, Apple submitted the first proposal to Khronos Group [11]. In 2008, the OpenCL specification was approved by Khronos Group and released. Now OpenCL is an open and royalty-free programming standard.

OpenCL platform is composed of a host device and many target compute devices as shown in Fig. 1. The host device dispatch jobs and transfer data to certain compute devices for computation. The target compute devices might be any high performance processors such as GPUs or DSPs. In the viewpoint of OpenCL platforms, a compute device could be further divided into many compute units; each compute unit can have many processing elements to have many multithreadings.

OpenCL defines an explicit memory hierarchy. Fig. 2 shows the memory model. Each processing element has its private memory and shares data with other processing elements via local memory. The global/constant memory can be accessed by every compute devices. Corresponding to the memory model, OpenCL language provides many address qualifiers (e.g., __global, __local, __constant, and __private) for users to specify data locations.

OpenCL also supports vector data types and vector operations. Vector data types, such as float4, consist of multiple homogeneous primitive data types in an instance. The vector component permutations and replications are also supported. Listing 1 shows an OpenCL vector permutation and replication example. In the first statement, a float4 vector A is declared and initialized. The internal four components of A are 1.0f, 2.0f, 3.0f, and 4.0f. The second statement performs a vector permutation, which makes the values of vector B be 2.0f, 4.0f, 1.0f, and 3.0f. The vector replication in the third statement will assign 1.0f, 2.0f, 2.0f, and 3.0f to vector C. Furthermore, most of common arithmetic operations and math functions in OpenCL can be applied on either scalar data types or vector data types, which imply a function overloading is required.

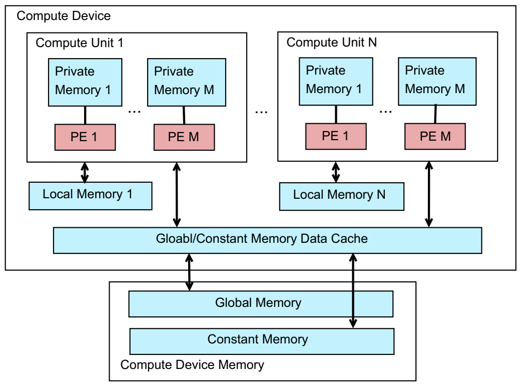


Fig. 2. OpenCL Memory Hierachy [19]

All of these linguistic features of OpenCL make the support of compiler important.

```
float4 A = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 B = A.ywxz;
float4 C = A.xyyz;
```

Listing 1. OpenCL vector permutation and replication

B. Open64 Infrastructures

Open64 [12] is an open-source compiler infrastructure derived from SGI MIPSPro compiler. It was released in 2000. Currently the official branch is maintained by Hewlett Packard and Delaware University; the latest release version is 4.2.4. Many advanced compiler optimizations are implemented in Open64 such as interprocedural analysis (IPA), loop nest optimizer (LNO), and global optimizer (WOPT), which founds the capability of Open64 to generate efficient code. As Open64 is a well-stabilizing infrastructure, many industry and academic research employ Open64 as the foundation for new target porting works. ORC [23] project adopts Open64 to generate VLIW code for the Itanium processor, AMD has a x86 compiler suite based on Open64, and NVIDIA also use Open64 as its CUDA compiler [7], [22]. In the embedded system site, Open64 is used for VLIW DSP with distributed register files [17], [20].

Open64 is capable to support many common programming languages such as C, C++, and Fortran; it also supports OpenMP [13], a shared memory programming interface. Open64 takes GNU GCC [6] as its front-end and translates the intermediate representation (IR) generated by GCC to its own format. Therefore, to support new language features in Open64 like this paper does, the GCC based front-end of Open64 have to be modified accordingly.

The intermediate representation of Open64, named WHIRL, can be divided into five levels: very high (VH), high (H), mid (M), low (L) and very low (VL). During the compilation processes, different optimizations are performed on different WHIRL levels and the WHIRL are lowered to the next level when needed. The lower the level is, the target dependent the WHIRL becomes. Then the VL WHIRL is translated into

TABLE I
COMMON NOUNS OF OPEN64 COMPILER

Item	*Description*
WHIRL	High level intermediate representation of Open64 compiler.
CGIR	Low level intermediate representation of Open64 compiler.
BB	Basic block, a portion of code fragment which has only one entry point and only one exit point.
OP	Machine-level operation, corresponding to hardware instruction.
TN	Temporary Name, can be mapped to hardware register or a constant.

CGIR form — another kind of intermediate representation of Open64 that is much closer to hardware instruction and can be used to describe hardware properties such as registers and instruction latencies. Some common nouns of Open64 used in this paper are listed in Table I.

III. OPENCL SUPPORT IN OPEN64

In this section we will describe how to support OpenCL in Open64. As developing a compiler, we discuss the supporting issues in three aspects: the compiler front-end, the compiler back-end, and optimizations. The front-end issues can be target independent but the back-end issues are mostly target dependent. For the back-end, we choose ATI GPU as our target and discuss the related issues.

A. Front-End

1) OpenCL Address Qualifiers: Since OpenCL is a C extension language which has many memory address qualifiers that can be used by programmers to access different memory spaces, the first thing we have to do is to make the front-end parser of Open64 to be able to parse these address qualifiers. We add the address qualifiers, __global, __local, __constant, and __kernel, into the front-end parser to be reserved words and then extend the data structure of the symbol table in the parser so that the parser can record required information in the symbol table while encountering the qualifiers. The extended symbol table will provide sufficient information for data layout at later compiler phases.

2) OpenCL Vector Parsing: The most significant difference between OpenCL and the traditional C language is that OpenCL has built-in vector data types. Each vector data type can have at least two or more internal components depending on the number users specify. Besides, the common binary operations in C language can be applied on each component separately. Vector permutation and replication are also supported. All of these properties that OpcnCL vector have make the parsing work of a compiler much difficult. One simple method for vector parsing is to translate OpenCL vector operations to be operations simulated by C or C++ languages and then use the native compiler to compile it. However, after the translation we may lose some OpenCL vector properties as well as some optimization possibilities. In order to preserve OpenCL vector properties, we add the vector types as the compiler built-in data types and then support the various vector operations in the front-end. As Open64 takes GCC as its front-end and GCC construct a tree intermediate representation in its front-end, we have to add built-in vector tree node in the front-end of GCC. The vector tree nodes we add have the similar properties like type define union and struct as Listing 2 shows so that it can support the vector component, X, Y, Z, and W.

```
typedef float __ocl_float4
              __attribute((vector_size(16)));
typedef union {
  struct {float x,y,z,w};
  __ocl_float4 _ocl_vec;
} float4;
```

Listing 2. OpenCL vector node

After the constructions for vector data types are ready, we can focus on parsing various vector operations that OpenCL has. The vector operations include initialization, permutation, replication, and common arithmetic operations. For vector initialization, one can easily refer to the grammar rules for scalar array initialization but add one more rule to allow the initial values to be surrounded by parentheses. To support the various vector operations, we extend the original two-stack method for parsing arithmetic expressions in the front-end of Open64. The two stacks where one is the OP stack for storing operator tokens and another one is the EXPR stack for storing variable tokens or computed expressions. As there are two stacks available, the front-end parser is able to parse infix arithmetic expressions.

The internal implementation of EXPR stack is an one-dimensional array. To support OpenCL vector operations, we extend it to be a two-dimensional array in which the extended second dimension is used for each vector component. Our parsing algorithm is shown in Algorithm 1 where the considerations for parentheses are omitted temporarily for the ease of understanding. Also we use .[number] to represent component selection for vector types. For example, tmp.[0] means the first component of variable tmp, tmp.x. While encountering a vector variable token, we first split it into separate vector components and then push them into the extended two-dimensional stack. If the input token is an arithmetic operator and the precedence is greater than the precedence of the OP on the top of the OP stack, we push the operator token into the OP stack. If the input operator has precedence less than or equal to the precedence of the OP on the top of the OP stack, we pop two elements from EXPR stack and pop one operator from OP stack to operate on the two elements from EXPR stack. After doing the operation, we push the computed new expression into EXPR stack and also push the input operator token into OP stack. In the process for popping elements, all of the elements in the second dimension that are at the same depth have to be popped together and to be applied on the same operator. If there are no input tokens and the OP stack is not empty, we pop one operator from OP stack and pop two elements from EXPR stack to do operation and then push the computed new expression into EXPR stack. This process goes until the OP stack is empty and then the only element in the

EXPR stack is the final expression we need.

Algorithm 1 Infix Expression Parsing for OpenCL Vector

```
1: $sp \Leftarrow 0$
2: while $token \Leftarrow lex\_one\_token()$ && $token \neq NULL$ do
3:   if $token\ is\ an\ OpenCL\ vector$ then
4:     $Split\ toekn\ into\ separate\ vector\ components.$
5:     $comp \Leftarrow 0$
6:     $sp++$
7:     while $comp < num\_of\_components(token)$ do
8:       $num\_stack[sp][comp] \Leftarrow token.[comp]$
9:       $comp++$
10:    end while
11:  else if $token\ is\ an\ operator$ then
12:    $\otimes \Leftarrow token$
13:    $\odot \Leftarrow op\_stack[sp]$
14:    if $Precedence(\odot) >= Precedence(\otimes)\ \|\ sp == 0$ then
15:      $comp \Leftarrow 0$
16:      while $num\_stack[sp][comp] \neq NULL$ do
17:        $num\_stack[sp-1][comp] \Leftarrow num\_stack[sp-1][comp] \odot num\_stack[sp][comp]$
18:        $comp++$
19:      end while
20:      $op\_stack[sp-1] \Leftarrow \otimes$
21:      $sp--$
22:    else
23:      $sp++$
24:      $op\_stack[sp] \Leftarrow \otimes$
25:    end if
26:  end if
27: end while
28: while $sp \neq 0$ do
29:   $\otimes \Leftarrow op\_stack[sp]$
30:   $comp \Leftarrow 0$
31:   while $num\_stack[sp][comp] \neq NULL$ do
32:     $num\_stack[sp-1][comp] \Leftarrow num\_stack[sp-1][comp] \otimes num\_stack[sp][comp]$
33:     $comp++$
34:   end while
35:   $sp--$
36: end while
```

Fig. 3 demonstrates a running example for parsing vector operation, A.xyz * B.yzw + C.xxx. At the beginning, the two stacks are all empty as shown in Fig. 3(a) and then A.xyz will be split and pushed into the corresponding position in EXPR stack as shown in Fig. 3(b). In Fig. 3(c) the multiplication operator will be pushed into OP stack and then B.yzw will be split and pushed into EXPR stack. While encountering an addition operator as shown in Fig. 3(d), since the precedence of addition is less than the precedence of multiplication, we pop the multiplication operator from OP stack and also pop two elements from EXPR to do multiplication. After apply the multiplication, we push the new expression into EXPR stack and push the addition token into OP stack. In Fig. 3(e), C.xxx will be pushed into the EXPR stack. Finally, we apply addition operation for the elements in EXPR stack as shown in Fig. 3(f). The final expression can be obtained from the bottom of the EXPR stack.

B. Back-End

1) Struct Transformation: Besides the basic porting such as adding target instructions or final assembly code expansion phases, there are some components to be modified to support OpenCL. Most compilers perform many optimizations or transformations on their intermediate representations. One common transformation is to translate the struct access of C languages to be accesses on separate struct fields. Since we

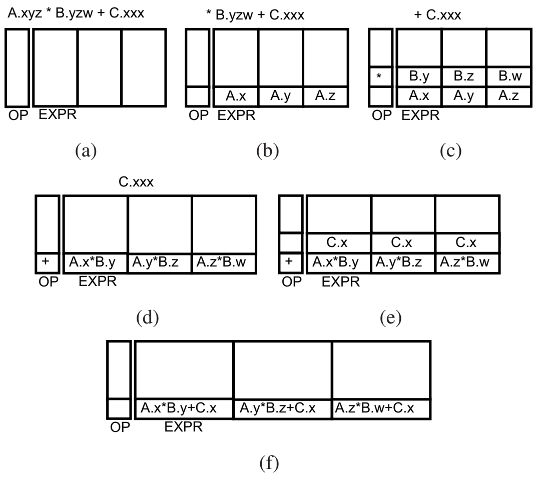


Fig. 3. Step for parsing A.xyz * B.yzw + C.xxx

adopt the similar methodology like type-defined union and struct to define an OpenCL vector. Without any modification on the compiler transformation, the vector operation will be translate to become operations on the separate vector components. This will hurt the performance. To avoid the performance degression we have to avoid this transformation while operating on vector variables.

2) Extension for CGIR Framework: The most common nouns in Open64's low level intermediate representation, CGIR, are TN and OP, which are used to map to hardware registers and instructions respectively. For most common processors, there is no problem for the representations of TNs and OPs. However, for most GPU architectures, the hardware registers can have four components internally and the instructions can also specify which register component it will be operated on. The TNs and OPs of Open64 do not have the concept of register components so that it will cause problems while generating instructions. Therefore, we modify the internal data structure for TN and OP. The TNs can have four components to be matched to hardware registers and the OPs also can select register components.

3) Register Allocation: Another important phase of compiler back-end is the register allocation phase (RA). If the target we want to support has limited number of registers, we have to use the traditional method to calculate variable live ranges and then spill the content of a register into memory if required. On the other hand, if the target architecture has a large number of registers like ATI GPU we choose, things will become different. Since ATI GPU has a lot of registers, we don't have to spill registers and we can put each whole function frame in registers. However, in order to preserve the content of one frame to not been written by another frames, we have to avoid allocating the same register for different frames. The method we used to allocate registers is shown in Algorithm 2. We iterate through each function frame, basic block (BB), and OP to allocate register to each TN. If the TN needs to be allocated register, we pick one register r from allocatable register set, α, for TN and then remove r from allocatable register set. The process goes until all the TNs in the program are allocated registers.

Algorithm 2 Register Allocation for ATI GPU

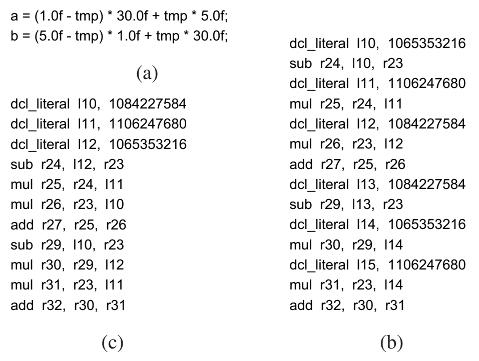
```
Require: r: register
Require: $\alpha$: allocatable register set
 1: for each $frame\ in\ OpenCL\ program$ do
 2:   for each $Basic\ Block\ in\ the\ frame$ do
 3:     for each $OP\ in\ the\ basic\ block$ do
 4:       for each $TN\ in\ the\ OP$ do
 5:         if $TN\ need\ to\ be\ allocate\ register$ then
 6:           $r \Leftarrow Pick\_one\_register(\alpha)$
 7:           $Allocate\ r\ to\ the\ TN$
 8:           $\alpha \Leftarrow \alpha \setminus r$
 9:         end if
10:       end for
11:     end for
12:   end for
13: end for
```

C. Optimizations

We describe optimizations in the infrastructures in our current design below.

1) Remove Redundant Literal Declaration: The first optimization we implemented in Open64 for ATI GPU is to reduce the redundant dcl_literal instructions which are used to declare immediate values. Since all of the ATI instructions excluding dcl_literal can not have immediate value as their operands, whenever immediate values are required to do operations, the dcl_literal instructions have to be generated to declare the immediate value in registers first. Without any optimizations, there will be a lot of dcl_literal instructions in the program and many of which are redundant. To address this issue, we can simply move each dcl_literal instructions to the beginning of the program and then check each dcl_literal pair. If the second dcl_literal declare the same immediate value as the first dcl_literal, we scan the whole program to find the use of the register defined by the second dcl_literal and then replace them to use the register defined by the first dcl_literal. Finally the second dcl_literal can be removed. Since registers will never be spilled, we can always have this transformation. Fig. 4 shows a simple example for this optimization. Fig. 4(b) is the generated code for the program in Fig. 4(a) without this optimization. After applying this optimization, the generated code can be reduced to Fig. 4(c).

2) Private Memory Load Reduction: To handle OpenCL private memory access on ATI GPU, a reasonable method is to map the private memory space to the large number of registers. Therefore we change the behavior of the expansion phase for memory load/store so that the load/store operations will become data movement between registers and the ATI mov instruction can be generated accordingly. However, without further optimizations, there will be many mov instructions in the generated code. To solve this problem, we use a frame table to record the mapping between each TN and variable. Then there is no need to generate mov instruction for private load, only a table lookup at compile time required to get the TN for the private load. Take Fig. 5 for example, there is an instruction in Fig. 5(a) to add A with B and the result will



```
a = (1.0f - tmp) * 30.0f + tmp * 5.0f;
b = (5.0f - tmp) * 1.0f + tmp * 30.0f;
```

(a)

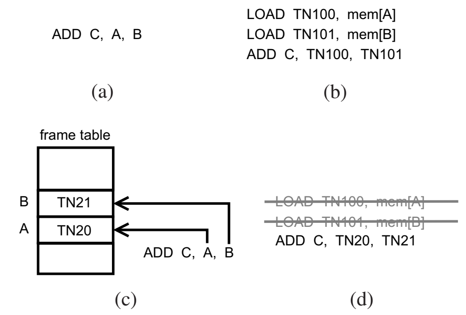
```
dcl_literal l10, 1084227584
dcl_literal l11, 1106247680
dcl_literal l12, 1065353216
sub r24, l12, r23
mul r25, r24, l11
mul r26, r23, l10
add r27, r25, r26
sub r29, l10, r23
mul r30, r29, l12
mul r31, r23, l11
add r32, r30, r31
```

(c)

```
dcl_literal l10, 1065353216
sub r24, l10, r23
dcl_literal l11, 1106247680
mul r25, r24, l11
dcl_literal l12, 1084227584
mul r26, r23, l12
add r27, r25, r26
dcl_literal l13, 1084227584
sub r29, l13, r23
dcl_literal l14, 1065353216
mul r30, r29, l14
dcl_literal l15, 1106247680
mul r31, r23, l14
add r32, r30, r31
```

(b)

Fig. 4. Remove Redundant dcl_literal



```
ADD C, A, B
```

(a)

```
LOAD TN100, mem[A]
LOAD TN101, mem[B]
ADD C, TN100, TN101
```

(b)

~~LOAD TN100, mem[A]~~
~~LOAD TN101, mem[B]~~
ADD C, TN20, TN21

(d)

Fig. 5. Table Lookup for Private Load

be store to C. Before doing this addition we have to load A and B from their memory address thus there will be two load instructions prior to the addition as shown in Fig. 5(b). To apply the table lookup method, we maintain a frame table as shown in Fig. 5(c) so that it can record that A will be mapped to TN20 and B will be mapped to TN21. Finally, we can do the addition on TN20 and TN21 without the two load instructions as shown in Fig. 5(d). Note that the register allocation method mentioned in the previous section is required to make this table lookup method workable.

IV. Experiment

We now describe the methodology for our experiments. We choose ATI Radeon HD5670 GPU as our target processor and use ATI Stream SDK 2.1 [21] as our runtime environment. The original OpenCL compiler of ATI SDK is based on LLVM and has several stages for the compilation. As shown in Fig. 6, the dotted line is the LLVM compilation flow that will first invokes clc command to compile the OpenCL program to its intermediate representation, .bc file. Then it will invoke llc to translate the .bc file into the ATI GPU assembly language, .il file. Finally, the runtime library uses a post-optimizer to optimize the .il file and then the final .il file will be executed. To integrate our Open64 based OpenCL compiler into the compilation flow, we replace the clc and llc commands with our Open64 compiler and generate the .il file directly, like the solid line in Fig. 6. However, as our compiler always generate the patterns that the post-optimizer can not recognize, the .il

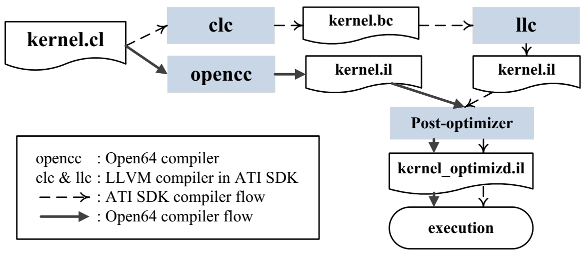


Fig. 6. Runtime Compilation Flow

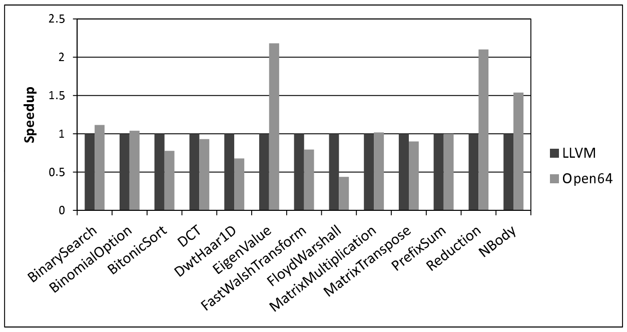


Fig. 7. Performance improvements of OpenCL compiler

file is not changed after the post-optimizer. We can not have any benefit from the post-optimizer.

The selected benchmarks for evaluating our Open64 based OpenCL compiler are from OpenCL samples of ATI Stream SDK which includes many common applications such as DCT, MatrixMultiplication, and EigenValue. Besides, the scalar operations and vector operations are also included, which is able to evaluate the functionality of the compiler front-end for vector parsing. By using these benchmarks, our Open64 based OpenCL compiler is proved to have high reliability.

We also use the original LLVM based compiler in ATI SDK runtime library as our comparison base to evaluate the performance of Open64. The experimental result can be seen in Fig. 7. As we can see from this figure, on some benchmarks such as EigenValue and Reduction, Open64 is two times faster than LLVM in ATI SDK and Open64 can have an average speedup of 11.7% for all benchmarks. However, we also find that the performance of Open64 decrease a lot for FloydWarshall. The reason is that LLVM performs common subexpression elimination very well on this program. As our compiler is just up and running, we expect performance improvements ahead in this infrastructure.

V. Conclusion

In this paper, many issues for supporting OpenCL based on Open64 compiler infrastructure are discussed, including the compiler front-end, the compiler back-end, and optimizations. The methodology to parse OpenCL specific address qualifiers and vector operations are discussed in the front-end. In the back-end, the target dependent modifications such as register allocation for ATI GPU are described. Besides, we implement two optimizations in Open64 for ATI GPU to support OpenCL. The experimental result shows that our OpenCL compiler is able to pass many OpenCL programs and can performs better than the original compiler in ATI SDK with 11.7% improvements in average in our tested benchmarks.

References

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Research and Development*, vol. 49, no. 4/5, pp. 589-604, 2005.

[2] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation cell processor," In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pp. 184-185, February 2005.

[3] H. Jiang, and T. A. Piazza, "Intel Next Generation Microarchitecture Code Named SandyBridge," In *Intel Developer Forum*, 2010.

[4] The AMD Fusion Family of APUs, *Website. Online available at http://fusion.amd.com.*

[5] K.-Y. Hsieh, Y.-C. Liu, P.-W. Wu, S.-W. Chang, and J. K. Lee, "Enabling Streaming Remoting on Embedded Dual-core Processors," In *Proceedings of the 37th International Conference on Parallel Processing (ICPP)*, 2008.

[6] GCC, the GNU Compiler Collection *Website. Online available at http://gcc.gnu.org.*

[7] NVIDIA CUDA, "Compute Unified Device Architecture," *Website. Online available at http://developer.nvidia.com/category/zone/cuda-zone*

[8] AMD's Close-to-the-Metal project, *Website. Online available at http://amdctm.sourceforge.net.*

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777-786, 2004.

[10] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[11] The Khronos Group - Open Standards for Media Authoring and Acceleration *Website. Online available at http://www.khronos.org.*

[12] Open64, *Website. Online available at http://www.open64.net.*

[13] L Dagum, and R Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, 5(1):46-55, Jan.-Mar. 1998.

[14] C. Lattner, and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[15] C. Lattner, and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.

[16] Apple Xcode, *Website. Online available at http://developer.apple.com/xcode.*

[17] Y.-C. Lin, Y.-P. You, and J.-K. Lee, "PALF: Compiler Supports for Irregular Register Files in Clustered VLIW DSP Processors," *Concurrency and Computation: Practice and Experience*, 2007:19:1-16, Wiley, 2007.

[18] Intel OpenCL SDK, *Website. Online available at http://software.intel.com/en-us/articles/opencl-sdk/.*

[19] Khronos OpenCL Working Group, "The OpenCL 1.1 Specification, (revision 36, September 30, 2010)," *Website. Online available at http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf*

[20] C.-B. Kuan, and J.-K. Lee, "SIMD Intrinsic Supports for VLIW DSP Processors with Distributed Register Files," In *Proceedings of Workshop on Compilers for Parallel Computing, (CPC)*, 2010.

[21] AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream), *Website. Online available at http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx.*

[22] M. Murphy, "NVIDIA's Experience with Open64," In *Proceedings of the Open64 Workshop at the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[23] R. Ju, S. Chan, C. Wu, R. Lian, and T. Tuo, "Open research compiler (ORC) for Itanium processor family," Tutorial presented at *the 34th Annual International Symposium on Microarchitecture (MICRO)*, 2001.