A Translation Framework for Automatic Translation of Annotated LLVM IR into OpenCL Kernel Function

Chen-Ting Chang, Yu-Sheng Chen, I-Wei Wu, and Jyh-Jiun Shann

Dept. of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{deferplay,ansoncat}@gmail.com, {wuiw,jjshann}@cs.nctu.edu.tw

Abstract. Heterogeneous multi-core processor is proposed to accelerate applications using an application-specific hardware, such as graphics processing unit (GPU). However, heterogeneous multi-core processor is difficult to program. Therefore, OpenCL (Open Computing Language) standard recently has been proposed to reduce the difficulty. A program of OpenCL mainly consists of the host code (executed on CPU) and the device code (executed on GPU or other accelerators). LLVM (Low Level Virtual Machine) is a compiler infrastructure and supports a variety of front-ends into LLVM IR (Intermediate Representation). To help translate programs written by different programming languages of LLVM front-ends to OpenCL, this work defines some extensions of LLVM IR to represent the kernel function of OpenCL. Furthermore, a translation framework is designed and implemented to translate annotated LLVM IR to OpenCL kernel function.

Keywords: OpenCL, LLVM, heterogeneous multi-core.

1 Introduction

Heterogeneous multiprocessor platforms have much higher potential performance gain than homogeneous multiprocessor platforms [1]. According to the characteristics of a program, the program may be partitioned into different properties of tasks, and then the tasks will be scheduled on suitable processors, for example, a parallel code is suitable on GPU. Nevertheless, heterogeneous multiprocessor platforms are harder to program. To improve this issue, at present, there have been many parallel programming standards being proposed, one of them is OpenCL.

OpenCL is an open standard for parallel programming of the heterogeneous processors such as multi-core CPU, GPU, Cell/B.E., and DSP and so on[1]. The program of OpenCL mainly consists of the host code (executed on the CPU) and the device code (executed on the GPU or other accelerators). In OpenCL, the device code is called kernel function. To support different platforms, the kernel function usually exists in the format of source code and is compiled dynamically. However, a lot of programs written by many parallel programming frameworks cannot be executed on the platform of OpenCL. To help translate these programs to OpenCL, this work defines several extensions of LLVM IR to represent the kernel function of OpenCL. Furthermore, a translation framework called the Annotated LLVM IR to OpenCL

J.-S. Pan et al. (Eds.): *Advances in Intelligent Systems & Applications*, SIST 21, pp. 627–636.
DOI: 10.1007/978-3-642-35473-1_62

Kernel Function Translation Framework is designed and implemented. The framework is derived from LLVM C back-end.

The rest of this work is structured as follows. Section 2 studies the background Section 3 then presents our translation framework. Next, Section 4 presents the experimental results. Conclusions are finally drawn in Section 5.

2 Background

This section introduces the parallel programming model of OpenCL and the compiler infrastructure of LLVM.

2.1 OpenCL

OpenCL is a standard for parallel programming of heterogeneous multiprocessors such as multi-core CPUs, GPUs, Cell/B.E., DSP and other [1]. OpenCL framework includes a language for writing kernel functions (executed on OpenCL devices), API, libraries and a runtime system to support software development. The core ideas of OpenCL specify four architecture models: platform model, execution model, memory model, and programming model [3].

The platform model is one host to connect one or more OpenCL devices. One or more compute units (CUs) compose an OpenCL device, and one or more processing elements (PEs) compose a compute unit. The processing elements within a device perform the computation. An OpenCL program is divided into two parts: (1) the host program which is executed on the host, and (2) the kernel which is executed on the OpenCL devices. OpenCL defines context and scheduling kernel to execute on OpenCL devices by the host program.

In OpenCL, all work-items executing a kernel have access to four distinct memory regions [3], namely global memory, constant memory, local memory and private memory. The global memory permits read/write access to all work-items in all work-groups. The constant memory is a region of global memory and only permits read access to all work-items. The local memory is shared by all work-items in the same work-group. The private memory is private to a work-item.

OpenCL supports data parallel programming model and task parallel programming model as well as the hybrids of these two models [3]. Data parallel programming model indicates that a series of instructions uses different element of memory objectives, scilicet each work item executes as same as program, inserts different data to execute work item by global ID or local ID. Task parallel programming model indicates that each work item of work space inside is absolutely independent executing kernel program then others item. In this model, each work item is equal to work in a single compute unit, which has only work item that executed only by it.

2.2 LLVM

LLVM (Low Level Virtual Machine) is a compiler infrastructure that is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in

arbitrary programming languages [4]. Figure 1 shows the LLVM framework, LLVM supports a variety of front-ends into LLVM IR, and LLVM provides analysis and optimization to transfer into optimum LLVM IR. Finally, the LLVM back-end generates high level languages or target assembly codes.

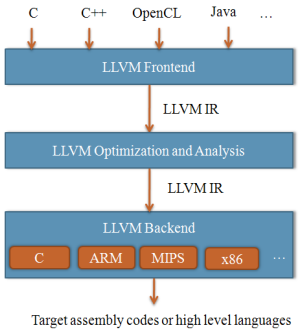

Fig. 1. LLVM framework

LLVM supports a language-independent instruction set and type system. Each instruction is in static single assignment form (SSA), i.e., each variable (called a typed register) is only assigned once [5]. SSA split operands value and storage location efficiently into a program, so that every definition gets its own version. Compiler optimization algorithms which are either enabled or strongly enhanced by the use of SSA, include constant propagation, value range propagation, sparse conditional constant propagation, dead code elimination, global value numbering, partial redundancy elimination, strength reduction, and register allocation.

The LLVM code representation has three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation [6]. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations [6].

3 Design and Implementation of Kernel Function Translation

Our translation framework consists of three parts: LLVM front-end, LLVM optimization phase and LLVM back-end, as shown in Figure 2. The LLVM front-end translates program languages with OpenCL annotations into annotated LLVM IR. The LLVM optimization phase optimizes annotated LLVM IR by a series of LLVM optimization passes (e.g., dead code elimination). Kernel extraction extracts the kernel function from the annotated LLVM IR of the input program and inserts OpenCL specific extensions, such as address space qualifier, into LLVM IR. Finally, the LLVM back-end of OpenCL translates annotated LLVM IR into OpenCL kernel function.

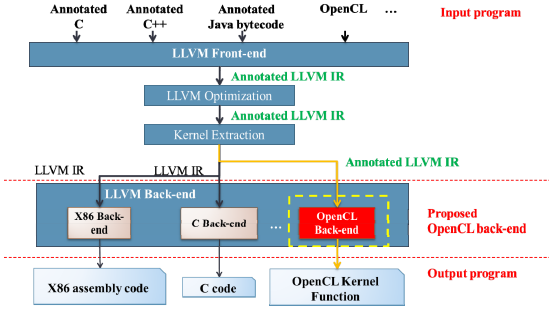

Fig. 2. System overview of modified LLVM

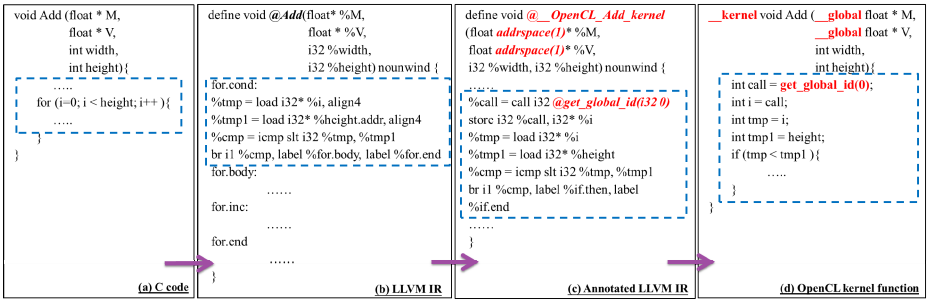

Fig. 3. Translation of the C code to OpenCL kernel function

Figure 3 illustrates how to translate C code to OpenCL kernel function through manually annotated LLVM IR. Firstly, the input program written by C is translated into LLVM IR by LLVM Clang C front-end as shown Figure 3 (b). The translated LLVM IR is manually inserted OpenCL annotations (e.g., kernel qualifier, address space qualifiers, and built-in functions of work-item). The OpenCL annotations are marked with red boldface in Figure 3 (c). The if statement in the annotated LLVM IR highlighted with blue dashed box is constructed from the for loop condition, and then loop's counter (e.g., increment and decrement) is replaced with built-in functions of work-item (e.g., get_globla_id(dim)). Finally, the annotated LLVM IR is translated into the OpenCL kernel function by our OpenCL back-end.

The OpenCL back-end consists of several steps as shown in Figure 4: initialization, function translation, and loop as well as basic block translation. The initialization declares all global variables, types and functions. In the function translation, the back-end translates the function prototype firstly. After translation, the back-end declares the local variable and then translate basic block within a function. If the back-end encounters a loop, it exits the basic block translation and enters loop translation. In the loop translation, the back-end constructs the loop body and then enters basic block translation again.

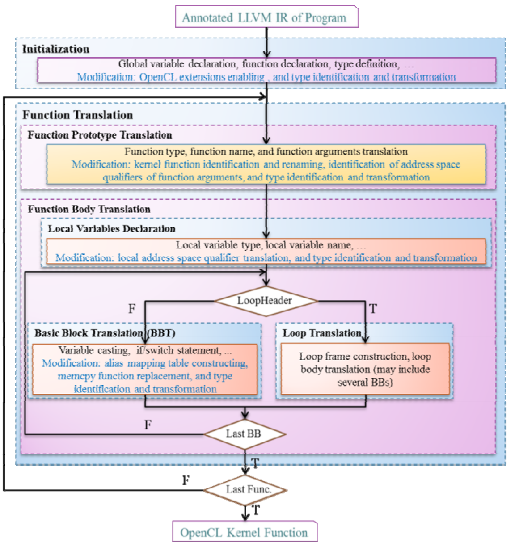

Fig. 4. Flow chart of the OpenCL back-end

The address space qualifier (i.e. location) of a variable in OpenCL program must be specified (such as global, constant, local or private); otherwise, the kernel function cannot be correctly executed because of the incorrect location of variable. To handle this problem, we add some extension in LLVM IR to indicate the type of and the address space qualifier of a variable. The mapping table between the address space qualifier of OpenCL and the address space attribute of LLVM IR is shown in Figure 5.

OpenCL		Annotated LLVM IR
Memory model	Address space qualifier	Address space attribute
Global memory	__global	1
Constant memory	__constant	3
Local memory	__local	2
Private memory	__private or none	no attribute

Fig. 5. The annotated LLVM IR mapping to the address space qualifiers of OpenCL

Some variable types used in OpenCL do not properly support in LLVM IR (e.g., half precision floating point, Image3D and Image2D). Thus, such variable types are difficult to translate back to C language from LLVM IR without any annotation. Half precision floating point (HPFP) data type is a 16 bit floating point format, and it must follow IEEE 754 -2008 half precision storage format. HPFP data type has one sign bit, five exponent bits, and ten mantissa bit. OpenCL supports HPFP data type and provides some operations of HPFP data type. Since LLVM IR did not support HPFP data type until LLVM 3.1 (Apr 2012), we also add some extension to support HPFP.

In OpenCL, the image object is a memory object that stores a two dimensional (2D) or three dimensional (3D) structured array. The image object consists of four parts: image data, dimensions of the image, description of each element in the image, and properties that describe usage information and which region to allocate from. In LLVM IR, the image data type of *image2d_t* and *image3d_t* are not supported. These image data types represent opaque structure type in annotated LLVM IR (e.g. struct._image2d_t or struct._image3d_t). The image data type is represented in LLVM IR as shown in Figure3-6. Due to the restrictions on the image data type, the variable operation of the image data type is not allowed in the kernel function. The OpenCL program uses the image data type of *image2d_t* or *image3d_t* based on the following restrictions: Image data type can only be the function argument, and it cannot be modified. Image data type cannot access the element of the image directly. OpenCL provides the built-in functions for image read and image write, pointers to image data type are not allowed. Image data type cannot be declared in a structure. The local variable and function return cannot use the image data type.

The sampler parameter must be set in the image read function when attempting to read the image variable in kernel function. The sampler data type is represented as*sampler_t* in OpenCL. The sampler data type is an unsigned 32 bits integer, and provides addressing mode, filter mode, normalized coordinates. The sampler data type has the following restrictions: it cannot be declared as a type of array or pointer. It cannot be defined without initializing local variables or as the return value in a function. It cannot be modified when the function argument is the sampler data type. In LLVM IR, sampler data type is not supported. The sampler data type represents i32 (integer) data type in annotated LLVM IR. Translation and identification of the sampler operation use the following steps:

I. Identify the name of the sampler argument of each image read function call in the kernel function.
II. For each sampler argument, construct the data dependence graph between the sampler argument, the function arguments and local variables of the kernel function.
III. If the root of the data dependence graph is a function argument:
    i. Change the data type of the function argument from i32 to sampler.
    ii. Replace the name of the sampler argument in the image read function call with the name of the function argument.
    iii. Eliminate all the local variables in the data flow graph, including the sampler argument.
IV. If the root of the data dependence graph is a local variable:
    i. Change the data type of a local variable from i32 to sampler.
    ii. Initialize the declaration of the sampler argument to the initial value of the local variable.
    iii. Eliminate all the local variables in the data flow graph, excepting the sampler argument.

The proposed OpenCL back-end identifies a kernel function from the function names of the annotated LLVM IR. A function name with annotations __OpenCL and _kernel is a kernel function, e.g., void__OpenCL_Add_Kernel as shown in Figure 6 (a). The OpenCL backend will rename the function name in the translated OpenCL program by eliminating the __OpenCL and _kernel annotations, e.g., Add, as shown in Figure 6 (b). The proposed OpenCL back-end identifies the address space attribute from the function arguments of annotated LLVM IR. Refer to Figure 5 for the mapping between the address space attribute in annotated LLVM IR and the address space qualifiers in OpenCL. In Figure 6, the proposed OpenCL back-end translates addrspace(1) of annotation LLVM IR into __global of OpenCL kernel function.

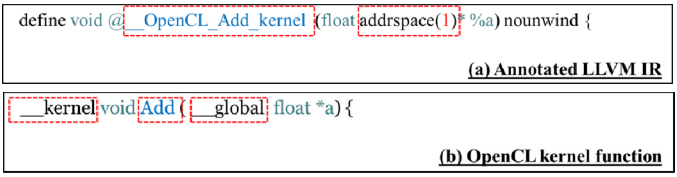

Fig. 6. An example code segment for function prototype translation and identification

The memcpy function is used to copy a block of memory from the source location to the destination location. In OpenCL, the memcpy function has a restriction that its arguments cannot contain address space qualifiers (e.g., __global, __local, __constant). The syntax of memcpy function is represented as memcpy (destination, source, length) in OpenCL and memcpy (addrspace destination, addrspace source, length, align, isvolatile) in annotated LLVM IR.

Traditional C language cannot support vector type, while OpenCL and LLVM IR can. he Vector data type defined by OpenCL is defined with a type name (char, short, int, float, long and unsigned) followed by a literal value n that defines the number of elements in the vector, n = 2, 4, 8, 16. The vector data type defined by LLVM IR includes the number of elements and element type. In this work, all vector variables in the input kernel function are translated to the vector data type of LLVM IR. To correctly translate back to OpenCL kernel function, the OpenCL back-end must translate the vector data type (LLVM IR) to corresponding format in OpenCL. Figure 7 is an example of translating the vector data type between LLVM IR and OpenCL.

Fig. 7. Example of translation between LLVM IR and OpenCL (vector data type)

The LLVM front-end would transform some local variables into global variables leading the program to have compiling and executing issue. To handle this isse, a mapping table is introduced as shown in Figure 8. For example, a variable of "__local int a" declared in OpenCL is considered as a global variable in LLVM IR.

OpenCL	Annotated LLVM IR	
Local variable declaration	Global variable declaration	Local variable declaration
__local int a;	*	
__local int a[1];	*	
__local struct a;	*	
__local int *a;		*
__local int4 a;		*
__local int *a[1];		*
__global struct a;		*
__global int *a;		*

Fig. 8. Function prototype identification and translation

Since the LLVM front-end will rename built-in function of OpenCL, the function calls of OpenCL program cannot call built-in function after translation. To overcome this problem, an alias mapping table is introduced as shown in Figure 9.

Function Class	OpenCL built-in function	Annotated LLVM IR
Math (C)	sqrt	sqrtf
Synchronization	Barrier (arg1)	Barrier (arg1 , arg2)
Math /native math	cos	_cos_f32
	Exp	_EXP_f32
Integer	abs	_abs_i32
Common	max	_max_2i32
Geometric	cross	_cross_4f32
3D image	imagef_image	__read_imagef_image3d4i32
2D image	imagef_image	__read_imagef_image2d2i32
Type conversion	convert_int4	__convert_int4_4u8
	convert_float4	__convert_int4_4f32
…		

Fig. 9. Alias mapping table

4 Experimental Results

This experiment is used to verify the correctness and performance of our OpenCL back-end. The experimental environment had a host with an Intel i7-920 and a device with NVidia GTX 460. The OpenCL back-end was developed based on LLVM 2.9. All benchmarks were selected from NVidia OpenCL SDK and translated to the annotated LLVM IR using the LLVM front-end developed by AMD. After performing optimization (-O2 in this study), the kernel function (LLVM IR format) is

then translated to the C code format through OpenCL back-end. By executing the translated the kernel function on the device (i.e. NVidia GTX 460), the correctness and performance could be verified. Figure 10 shows the executed time of before and after translated kernel functions. Original OpenCL and translated OpenCL denotes the before and after translated kernel functions, respectively. According to the experimental results, all benchmarks could be executed correctly and almost no performance is loss.

NVidia SDK	Execution time of clock cycle		Ratio (T/O)
	Original OpenCL	Translated OpenCL	
MatrixMul	68,413,437	68,166,441	0.99639
ConvolutionSeparable	164,322,856	162,524,385	0.98905
BlackScholes	101,483,905	101,712,190	1.00224
Tridiagonal	282,236,392	283,328,057	1.00386
Transpose	585,751,696	585,751,175	0.99999
SortingNetworks	450,134,553	450,791,892	1.00146
SimpleMultiGPU	179,908,761	179,744,934	0.99873
Scan	4,764,150,345	4,764,139,172	0.99999
RadixSort	2,333,536	2,404,680	1.03048
QuasirandomGenerator	1,005,084	1,060,628	1.05526
DXTCompression	58,962,028	59,706,196	1.01262
DotProduct	46,546,896	43,581,255	0.93628
FDTD3d	1,121,176,436	1,136,779,335	1.01391
VectorAdd	191,660,748	190,942,509	0.99625
SobelFilter	15,052,436	15,098,284	1.00304
SimpleTexture3D	1,372,866	1,374,720	1.00135
SimpleGL	1,308,694	1,309,972	1.00097
RecursiveGaussian	24,529,491	24,326,601	0.99172
PostprocessGL	7,519,812	7,552,768	1.00438
MedianFilter	26,438,834	27,167,774	1.02757
DCT8x8	74,749,455	74,676,294	0.99902
MarchingCubes	1,579,408	1,583,066	1.00231
Nbody	8,043,914	8,015,520	0.99647
Particles	3,512,163	3,459,612	0.98503
MersenneTwister	857,732	840,019	0.97934
HiddenMarkovModel	369,065,620	371,369,797	1.00624
CopyComputeOverlap	10,221,842,360	10,210,928,766	0.99893
Histogram	90,223,175	95,434,984	1.05776
MatVecMul	1,044,741,185	1,056,151,563	1.01092
Average			1.00350

Fig. 10. Execution time of before and after translated kernel functions

5 Conclusion

In this work, we designed and implemented an OpenCL back-end in LLVM compiler infrastructure. According to the experimental results, the OpenCL back-end could correctly translate all benchmarks from NVidia SDK and almost no performance loss

occurs. In this work, we only performed the default optimization (O2) of LLVM compiler framework on the kernel function. However, the program characteristic of the kernel function (with massive parallelism) is different from the sequential function or the function with limited parallelism. And, the target device architecture of OpenCL is significantly different from the host one. As consequence, most optimization passes in the default optimization does not perform well on the kernel function. To further increase the performance, designing the optimization pass specified for the kernel function would be of interest in the future. Furthermore, a new specification called SPIR (Standard Portable IR) [7] for OpenCL Kernel is released by Khronos Group Inc. at August 24, 2012. SPIR is a mapping from the OpenCL C programming language into LLVM IR. Obviously, the major objective of both our IR extension and SPIR is almost same. Therefore, modifying our framework to support SPIR would be another interesting topic.

References

1. Tsai, T.-C.: OMP2OCL Translator: A Translator for Automatic Translation of OpenMP Programs into OpenCL Programs. MS. Thesis, National Chiao Tung University (2010)
2. Hruska, J.: AMD Announces New GPGPU Programming Tools, http://hothardware.com/News/AMD-Announces-New-GPGPU-Programming-Tools
3. Khronos OpenCL Working Group, The OpenCL Specification 1.0 (2009)
4. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (2004)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. In: ACM Transactions on Programming Languages and Systems (1991)
6. LLVM Developer Group, LLVM Language Reference Manual, http://llvm.org/docs/LangRef.html
7. Khronos Group, SPIR 1.0 Specification for OpenCL (2012)