Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 352–361

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

APTCC : Auto Parallelizing Translator From C To CUDA

Takehiko Nawata[a,*], Reiji Suda[a,b]

[a] *Department of Computer Science, Graduate School of Information Science, the University of Tokyo*
[b] *Japan Science and Technology Agancy, Core Research for Evolutional Science and Technology*

Abstract

This paper proposes APTCC, Auto Parallelizing Translator from C to CUDA, a translator from C code to CUDA C without any directives. CUDA C is a programming language for general purpose GPU (GPGPU). CUDA C requires us a special programming manner differently from C. Although there are several pieces of research to reduce this difficulty, the result of those researches still compels us to beware of GPU architecture. It is better however that we are able to concentrate on the algorithm. Hence we propose translation of C code into CUDA C optimized to the target GPU architecture without directives, where the complexity of the GPU hardware is transparent to the programmer.

In translating a C code to a CUDA C code, two questions have to be answered. The first question is how to select the code fragments which should be translated into CUDA C, and the second question is how to translate the selected code fragments into CUDA C. To the first question, this paper proposes a heuristic selection scheme based on the loop structure of the source code. The current implementation of APTCC selects nested loops for the target of translation. To the second question, APTCC translate all the statements in the body of outermost loop into CUDA C.

This paper explains the detailed implementation of APTCC and compares the results of performance comparison of a few experimental input source codes.

Keywords: GPGPU, CUDA, Translation, Auto-Parallelization

1. Introduction

In recent years, GPU comes to be used for general computation rather than only for graphics processing. Because the GPU is designed to process a huge amount of pixels in a short time, it is suitable for applying the same sequence of operations to a large number of data. Stencil computation is a typical example of this kind of computations from the areas other than graphics processing [1].

Although in old days there were no standard language to control GPU, a C-extension language for general purpose GPU (GPGPU) called CUDA C has been released by NVIDIA in 2007[2]. CUDA stands for Compute Unified Device Architecture. CUDA C is not the first high-level language to control GPU. Direct3D has a high-level programming language called High Level Shader Language (HLSL)[3] to write shader programs and HLSL became available from DirectX3D versions DirectX9 and later. OpenGL also has a high-level shader language called OpenGL Shading Language (GLSL)[4]. Cg [5] is a high-level language to write shader programs which was offered by NVIDIA before

*Corresponding author
Email addresses: samugari@is.s.u-tokyo.ac.jp (Takehiko Nawata), reiji@is.s.u-tokyo.ac.jp (Reiji Suda)

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.037

CUDA C. General purpose programming were tried with those shader languages, but it was required to program with strong restrictions and tricky usage of Graphics API for general purpose computing. Comparing with these high-level languages, CUDA C is a language for GPGPU. There is a new specification of a programming language for heterogeneous programming called OpenCL[6]. GPU is included as one of the target of OpenCL. The core programming language of OpenCL is called OpenCL C and it is similar to CUDA C. But CUDA C is the only target of this paper.

To learn and to write CUDA C is easy because CUDA C is a simple extension of C language. However, even using CUDA C, getting good performance — it must be the main reason for using GPUs for general purpose computing — is still difficult. The cause for this difficulty is the fact that CUDA C is too close to GPU hardware. We are forced to consider about the hardware implementation of GPU whenever we write high performance programs using CUDA C. We want to make it easier to get a CUDA C code that achieves good performance.

This paper proposes a translator named APTCC which takes C language source codes as inputs and outputs a CUDA C code which carries out the same calculations on GPU. In our approach, the knowledge of characteristics of GPU is included to APTCC and APTCC outputs the optimized CUDA C code based on that knowledge. We selected the programming language C to write source code because it is widely used.

There are many other approaches to make CUDA C easier: Wrapping typical functions into a library[7, 8], making a new language[9], defining a Domain Specific Language[10, 11], directive-based approach[12] and so on. Some are for writing CUDA C easier and some are for getting good performance CUDA C code easier. Some of these approaches are not targeted to only at CUDA C but also at OpenCL.

When we compare APTCC with these researches, the biggest advantage of APTCC is the elimination of any additions to the C language. APTCC does not introduce any new function, syntax, class and directive. APTCC accepts the standard C code and translates it into CUDA C code.

In this paper, the preliminary implementation of APTCC is discussed. Section 2 is for brief introduction of CUDA C and in section 3 we outline our proposal. The detail of current APTCC implementation is explained in section 4, and the performance of this preliminary implementation is evaluated in section 5.

2. CUDA C

This section is for brief introduction of CUDA C. To get more detailed information, programming guide of CUDA C[13] issued by NVIDIA is a good reference.

The syntax of CUDA C is almost same as C language but a few specifiers are added. In a program written in CUDA C, we call the CPU "host" and the GPU "device." To use one GPU, we have to write the data transfer codes and the definition of functions that are carried out computations on GPU. First the memory on the "device" must be allocated and the data for succeeding calculation on the "device" must be sent to the "device." The calculation on the device must be expressed as a function whose definition is given with the special specifier `__global__`. This function is called "kernel" and the "device" executes the "kernel" when the "host" invokes it. A GPU has several Multi Processors (MPs) and each MP has many Stream Processors (SPs). In the the latest product GTX580, the numbers of MP and SP is 16 and 32, respectively.

"Block" and "thread" in CUDA C are the abstraction of MP and SP, respectively. When the "host" invokes a "kernel," the "host" must specify the number of "block" and the "threads" per "block." One "block" runs on one MP and one "thread" runs on one SP. From the latest architecture called Fermi, we can invoke another "kernel" before preceding "kernel" invocation finishes.

The parallelism is important to get a good performance in CUDA C. At least, 16 "blocks" for fully use of the MPs and 256 "threads" for hiding the latency of memory accesses are required according to the recommendation from NVIDIA. SPs that belongs to the same MP can cooperate by taking the synchronization and sharing data through the small, high-speed MP-local memory called shared memory. Before Fermi architecture comes out, coalescing the memory access pattern and timing, and using the shared memory efficiently were recognized as the most important factors to get the good performance. However, nowadays importance of these techniques are declining because Fermi architecture has cache memory and the hardware for memory access has been improved.

3. Proposal

Although CUDA C is a high-level language for GPGPU, to achieve good performance is still difficult. Many aspects of GPU can be controlled through CUDA C, but it is achieved by making CUDA C quite close to the hardware. Which of global or shared memory should be used to store each temporal data can be chosen, the number of the "blocks" and the "threads" to execute each "kernel" can be selected, and so on. Programming with care about the hardware architecture is required for the good performance because each language feature has a direct correspondence to a GPU hardware feature. One have to take care about the memory architecture and have to consider how to put the parallelism into practice upon MP and SP.

In consideration of these aspects, a language system which can make it easier to get a good performance with CUDA C is desirable. APTCC is proposed to answer that requirement. APTCC takes a function written in programming language C and translates it to CUDA C. The output of APTCC is source codes that carry out same calculation as the input code, but will be executed using GPU. To generate a high performance code, APTCC has the information about characteristics of GPUs. Making the best use of that information, APTCC generates the CUDA C code.

In translating C code to a CUDA C code, two different questions should be answered. The first question is how to select the code fragments which should be translated into CUDA C code, and the second question is how to translate the selected code fragments into CUDA C. Those two questions are not completely separate questions because the choice of the code fragments for translation should be based on the performance of resulting CUDA C code, but the performance of resulting CUDA C code is depends on how the selected code fragments is translated.

4. Preliminary Implementation

4.1. Selection of code fragments

Given a function definition, the current implementation of APTCC selects nested loops as the targets of translation and generates CUDA C codes for the statements in the body of the outermost loop.

That heuristic is inspired by a typical pattern of computations in scientific simulations and iterative numerical methods. For example, a partial differential equation is solved by iteratively calculating the state of the next time step based on the state of the current time step. Typical iterative numerical methods has a loop whose terminal condition is defined by the precision and the iteration count. Those kind of loops consume main computational complexity in many applications. The current implementation of APTCC detects those kind of loops and focuses to make the computation that is included in those kind of loops faster by using GPU.

Although Fermi architecture can handle the task parallelism , GPU is more suitable for handling data parallelism. Huge data parallelism is required to draw out the good performance of GPU. Thus choosing loops for translation into CUDA C is a reasonable method because huge data parallelism is typically described with loop syntax. According to above consideration, the current implementation of APTCC targets on the nested loop.

Selection of code fragments for translation has huge room for improvement. A loop that includes dependency is not providing the parallelism that is requested by the GPU. Even if a loop do not includes any dependency, the loop that terminates in short time can not be faster if it carries on GPU. To make the number of memory transfer as low as possible is preferable because memory transfer between CPU and GPU is one of the cost. Considering those factors, more sophisticated methods for selecting the code fragments that are carried out on GPU can be developed. This topic is one of our future work.

4.2. Overview of translation

The target loop statement of translation is called **main-for** in this paper. Given a function definition written in C language that should be translated it into CUDA C code, APTCC conducts series of operations, which are classified into two broad classes those related to memory management and to computation.

As for the memory management, APTCC generates codes for variable declaration, device memory allocation, data transfer between the "host" and the "device," and device memory deallocation. The variables that are going to point the GPU global memory address is declared at the top of the translated function. Before the **main-for**, memory areas on the GPU are allocated by estimating the size from the for statements and initialized by transferring the data from the "host" to the "device." The data which is referred on the "host" during the iteration, such as the value used in the

terminal condition of the **main-for** loop, is brought back to the "host" on each iteration. After **main-for**, GPU sends all the data on the memory areas that are initialized before the **main-for** to the "host" memory. The allocated GPU global memory is released at the end of function. Details of the translation will be described in section 4.3.3.

In the translation of computation, APTCC replaces all the statements in **main-for** by "kernel" invocations and generates definition of each "kernel" function. The loops that have only the statements without dependency is translated into the codes that runs in parallel on one GPU. APTCC replaces the loop counter of such loop by the "thread" identifier to parallelize the calculation. In the current implementation of APTCC, substitution statement are also executed on GPU. APTCC generates a tiny "kernel" function that is invoked with a single thread, for each substitution statement.

Before describing the details of the implementation, we define the **simple for-statement** because the for-statement is important to get information for translation.

```
for(i = a; i < b; i+=c) {
    p[i] = d*q[i*w + N];
}
```

If all of `a,b,c,w,N` are the immutable variables in the scope of **main-for** statement, it is called **simple for-statement** in this paper. In other words, there are two requirements to for-statement to claim it is a **simple for-statement**.

1. Only immutable variables are used to describe the initialization of the loop counter and the terminal condition. Let `i` be a loop counter and `c` be an immutable variable, step-form must be one of the following expressions: `i++, ++i, i--, --i, i+=c, i-=c`.
2. Only a combination of the loop counter and immutable variables constructed by arithmetic operators are used to express the dereference. More specifically, array index must be linear to the loop counter.

In the current implementation of APTCC, each pointer variable in a input code must appear in one or more `simple for-statement`s.

4.3. Memory

4.3.1. Declaration of variables on device memory

The pointer variables that refer to allocated memory on "device" are declared at the top of translated code. The type of each declared variable is decided based on the following rule. If a variable is non-pointer type T in the original code, the type of the corresponding variable is T*. Otherwise the variable is declared with the same type to the original variable.

Non-pointer variables that is accessed in the loops that are parallelized within the **main-for** loop must be declared as a "thread" local variables. The declaration and the allocation for those variables are not necessary.

4.3.2. Allocation of device memory

Next, device memory areas are allocated to the declared variables, where the size of each variable must be determined in addition to its type. To allocate a memory area for a variable `var_T` of a non-pointer type T, calling `cudaMalloc` function with the argument `size(T)` is enough.

```
cudaMalloc((void**)&var_T, sizeof(T));
```

However, to allocate memory for a pointer type variable is not simple because the size description of the original variable may not be visible. If the allocation code (`malloc` function call, typically) is visible, then the size description is taken from the allocation code. In the case of a memory area that is given by the callee of the target function through a pointer variable, the size description can not be acquired directly.

APTCC therefore, estimates the size of the allocated memory from its dereferences. Currently APTCC assumes the array accesses appear only within **simple for-statement**s. Consider the following **simple for-statement**.

```
for(i = a; i < N; i++) {
    p[i] = alpha * q[i];
}
```

From the above **simple for-statement**, APTCC gets the size estimations for the arrays `p`, `q` as `N`.

To decide the size of an allocated memory, APTCC first collects the size estimations of all its references as discussed above and constructs the lexical description of each estimation. **Simple for-statements** that appear in the functions which are called in the **main-for** are also the targets of this collection.

Next, APTCC declares an integer array which is initialized by all of these estimations and a integer variable that is initialized by the maximum element of that. The maximum value will be calculated on runtime, as the following illustrates, where `max(int *p, int n)` calculates the maximum value of an integer array `p` of length `n`.

```
int est_p = {N, N/2 +1};
int size_p = max(est_p, 2);
```

After deciding the size of the arrays, `cudaMalloc` function is called to allocate device memory as follows.

```
cudaMalloc((void**)&d_p, sizeof(float)*size_p);
```

4.3.3. Data transfer

Before the **main-for**, APTCC puts codes for data transfers. It is the initialization of GPU global memory.

Some data have to be brought back to "host" in each iteration and some have not. If the terminal condition is given by a kind of norm of a vector for example, the calculation of the norm is carried out on GPU to avoid transfer of the vector data from GPU to CPU in each iteration. However, the use of the norm is on the "host" — to decide whether the iteration should be terminated or not. Only the data that is used at the terminal condition of the **main-for** is required to be brought back to the "host" because whole body of the **main-for** is carried out on the "device" in the output code of APTCC.

After the **main-for**, APTCC transfers the data on the " device" to the "host." Although the optimization that minimizes the size of transferred data is not implemented in the current implementation of APTCC, that memory transfer is not a big overhead in the experiments that are discussed in section 5.

4.3.4. Release

The allocated memory on the "device" must be released at the end of the function. Thus APTCC must put the code that releases all the allocated memory on the "device" at any place where the control can exit from the function.

There are three kinds of places where the control can exit: the end of a function, return statements, and some special functions like `exit`. APTCC puts the release codes at all those three kinds of places.

4.4. Computation

4.4.1. Parallelizable loops

A for-statement without dependency can be parallelized by replacement of the loop counter to a "thread" identifier. Assume that the original for-statement is the following code with the integer variable `n` and float pointers `p`, `q`.

```
for(i = 0; i < n; i++) {
    p[i] = q[i];
}
```

The generated "kernel" function is as following.

```
__global__ void autodefined0(float *d_p, float *d_q, int *d_n) {
    int tid = (((blockIdx.y*gridDim.x + blockIdx.x) *blockDim.z + threadIdx.z)
                    * blockDim.y + threadIdx.y) * blockDim.x + threadIdx.x;
    if(0 <= tid && tid < (*d_n)) {
        d_p[tid] = d_q[tid];
    }
}
```

Where d_p, d_q and d_n are variables on device memory corresponding to p, q and n on the host memory, respectively.

APTCC puts an if-statement wrapping the body of the original for-statement to avoid the illegal access because APTCC may choose the number of the threads different from the loop count n. The "kernel" invocation code for above "kernel" function autodefined0 requires n "threads." But the actual value of n may be unknown till runtime. Thus APTCC chooses the numbers of the "blocks" and the "threads" as follows. The number of the "threads" per "block" is fixed to 256. The total number of "threads" must be larger than the maximum value of loop counter. The number of "blocks" is decided on runtime to meet this requirement. In the case of above example, the number of the "blocks" is (n+255)/256. The "kernel" invocation code is generated as:

```
autodefined0<<<(n+255)/256, 256>>>(d_p, d_q, d_n);
```

The best suitable number of the "threads" per "block" may be different from 256 for some applications. Empirically however, 256 "threads" per "block" gives a good performance for many applications. In the case of a nested for-statement, APTCC parallelizes the outermost for-statement. Parallelizing the inner for-statement may be a good strategy but it is still not available in APTCC.

4.4.2. Function calling

In **main-for**, only two kinds of the functions are able to be called. The first functions are the functions that are built into CUDA C. The second functions are the user-defined functions whose definitions are available. APTCC requires that the second functions are not used in nested loops in the **main-for**. The definitions of the corresponding "kernel" functions must be generated for the second functions. Interprocedural analysis is required to generate this definition. Current implementation of APTCC handles only the two types of the functions that are listed below.

A function of the first type consists of a parallelizable **simple for-loop** such as an addition of two vectors. APTCC replaces the loop counter of the outermost loop by the "thread" identifier. The "kernel" invocation code for this type of function is same as the replacement of for-statement which is discussed in 4.4.1.

A function of the second type consists of a reduction loop such as an inner product of two vectors. In this kind of operation, the result is stored in one scalar variable and the variable is substituted many times. To translate this kind of operation with good performance, APTCC generates two "kernel" functions. The first "kernel" carries out partial calculation on each "block." In each "block," each "thread" issues the unit operation and keeps the result on shared memory. These partial results are aggregated to the first element of shared array by using the Tree-based approach[14]. Finally "thread" which has 0 as its identifier stores this partial result to global memory through the pointer which is provided as an argument. The second "kernel" is given a pointer that points the partial result of the first "kernel" and a pointer to store the result value. This "kernel" will be invoked with one "block" and 1024 "threads" per "block" and aggregates the partial values into one variable by the Tree-based approach[14].

All other kinds of functions are translated into "device" functions which are invoked with only one "thread." In the experimental translation for K-means method described at 5.3, there is a loop that includes nested dereference such as a[b[i]]. This is a concrete example of the function that APTCC generates the "kernel" function that is invoked with only one "thread." This "kernel" is the bottle neck of the translated code in this experiment. (More than 95% of the calculation time is used for this function in this experiment.)

In the case of the function that has a return value, generated function will be added one extra argument. The result value will be stored in the memory address that is pointed by the extra argument, because a "kernel" function in CUDA C can not return any value.

4.4.3. Substitution statement

From a simple substitution statement which does not includes any function calls, APTCC generates a "kernel" function carries out the same operation and that "kernel" function will be invoked with only one "thread." On the other hand, some substitute statements have both operators and function calls on the right hand side of the statement as following.

```
a = b / f(c);
```

This kind of substitution statement is decomposed into two separate statements. The first statement saves the return value of the function into a temporal variable. The second statement is similar to the original statement but the function call is replaced by the temporal variable. The above example will be translated into the following code.

```
tmp = f(c);
a = b / tmp;
```

These decomposition and replacement are applied to the input source code before any other translation processes. In the translation, the first statement is replaced by the "kernel" function as mentioned in 4.4.2 and the second statement is replaced in the same way as the simple substitution case which is mentioned above.

5. Experiments

Five example codes were written for experiments to translate: partial differential equation (PDE), image filter, k-means method, N body problem, and conjugate gradient method. This section gives a brief explanation of each implementation and the performance measurement results of the original code and the translated CUDA C code.

The main purpose here is to compare the performance of the translated CUDA C code to the original C language implementation. Therefore, after confirming the translated code actually gives the same result as the original code, the number of iterations is fixed to smaller numbers.

The experiments was carried out on three different GPUs: Tesla C2050 with 3 GB global memory, GeForce GTX 470 with 1280 MB global memory and GeForce GTX 580 with 1.5 GB global memory. The competing CPU is Core i7 3.33 GHz with 3 GB memory.

5.1. PDE

The first example is a code that solves a two dimensional partial differential equation (PDE). Let i and j, be the grid indices of direction x and y , respectively, k be the index of time step, and N be the length of one side of the square mesh. PDE was discretized as following.

$$u_{i,j,k} = (1-4r)u_{i,j,k} + r(u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+,k} + u_{i,j-1,k})$$

$$u_{0,j,k} = u_{i,0,k} = u_{N-1,j,k} = u_{i,N-1,k} = 0 \quad : \quad \text{Boundary condition}$$

$$u_{i,j,0} = 1 \quad (1 \le i, j < N-1) \quad : \quad \text{Initial condition}$$

The value of r is fixed to 0.2 for this experiment. The iteration count of this experiment is 1000.

The execution time is plotted in Figure 1. Figure 2 shows the transition of proportion in consumed time by each step of this calculation. Each area corresponds to calculating next step, calculating the norm of the area partially, wrap up the partial calculation of the norm, and transfer the value of the norm to the "host," from left to right.

The translation of reduction function which is discussed in 4.4.2 is applied to the calculation of the norm. The calculation time which is consumed by the latter part of norm calculation is constant in this approach so that the third area declines along with the problem size increasing. The calculation time of the former part is increasing as the problem size increasing but there are two factors. If the problem size is smaller than 1024, there are several "blocks" which do not work at all because each "block" owes to one row. Otherwise, each "block" works on one or more rows. This is why the tendency of proportion varies from the problem sizes 1024 to 2048 in Figure 2. The another factor is the number of "threads." If the problem size is larger than 256, each "thread" works on more than one element because one "block" has 256 "threads." Meanwhile if the problem is smaller than 256, some "threads" do nothing.

Comparing with the performance of original code that was executed on Core i7, the translated code shows better performance when the problem size is bigger than 1024. This figure shows that APTCC generates a high performance CUDA C code for PDE code if the problem size is sufficiently big.

5.2. Filter

The second example is a code that performs 9×9 moving average filter to a given picture. First, the summation of the illuminance among surrounding 9×9 square domain of each pixel is calculated. Next, it is divided by the square measure of the domain — 81. The result value is the illuminance for the target point. 9×9 moving average filter is a filter which applies this process to all pixels in the given picture.

This filter is not a iterative method and does not have any condition about precision and iteration. Therefore we wrap whole process of this filter in a for statement that iterates only one time to make fit this program to the requirements of APTCC. So to speak, the iteration count of this experiment is 1.

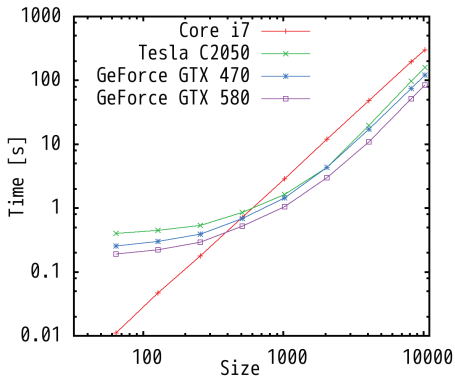

Figure 1: Execution time of the PDE codes. The seconds for different size areas are shown. The iteration count was fixed to 1000.

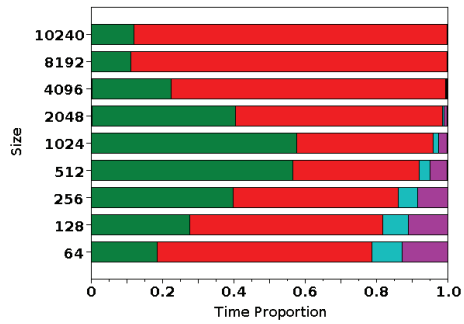

Figure 2: Proportion of consuming time of translated PDE code on GeForce GTX580. The proportion of consuming time for different problem sizes are shown. Each area corresponds to the time for calculating next step, calculating the norm of the area partially, wrap up the partial calculation of the norm, and transfer the value of the norm to "host" , from left to right.

The execution time is plotted in Figure 3. If the size of the given picture is lager than 384, the performance of the translated code is better than the original code. This example shows that not only for a code that implements iterative process such as PDE but also for a code that implements a process runs only once, APTCC generates a high performance code if there is huge data parallelism in the process.

5.3. K-means

The third example is a clustering method called K-means. In this algorithm, the number of clusters, the distance function, and many points are given. The centroid of a cluster is the average point among all the points that belong to the cluster. The goal of this algorithm is to produce the cluster which minimize the summation of distance between points and the centroid of the cluster where the points belong. The terminal condition about precision is given by the number of migrated points from previous cluster. The algorithm terminates when the number of migrated points becomes 0. But the iteration count was fixed to 100 for this experiment, as described at the top of this section, though this algorithm did not terminate in 100 times iteration for any data set.

The execution time is plotted in Figure 4. More than 95% of GPU time was used by the "kernel" function which includes nested dereference. As explained in 4.4.2, this statement is not parallelized at all and the "kernel" function is invoked with one "thread." It means that almost all part of this example is executed with only one "thread" and this is the reason for similarity between the graph of Core i7 and the graphs of GPUs in Figure 4. But the translated code shows better performance.

5.4. N body

The fourth example is an N body simulation. Only the mass,velocity, and acceleration are considered to decide the state on next time step. This example does not require the terminal condition from precision and the iteration count was fixed to 1000.

The execution time is plotted in Figure 5. This result is similar to the case of PDE(5.1). If the size of the problem is big and sufficient data parallelism is available, the translated code gets a higher performance than the original code.

5.5. CG

The last example is the conjugate gradient (CG) method with no preconditioning. CG is a well-known iterative method to solve systems of linear equations with a positive symmetric definite matrix coefficient. This CG code was used in a one dimensional finite element method code as a linear solver. The iteration count was fixed to 1000.

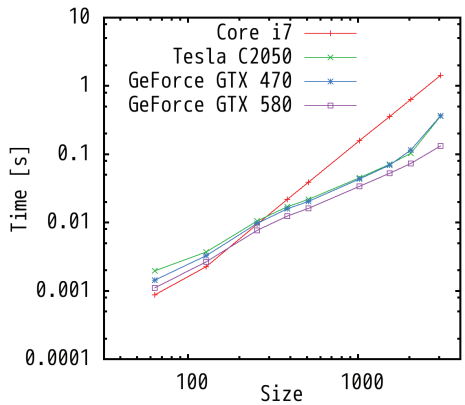

Figure 3: Execution time of the 9×9 moving average filter codes. The seconds for different size pictures are shown.

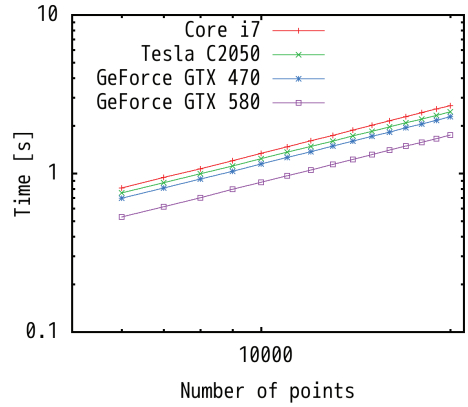

Figure 4: Execution time of the K-means codes. The seconds for different numbers of points are shown. The number of cluster was fixed to 250, and the iteration count was fixed to 100.

The execution time is plotted in Figure 6. The consumed time in the initialization of the array that is used to store the result of the one dimensional finite element method and construction of the coefficient matrix are not included in Figure 6. Only the time which is consumed by CG solver is shown in the figure.

The original source code of this method included a larger number of statements than the other four examples. As a result, the translated code includes more "kernel" invocations than the other examples. According to the figure, the code translated by APTCC has better performance compare with the original code.

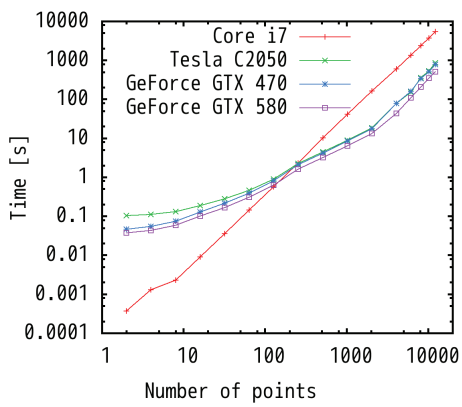

Figure 5: Execution time of the N body simulation codes. The seconds for different numbers of particles are shown. The iteration count was fixed to 1000.

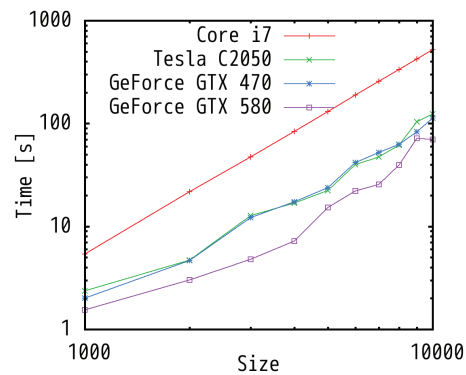

Figure 6: Execution time of the Conjugate Gradient method in a one dimensional finite element method codes. The seconds for different sizes of coefficient matrix are shown. The iteration count was fixed to 1000.

6. Summary

This paper proposes APTCC, Auto Parallelizing Translator from C to CUDA. APTCC assumes the codes written in C language as inputs. Focusing on the nested loop in the codes, APTCC translates the given function into CUDA C code. APTCC puts the device memory management codes and replaces all the statements in the nested loop with the statements that invoke functions which are executed on GPU. Five experiments were performed for the performance

evaluation. For all of those five experiments, the performance of the translated codes that were executed on GPU are better than the original codes written in C language.

7. Future Work

The current APTCC implementation is still under way and there are many respects to improve: improving the usage of shared memory, removing unnecessary codes, concatenating some generated "kernels" and so on. Especially, using multiple GPUs is a big deal. Although it is not supported in the current implementation, using multiple GPUs is important to solve big practical problems.

The original source code written in C language is required to use APTCC. By using the original source code, APTCC can generate a code that selects which processor of GPU and CPU should be used to minimize some objective functions. In particular case, the objective function is time but in next era it can be power consumption. And also, if the cost for communication between the "host" and the "device" is low, using both of CPU and GPU could be better.

The current implementation of APTCC assumes which function should be translated into CUDA C code because to generate the CUDA C code was our first milestone. Telling which part could be parallelized and generating the parallelized code are independent problems. But to distinguish which part of the original code could be parallelized is another interesting problem and if it will be achieved, the translation becomes fully automatic. On that stage, we do not have to learn CUDA C language anymore.

These three topics, to improve the current implementation, to generate the code that CPU and GPU collaborates, and to distinguish which part of source code could be parallelized are our future work.

8. Acknowledgements

This work is partially supported by CREST project ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies of Japan Science and Technology Agency.

References

[1] Kaushik Datta et al., Stencil Computation Optimization and Auto-tuning on state-of-the-Art Multicore Architectures.
[2] NVIDIA, NVIDIA CUDA Linux Release Notes Version 1.0.
URL http://developer.download.nvidia.com/compute/cuda/1_0/CUDA_Release_Notes_linux_1.0.txt
[3] Microsoft, HLSL.
URL http://msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx
[4] The OpenGL Shading Language.
URL http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf
[5] NVIDIA, Cg Documentation.
URL http://http.developer.nvidia.com/Cg/index.html
[6] Khronos OpenCL Working Group, The OpenCL Specification.
URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
[7] CUDA CUBLAS Library.
URL http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf
[8] CUDA CUFFT Library.
URL http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUFFT_Library.pdf
[9] Ian Buck et al., Brook for GPUs: Stream Computing on Graphics Hardware.
[10] Sean Lee, Manuel M.T. Chakravarty, Vinod Grover, Gabriele Keller, GPU Kernels as Data-Parallel Array Computations in Haskell, Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods.
[11] Stefanus Du Toit, Michael McCool, RapidMind: C++ Meets Multicore, Dr. Dobb's Journal (2007) 57–62.
[12] CAPS enterprise, HPMM Workbench - a directive-based compiler for hybrid computing.
URL http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36
[13] NVIDIA CUDA C Programming Guide.
URL http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
[14] Mark Harris, Optimizing Parallel Reduction in CUDA.
URL http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf