GPU-S2S: a compiler for source-to-source translation on GPU

Dan Li, Haijun Cao, Xiaoshe Dong, Bao Zhang
Department of Computer Science and Technology
Xi'an Jiaotong University
Xi'an, China, 710049.
li.dan@stu.xjtu.edu.cn, { caohaijun, xsdong }@mail.xjtu.edu.cn

Abstract

CUDA facilitates the development of General Purpose computing on Graphics Processing Units (GPGPU), however, its complex memory system, thread-level structure, and data transmission control between memories have brought great challenges for programming on GPU. In order to facilitate the development of parallel programs on GPU and reuse existing sequential codes, in this paper we propose a novel directive based compiler guided approach. Through combining automatic mapping and static compilation, we have implemented a prototype of automatic source-to-source translation tool named GPU-S2S, capable of translating the C sequential code with directives into CUDA code. Experimental results show that CUDA code generated by GPU-S2S can achieve comparable performance with that of CUDA benchmark provided by NVIDIA CUDA SDK, and has significant performance improvements compared with its original C sequential code.

1. Introduction

CUDA (Compute Unified Device Architecture) has brought great changes for computing on GPU. However, to explore GPU's tremendous computational power, developers still need to optimize applications from load balancing [1], memory space management [2] [3], and so on. Recently many researchers devote themselves in compiler directives to solve software portability and scalability on GPU.

NOAA's Earth System Research Laboratory developed a compiler called F2C-ACC [4] that is to convert Fortran into CUDA. University of Waterloo, Canada developed Sh [5] which is used to draw and make general purpose computing on GPU. Also, school of ECE, Purdue University develops a compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications [6]. Three software translation tools mentioned above require highly about mapping computing to graphics hardware; most of times, before getting an optimized program, programmers need to optimize manually even make significant code changes.

Impact Research Group at UIUC developed an automatic memory optimization translation tool via annotations for CUDA code called CUDA-Lite [7]. But programmers still need to write a complete CUDA code first.

In order to solve these problems, in this paper we develop a prototype of automatic source-to-source translation tool named GPU-S2S, capable of mapping compute-intensive applications from homogeneous platform to GPU streaming architecture platform.

2. GPU-S2S

Figure 1 shows the structure of our source-to-source compiler translation tool GPU-S2S. The procedures of GPU-S2S are as follows: (1) Compiler verification. In order to guarantee the correctness of the C sequential code with directives (*.c and *.h file), firstly it is compiled by the C compiler. (2) GPU-S2S translation. As the most important phase, GPU-S2S receives above verified C sequential code with directives as input, and then transforms it into the equivalent CUDA code (*.h,*.cu, and *.c file) that uses CUDA runtime library to replace corresponding directives. (3) CUDA compiling. Generated CUDA code in the second step should be compiled by CUDA Compiler Tool Chain from NVIDIA, resulting in executable code (*.o file).

GPU-S2S consists of 3 modules: the directive recognition and parsing module(the directive specification will be detailed in Section 2.1), the share memory allocation and the thread block partition module(detailed in Section 2.2), and the CUDA code generation module(detailed in Section 2.2).

978-0-7695-4312-3/10 $26.00 © 2010 IEEE
DOI 10.1109/PAAP.2010.43

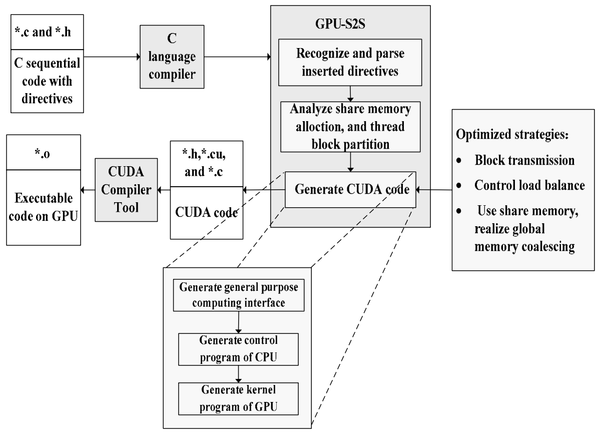
*.c and *.h
C sequential code with directives
C language compiler
GPU-S2S
Recognize and parse inserted directives
Analyze share memory alloction, and thread block partition
Generate CUDA code
Optimized strategies:
Block transmission
Control load balance
Use share memory, realize global memory coalescing
*.o
Executable code on GPU
CUDA Compiler Tool
.h,.cu, and *.c
CUDA code
Generate general purpose computing interface
Generate control program of CPU
Generate kernel program of GPU

Figure 1. Structure and execution flow of GPU-S2S

In order to generate optimized CUDA code, at the the CUDA code generation module, according to the extracted information above, GPU-S2S adopts corresponding optimized strategies to finish the source-to-source translation. The procedures in detail can be seen in Figure 1. Until now, GPU-S2S supports three optimized strategies shown in Figure 1.

Figure 2 presents a standard framework of program before and after translation. From Figure 2, we can see that both the native C sequential code with directives and the generated CUDA code contain the user defined part and the calling shared library.

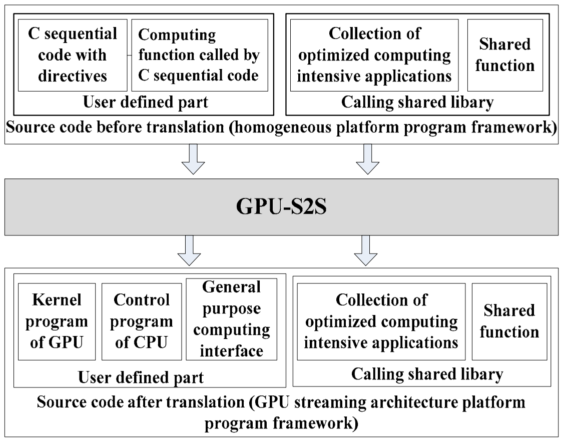
C sequential code with directives
Computing function called by C sequential code
User defined part
Collection of optimized computing intensive applications
Shared function
Calling shared libary
Source code before translation (homogeneous platform program framework)
GPU-S2S
Kernel program of GPU
Control program of CPU
General purpose computing interface
User defined part
Collection of optimized computing intensive applications
Shared function
Calling shared libary
Source code after translation (GPU streaming architecture platform program framework)

Figure 2. Standard framework of program before and after translation by GPU-S2S

2.1. GPU-S2S directives

Until now, GPU-S2S provides five directives.

1) Memory mapping directive

This directive is mainly used to declare variables required to be allocated in the global memory. It has the following format:

#pragma task input: \
Variable_Type Variable_Name Variable Space

#pragma task output: \
Variable_Type Variable_Name Variable Space

Here input refers to data copy from CPU memory to GPU global memory. Output refers to data transmission in the opposite direction. Variable_Type is the type of the variable; Variable_Name is the name of the variable; and Variable Space specifies the global memory space needed to be allocated for the variable. Declare format is as follows:

[Dimensions_Size]{[Dimensions_Size] *}

Here {} represents additional parameters if more than one dimension.

Two directives above shield the copy process between CPU and GPU and realize the transparent of global storage variables to programmers. As to variable declared by input directive, those data can be copied to the GPU global memory once in batch if there is enough global memory space on GPU.

2) Kernel region identify directive

This directive determines the computing kernel that will be executed on GPU in the C sequential code and parallelization factors. The computing kernel will be executed by each thread. It has the following format:

```
#pragma kernel:tblock < ThreadBlocks_Partition >\
thread < Threads_Partition >
……    //the code of computing kernel
#pragma kernel_end:
```

Here ThreadBlocks_Partition defines the arrangement of thread block in the grid. Declaration format is as follows:

ThreadBlocks_Per_Grid {, ThreadBlocks_Per_Grid *}

Threads_Partition defines the arrangement of thread in the thread block. Declare format is as follows:

Threads_Per_Block {, Threads_Per_Block*}

Parameters above are the linear combination of data size and thread block size when the dimension is taken into account. They affect the parallel significantly.

3) Loop mapping directive

In the computing kernel of C sequential code with directives, if this directive is inserted before a loop, the loop can be executed by threads concurrently. It has the following format:

#pragma parallelizable loop part

According to the dimension of variable declared in the memory mapping directive, GPU-S2S converts the iteration space of loop to thread-block space in CUDA. It is worth mentioning that if this directive is not inserted into the front of a loop, all of the iterations will be serially executed by each thread.

For example, in the FFT program,

```
for (int L = 1; L <= level; L++){  // conveys that all
                                   //of the iterations will be serially
                                   //executed by each thread
   ......
```

```
  #pragma parallelizable loop part:
                              //Loop mapping directive
  for (int k = 0;k < N / 2; k++){//each thread only
                         //executes one iteration of loop
     ......
  }
 }
```

4) Share memory directive

Share memory directive is used to allocate the share memory for input or output variable declared in Memory mapping directive. It has the following format:

```
#pragma shared alloc copyin: \
Variable_Type Variable_Name Variable _Space
#pragma shared copyout to: \
Variable_Type Variable_Name Variable _Space
```

Here copyin refers to copying the data from global memory to share memory on GPU. Copyout to refers to the data transmission in the opposite direction. Variable Space specifies the share memory space needed to be allocated for the variable. Declare format is as follows:

[Xdimension_Relative_StaAddress:Xdimension_Relative_StaAddress+Size]{[Ydimension_Variable][Zdimension_Variable]}or{[Xdimension_Variable]}[Ydimension_Relative_StaAddress:Ydimension_Relative_StaAddress + Size] {[Zdimension_Variable]}

During the process of transformation, according to the variable in this directive, GPU-S2S uses corresponding share variable to replace it.

5) Synchronous directive

Synchronous directive is used to ensure that all threads in a thread block have finished computing. It has the following format:

```
#pragma synchronous:
```

2.2. Translation strategy

As showed in Figure 1, after the programmer inserts directives into the proper place of C sequential code, the directive recognition and parsing module of GPU-S2S scans the C sequential code one by one, extracts relevant information, and stores them in the 3-dimensional array, preparing for generating CUDA code. It is worth mentioning that GPU-S2S makes use of profiling guided optimization, that is, through inserting some statistical code in C sequential code, GPU-S2S collects the frequency of each program branch by executing the source code several times, finally it makes sure computing kernel to guide programmer to insert kernel region identify directive.

Because of some features of GPU architecture (1) GPU's share memory space is limited, to some applications, input data can't be loaded into share memory at a time; (2) Number of thread blocks and threads supported by each SM(streaming multiprocessor) is also limited. Based on the consideration above, the share memory allocation and the thread block partition module in GPU-S2S makes the relevant judgement.

The algorithm can be described as follows.

```
Input: Set of share memory directive – S
Output: Number of active thread blocks per SM actually –
          Thread_Block_Num;
        Allocated whole share memory per SM–Whole_Share_Size;
        Number of sequentially executable kernel.
for each $s_i \in S$ //$s_i$ represents i-th Share memory directive
    Extract the expression of allocated share memory for
    variable from $s_i$;
    Scan C sequential code with directives, replace macro
    variable in expression into corresponding value, and
    then calculate the value of the expression;
    Scan type size of variable in variable table, and then
    compute space size of the share memory allocated for
    the variable: Share_Size;
    Share_Per_ThreadBlock += Share_Size;
     \\Share_Per_ThreadBlock represents space size of the
     \\share memory allocated for the variable in all Share
     \\memory directives per thread block.
end for
Compute the number of active thread blocks per SM actually
 – Thread_Block_Num;
Whole_Share_Size = Share_Per_ThreadBlock *Thread_Block_Num;
if (Whole_Share_Size > maximum space of share memory)
            // distribution is incorrect
    Stop, make redistribution;
else  //distribution is correct
    if (step between computing data > data size which threads in each
          thread block are capable of computing)
        Distribute computing kernel to several sequentially
        executable kernel, and export the number of
        sequentially executable kernel;
    else
        Distribute computing kernel to one executable kernel,
        and export the number of executable kernel is 1;
    end if
end if
```

After finishing these above two steps, the CUDA code generation module begins to translation. It mainly consists of three steps: (1) According to framework of homogeneous platform program before translation, GPU-S2S generates the shared library calling and general purpose computing interface; (2) Combined with C sequential code with directives, GPU-S2S generates control program of CPU. Through analyzing memory mapping directives, GPU-S2S will insert the memory allocation code, data transmission code, and free memory space code in the generated CUDA code;

and (3) GPU-S2S generates the kernel program of GPU.

Figure 3 shows the translation mechanism of generating kernel program.

(1) According to the information extracted from the loop mapping directive, GPU-S2S uses it to perform loop transformation, that is, GPU-S2S converts the iteration space of loop in C sequential code to thread-block space in CUDA.

(2) According to the information extracted from the share memory directive, GPU-S2S has each thread copy data from the appropriate place in global (share) memory to the corresponding place in share (global) memory. Therefore, GPU-S2S realizes global memory coalesced [8] loading or storing to share memory.

(3) According to the information extracted from the synchronous directive, GPU-S2S generates the corresponding CUDA runtime functions.

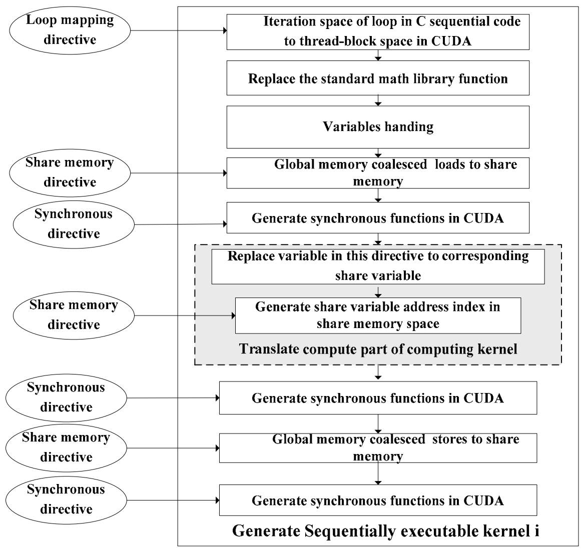
Loop mapping directive
Iteration space of loop in C sequential code to thread-block space in CUDA
Replace the standard math library function
Variables handing
Share memory directive
Global memory coalesced loads to share memory
Synchronous directive
Generate synchronous functions in CUDA
Replace variable in this directive to corresponding share variable
Share memory directive
Generate share variable address index in share memory space
Translate compute part of computing kernel
Synchronous directive
Generate synchronous functions in CUDA
Share memory directive
Global memory coalesced stores to share memory
Synchronous directive
Generate synchronous functions in CUDA
Generate Sequentially executable kernel i

Figure 3. Translation mechanism of generating kernel program on GPU

In addition to above transformations, when generating kernel of GPU, GPU-S2S has to do other work, for instance, replacing the standard math library function to CUDA standard library function; and if a variable used in C sequential code is not satisfied with the requirement of CUDA, GPU-S2S has to introduce new variables, assigns them, and changes values of old variables.

3. Experiment

In this Section, we make experiments to analyze and verify the correctness and performance of GPU-S2S.

The experimental platform adopts RedHat Enterprise Server version 5.3 Operating System and CUDA version 2.3 which runs on Inspur NF5588 server.

Table 1 presents the execution time of matrix multiplication for different size of input array. The program execution time on GPU here includes the time required for data transfer between CPU and GPU. In table 1, GPU-S2S represents the generated CUDA program which is translated by GPU-S2S from C sequential code with directives; SDK-Example represents the standard CUDA matrix multiplication benchmark from NVIDIA SDK; and CPU represents C matrix multiplication sequential program. Experimental results show that for different size of input array, CUDA code generated by GPU-S2S can achieve comparable performance to that of SDK-Example, in some cases, translated code has better performance, because although both porgrams make use of global memory coalesced load to share memory, algorithm and optimized strategies are different, also there may exist accuracy error in repeated test. As the size of input array increases, the speed-up of execution time between CPU and GPU-S2S grows fast, from 96 to 1677.5. This is mainly due to the GPU’s powerful parallel processing capability and high memory bandwidth. When the size of input array is small, the execution time contains the overhand caused by transferring data between GPU and CPU. In contrast, when the size of array is big enough, parallel processing capability of GPU can be fully exploited.

Table 1. Execution time of matrix multiplication for different size of input array (ms)

	256*256	512*512	1024*1024
GPU-S2S	0.5280	3.0632	14.2465
SDK-Example	0.5576	3.1129	14.2675
CPU	50.9270	447.3130	10577.0918

	2048*2048	4096*4096	8192*8192
GPU-S2S	96.7969	728.3226	5692.1143
SDK-Example	96.6832	727.6077	5691.3530
CPU	95450.297	837221.25	954636

Figure 4 compares the execution times of matrix multiplication generated by GPU-S2S for different size of input array (divided by 16 and non-divided by 16). Experimental results show that in each case, although the size of the input data are close, the input data size which is non-multiples of 16 costs more execution time on GPU. For example, to be computed on GPU, matrix of 1000*1000 costs more time than matrix of 1024*1024, and this trend becomes clear as the size of input matrix grows. It is mainly because the memory controller of GPU reads data from the address of multiple of 16 bytes; at the same time, when the size of input array is not the multiple of thread number contained in half-warp (one half-warp contains 16 threads on G80), the last thread block may have some idle threads which will not participate in computation. Above all, it leads to poorer performance when the size of input array is non-divided by 16.

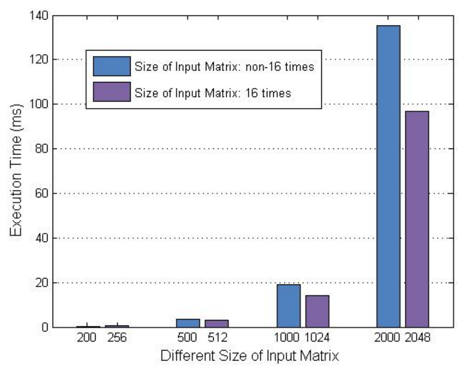

Figure 4. Comparison of execution time for matrix multiplication generated by GPU-S2S

Figure 5 illustrates the execution time of Fast Fourier Transform (FFT) for different size of input data. Here, CUFFT represents program calling CUFFT function library provided by NVIDIA. Experimental results show that for different size of input data, CUDA code generated by GPU-S2S can achieve better performance than that of CUFFT program provided by NVIDIA. With the increasement of input data size, the execution times of the CUDA program transformed by GPU-S2S and the CUFFT program increase slowly and steadily, while the execution time of the serial FFT program undergoes an exponential increase. This fully demonstrates the powerful parallel processing capability of GPU.

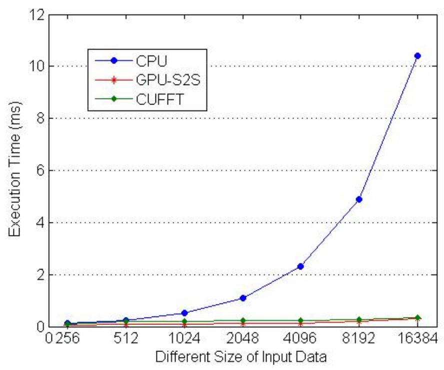

Figure 5. Execution time of FFT for different size of input data

4. Conclusions

In this paper, we have implemented a prototype of automatic source-to-source translation tool named GPU-S2S, capable of translating the C sequential code with directives into CUDA code. Experimental results show that CUDA code generated by GPU-S2S can achieve comparable performance to that of CUDA benchmark provided by NVIDIA and has significant performance improvements compared with its original C sequential code.

Acknowledgments

This work is supported by the National High Technology Research and Development Program (863 Program) of China under grant No. 2009AA01Z108, No. 2009AA01A135, and 2006AA01A109. Part of the work is supported by the Fundamental Research Funds for the Central Universities under grant 08142007.

References

[1] S. Ryoo, S.S. Rodrigues, C.I. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.M.W. Hwu, "Optimization Principlesand Application Performance Evaluation of a Multithreaded GPU Using CUDA", *In: Proceedings of Sympium on Principles and Practice of Parallel Programming*, 2008, pp.73-82.

[2] M. Moazeni, A. Bui, and M. Sarrafzadeh, "A Memory Optimization Technique for Software-managed Scratchpad Memory in GPUs", *In: 2009 IEEE 7th Symposium on Application Specific Processors*, 2009, pp.43-49.

[3]. N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A Memory Model for Scientific Algorithms on Graphics Processors", *In: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[4]. M. Govett, J. Middlecoff, and T. Henderson, "Runningthe NIM Next-Generation Weather Model on GPUs", *In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, VIC, Australia, May 2010.

[5]. M.D. McCool, Z. Qin, and T.S. Popa, "Shader Metaprogramming", *In: Proceedings of ACM SIGGRAPH / EUROGRAPHICS*, 2002, pp. 57–68.

[6]. S. Lee, S.J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization", *In: Proceeding ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[7] S.Z. Ueng, M. Lathara, S.S. Baghsorkhi, and W.M.W. Hwu, "CUDA-lite: Reducing GPU Programming Complexity", *In: Languages and Compilers for Parallel Computing (LCPC) 21st Annual Workshop*, August 2008, pp. 1-15.

[8] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide (Version 2.0), NVIDIA Corporation, July 2008.