SM-Centric Transformation Performance Report

Rodinia Benchmark Suite (version 3.1)

Qiufeng Yu

Department of Computer Science, North Carolina State University, USA

qyu4@ncsu.edu

Abstract

GPUs are massively paralleled processors which are designed to provide large throughput and parallelism. On a typical GPU, there are numbers of Streaming Multiprocessors (SM) and a lot of thread block with each thread block has a large number of threads. When GPU kernel gets to run, each thread block will be scheduled to one of the Streaming Multiprocessors and each thread will be working on one particular task which is determined by the ID of that thread block. By default, the GPU execution model is thread ID centric, which means that it is the responsibility of the thread scheduler (hardware) to decide which thread block will be assigned to which SM at any given time. Since all scheduling is done by the hardware, and the assignment algorithm for the thread scheduler has not been disclosed to the public, it limits many software optimizations. From a programmer's point of view, gaining software-level control of scheduling is important, because it allows us to be able to optimize the program executions via software.

In order to solve the problem introduced above, a more flexible program-level control of scheduling of the GPU processors is needed. The SM-Centric Program Transformation [1] is a technique which can be used to enable and exploit the flexibilities of the task assignment on GPU. It introduced a transformation of the original CUDA code, which centered on Streaming Multiprocessors (SM) to circumvent the limitations of the hardware scheduler, as a result it allows flexible program-level control of scheduling over the GPU processors.

This report will list all the changes that I applied to my SM-centric compiler in order to make it workable for as many benchmarks in the Rodinia Benchmark Suite[2] as possible. In addition, this report will also show the performance before and after the transformation for all the benchmarks that works after the SM-centric transformation.

Categories and Subject Descriptors Programming Languages [*Programming Languages*]: Processors - optimization, compilers

General Terms Performance, Experimentation, Implementation

Keywords GPU, CUDA, Scheduling, Compiler Transformation, SM-Centric Transformation, Clang, LLVM, C++

1. Improvement

The original version of my sm-centric compiler was relatively limited in terms of its abilities to adapt more general CUDA souce code. Although it works fine for the matrix addition and matrix multiplication examples that Dr. Shen provided, it does not work for almost all benchmarks included in the Rodina benchmark suite. In this section, I will list all the changes I made to the compiler, why the original version does not work, and how it works after the modification within different CUDA source code.

1.1 Grid Variable Name

One of the important jobs that smc.h file does is replacing the original grid variable name with "grid" and initiates the three variables with the number of workers needed, the array of the desired sequence of IDs of job chunks, and an all-zero counter array to count active works. The actual implementation of the transformation can be found inside the smc.h file within the __SMC_init() macro. My original compiler works because for the matrix addition and matrix multiplication examples, both examples use the name "grid" as the variable name, as a result, when launching the kernel, "grid" will be the name as the first parameter within the CUDA triple angle brackets. In this case, all I need to do is to replace the grid size definition with __SMC_orgGridDim and the __SMC_init() will do the job for us. However, this is not a general case because most of the programs in Rodinia benchmarks did not use the name "grid" as their gird size variable name, therefore, I need to come up with a more general solution for this replacement. There are several ways of tackling this particular problem, the way I did is as follows.

First, every time I found a function declaration, first check if this function declaration is a CPU function. In other words, does it have a __global__ attribute. If it does not, I can be certain that this function is indeed a CPU function. The reason being is that, CUDA kernel calls are only inside the CPU function. After obtaining the CPU function, I then traverse the AST tree statement by statement to find the kernel function call. Once I found the kernel function call, retrieve the name of the first argument inside the triple angle bracket, which is most likely the grid variable name. Once I have successfully get the grid variable name, I store it in a global string variable and then perform a AST matching on this particular string. The following code will match all variable declarations with the name stored in the "kernel_grid" global variable, and once I found the grid size declaration, I and simply replace its name with __SMC_orgGridDim and keep its arguments untouched.

```
Matcher.addMatcher(varDecl(hasName(kernel_grid))
    .bind("gridcall"), &HandleGrid);
```

One thing needs to be noticed is that, since the grid variable was not named "grid", I also need to rewrite the first argument of the kernel launching statement. It's crucial for our transformation because in smc.h file, the grid variable was named ”grid”, therefore, I have to apply this change to our kernel launching statement as well. The following code illustrates my implementation of the replacement.

```
Rewrite.ReplaceText(grid->getLocStart(),
                    kernel_grid.length(),
                    "grid");
```

1.2 Different grid variable initialization styles

One of the biggest challenges I encountered during this project was that, different benchmarks have their own styles of grid variable initiations. Some benchmarks choose not to use the struct dim3 to declare their grid variable, instead they simply use an integer to hold the grid size, which is a 1-D grid size. For example, in the particle filter program, we have the following grid size declaration.

```
//Set number of threads
int num_blocks = ceil((double) Nparticles /
                (double) threads_per_block);
//KERNEL FUNCTION CALL
kernel <<< num_blocks, threads_per_block >>>
    (arrayX_GPU, arrayY_GPU, CDF_GPU, u_GPU, xj_GPU,
         yj_GPU, Nparticles);
cudaThreadSynchronize();
```

Some programs used dim3 but did not initialize it directly using the syntax dim3 grid(x, y, z). A lot of the time they are more likely to initialize the x, y and z dimensions separately. For example, the lavaMD/kernel/kernel_cuda_cpu_wrapper.cu file:

```
dim3 blocks;
// EXECUTION PARAMETERS
blocks.x = dim_cpu.number_boxes;
blocks.y = 1;
```

Another special case is when the grid size is defined as one of the parameters of its function. An example would be the leukocyte/CUDA/track_ellipse_kernel.cu file as shown below

```
// Host function that launches a CUDA kernel to compute
//the MGVF matrices for the specified cells
void IMGVF_cuda(MAT **I, MAT **IMGVF, double vx, double
     vy, double e, int max_iterations, double cutoff,
     int num_cells) {

// Initialize the data on the GPU
IMGVF_cuda_init(I, num_cells);

// Compute the MGVF on the GPU
IMGVF_kernel <<< num_cells, threads_per_block >>>
             ( device_IMGVF_array, device_I_array,
                 device_m_array, device_n_array,
             (float) vx, (float) vy, (float) e,
                 max_iterations, (float) cutoff );

// Check for kernel errors
cudaThreadSynchronize();
cudaError_t error = cudaGetLastError();
if (error != cudaSuccess) {
   printf("MGVF kernel error: %s\n",
        cudaGetErrorString(error));
   exit(EXIT_FAILURE);
}

// Copy back the final results from the GPU
IMGVF_cuda_cleanup(IMGVF, num_cells);
}
```

As you can see in the code snippets, instead of declaring the grid size, the program directly launches the kernel function and passes one of its function parameters as the size of the grid (num_cells). This is a very special case which I did not find anywhere similar to this in other benchmarks.

These three variations of the grid variable declarations can be handle by my compiler, however, I have to traverse the translation unit declaration more than one time, to be precise, three times. The first time has been explained in the previous section, its main job is to find the name of the grid variable and stores it in a global string variable called kernel_grid.

During the second time of the traversal, our job is to determine whether the grid variable was declared using the dim3 struct, just like the example shows above from the lavaMD/kernel/kernel_cuda_cpu_wrapper.cu file. If the benchmark used the dim3 struct to declare the grid variable, and it did not initialize the grid size directly, then it must have something like grid.x = ... and grid.y = ... in its later source code. In this case, all I have to do is to find the BinaryOperator of grid.x = ... and grid.y = ..., retrieve its left-hand-side and right-hand-side. Once I retrieved both x dimension and y dimension, I will simply insert a new line of code after these two initialization:

```
dim3 blocks;
// EXECUTION PARAMETERS
blocks.x = dim_cpu.number_boxes;
blocks.y = 1;
// *** new grid variable added ***
dim3 __SMC_orgGridDim (dim_cpu.number_boxes, 1);
```

During the second traversal, I also handled the special case where the grid size is directly passed as one of the function parameters. The approach I used to solve this problem was using two global list variables, one to store the parameters of the function and the other one to store the name of the functions who has a kernel function call in them. Each time a FunctionDecl is traversed, I store all its parameters in a global list called parameterList, and each time a CUDAKernelCallExpr is found, I store its parent function name in a global list called functionnameList. Then, for every particular function who has a kernel call, all I need to do is to check if the grid variable name is equal to one of its own parameters. If it is the case, I simply add dim3 __SMC_orgGridDim (parameter name) to the first line of the function. Here is the code after running the compiler:

```
// Host function that launches a CUDA kernel to compute
     the MGVF matrices for the specified cells
void IMGVF_cuda(MAT **I, MAT **IMGVF, double vx, double
     vy, double e, int max_iterations, double cutoff,
     int num_cells) {
// *** added for grid size ***
dim3 __SMC_orgGridDim(num_cells);


// Initialize the data on the GPU
IMGVF_cuda_init(I, num_cells);

// Compute the MGVF on the GPU
__SMC_init();
IMGVF_kernel <<< grid, threads_per_block >>>
            ( device_IMGVF_array, device_I_array,
                device_m_array, device_n_array,
            (float) vx, (float) vy, (float) e,
                max_iterations, (float) cutoff ,
                __SMC_orgGridDim, __SMC_workersNeeded,
                __SMC_workerCount, __SMC_newChunkSeq,
                __SMC_seqEnds);
// Check for kernel errors
cudaThreadSynchronize();
cudaError_t error = cudaGetLastError();
if (error != cudaSuccess) {
   printf("MGVF kernel error: %s\n",
        cudaGetErrorString(error));
   exit(EXIT_FAILURE);
}

// Copy back the final results from the GPU
IMGVF_cuda_cleanup(IMGVF, num_cells);
}
```

For those who have its grid variable declared as an integer, I have to traverse the translation unit declaration for the third time. The reason why I traverse three times is because I have to filter out all the possible variations of the grid variable initialization, it could not be possible for us to determine whether it used dim3 or integer during half of the traversal process, therefore, three time traversal seems necessary.

As I mentioned above, the first traversal is to determine if the program used dim3 and direct grid initialization, for example, dim3 grid(x, y, z). The second time is to figure out if it used dim3 but initialize indirectly, such as the lavaMD/kernel/kernel_cuda_cpu_wrapper.cu example. If these two situations do not apply, I traverse the third time and see if I can find any integer variables which have the same name as the first argument in the kernel launching statement. In this case, much like the second traversal, I simple add an extra line of code below the original grid initialization:

```
//Set number of threads
int num_blocks = ceil((double) Nparticles /
                (double) threads_per_block);

// *** new grid variable added ***
dim3 __SMC_orgGridDim (num_blocks, 1);
//KERNEL FUNCTION CALL
kernel <<< grid /*original grid name changed to grid*/,
     threads_per_block >>>
    (arrayX_GPU, arrayY_GPU, CDF_GPU, u_GPU, xj_GPU,
        yj_GPU, Nparticles);
cudaThreadSynchronize();
```

There are all different kinds of variable declarations, especially for the grid size because it has three dimensions. I am not perfectly sure that my compiler handles all different situations, but it works fine for most of the programs in Rodinia Benchmarks. Some of the situations could not be handled solely by the compiler due to the limitation of the original smc.h file, such as multiple grid sizes in one function, which I will be disscussing about later in this report.

1.3 No Arguments in CUDA Kernel Call or CUDA Kernel Function Declaration

There are some situations where no arguments are provided for some of the kernel calls or kernel function declarations. For example, in heartwall/main.cu

```
// launch GPU kernel
kernel<<<blocks, threads>>>();
```

I have never thought about those cases until I saw some of the benchmarks. This improvement is much easier than the above one, so before I insert the five extra parameters, I check if the kernel call or kernel function declaration has any parameters. By calling getNumArgus() function, it returns the number of arguments for a particular FunctionDecl or CUDAKernelCallExpr. If the return value is 0, instead of inserting

```
Rewrite.InsertText(FTL.getRParenLoc(),
                 ", dim3 __SMC_orgGridDim,
                 int __SMC_workersNeeded,
                 int *__SMC_workerCount,
                 int * __SMC_newChunkSeq,
                 int * __SMC_seqEnds",
                 true,
                 true);
```

I just take the leading comma out so that the dim3 __SMC_orgGridDim becomes the first arguments of the function or kernel call.

2. Working Benchmarks

The Rodinia Benchmark Suite includes a total number of 23 benchmarks for heterogeneous computing infrastructures with CUDA implementations. About half of the benchmarks have a built in kernel performance metrics to be used to evaluate the kernel running time. For those benchmarks that don't have a built in execution time meter, I used the following CUDA built in cudaEvent API to measure the elapsed time of the kernel execution.

```
cudaEvent_t start, stop;
float runningTime;

cudaEventCreate(&start);
cudaEventRecord(start,0);

// LAUNCH THE KERNEL...
// kernelFunction<<< >>>();

cudaEventCreate(&stop);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&runningTime, start,stop);
printf("Execution time : %f ms\n" , runningTime);
```

Rodinia provides a large variety of CUDA benchmarks, domains include medical imaging, bioinformatics, data mining, sorting algorithms, image processing, linear algebra, etc.. I will be showing the performance of the original and transformed CUDA programs in the following subsections, however, due to my limitation of CUDA programming experiences, I may not be able to explain some of the results very well. For example, some of the benchmarks runs significantly faster after SM-centric transformation (more than 10 times faster), which should not be possible. For those strange situations, I double checked my compiler and ensured that the transformation applied to the original code was correct, but it seems that there were still some problems in the transformed kernel function which I could not be able to explain why. Nevertheless, I will try my best to show my findings and explain the reasons behind them. All benchmarks were performed on NVIDIA GeForce GTX TITAN X (Maxwell) GPU.

2.1 Heart Wall

The Heart Wall benchmark is the first one that draws my attention. As you can see in the figure below, in the first iteration, the original programs runs 0.049498 ms and the smc version runs 0.054797 ms, they are pretty close in terms of execution time, given that smc runs a little bit slower than the original version. However, starts from iteration 2 to iteration 20, smc stays at the same execution time whereas the original program starts to run much slower. I think the reason is that, the kernel function call is inside a for loop. Every

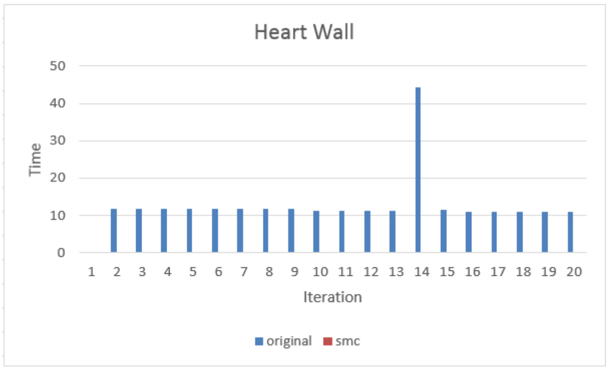


Figure 1. Heart Wall

time the program launches the kernel, it first needs to copy all the data from device to the GPU.

```
// copy frame to GPU memory
cudaMemcpy(common_change.d_frame, frame,
     common.frame_mem, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(d_common_change, &common_change,
     sizeof(params_common_change));
```

However, our __SMC_init() is outside the for loop, so it only works for the first iteration of the loop. Ever since the second iteration, new data are copied to the GPU but __SMC_init() is not called. I think this is the main reason why the smc version takes almost 0 time to run iterations 2 to 20. However the problem is that, if we place the __SMC_init() inside the for loop, smc version will run significantly slower than the original version.

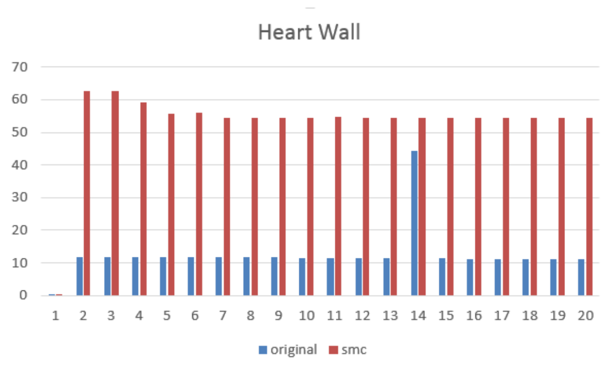


Figure 2. Heart Wall

2.2 CFD Solver

The CFD solver program has four benchmarks: euler3d, euler3d_double, pre_euler3d, and pre_euler3d_double. Those four benchmarks had similar results, so I will only show the result for euler3d in this report, the complete raw data can be found in the raw data folder. The result shows that after smc transformation, the program slows down the performance to about 300%, which is also not quite reasonable.

2.3 LU Decomposition

For the LUD, we encountered the same issue as the heart wall program. Not only the kernel launching statement was declared inside the for loop, the grid size variable was also declared and initialized inside the loop. In order to make the transformation work, the __SMC_init() had to be place inside the for loop this time, because if __SMC_init() was not called after the grid size declaration, it will throw a compilation error indicating __SMC_orgGridDim was not defined.

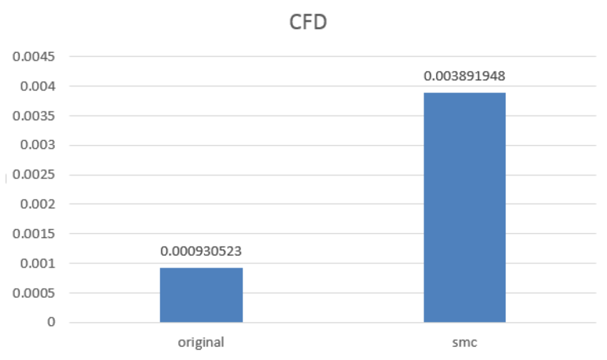


Figure 3. CFD - euler3d

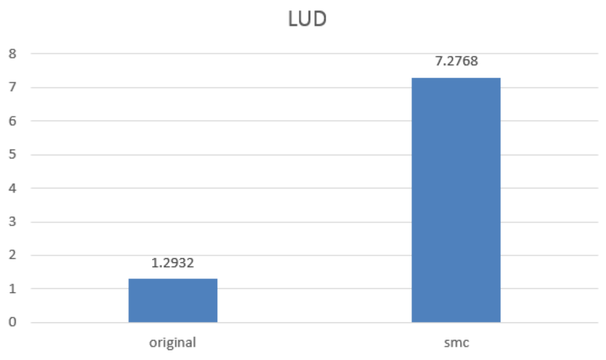


Figure 4. LUD

Since __SMC_init() was placed inside the for loop, execution time was much slower for the smc version than the original version.

2.4 Back Propagation

Back propagation did not have a built-in function to measure the execution time, so I manually added a timer to record the before and after the kernel execution. The result is quite interesting. There are two kernel functions for this program, layer forward and adjust weight. The original version runs faster for layer forward, but much

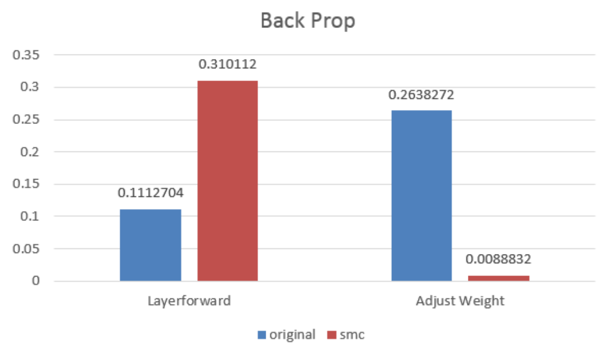


Figure 5. Back Prop

much slower than the smc version when executing the adjust weight kernel function call.

2.5 StreamCluster

For the StreamCluster program, as you can see in figure 6. The original version still out performed the smc version about 50% in terms of running time.

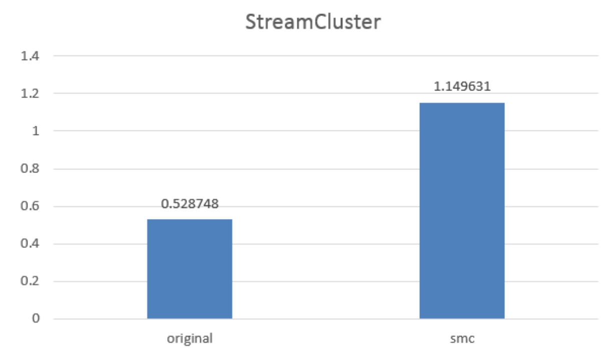


Figure 6. StreamCluster

2.6 PathFinder

PathFinder is a program which uses dynamic programming to find a path on a 2-D grid. The implementation of this program is relatively simple, so my compiler can easily do the correct transformation on the source code. Although it did not have a timing feature, I was able to manually add a timer to track the kernel execution time during each kernel function call. Results are produced as a simple result.txt file, under the working directory, and I was able to checked the result before and after the transformation to make sure the results are correct.

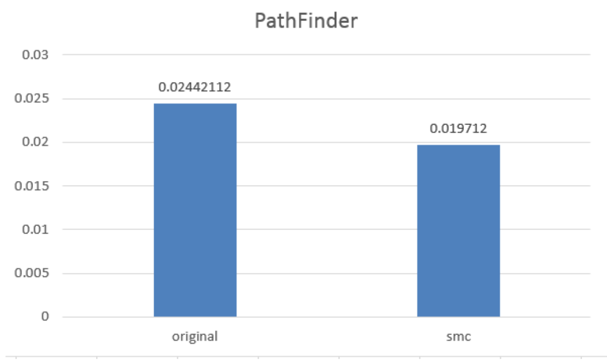


Figure 7. PathFinder

The PathFinder program produced a more reasonable result than most of the other benchmarks, where smc transformed version executed about 23.89% faster than the original program.

2.7 LavaMD

The LavaMD benchmark is a program that calculates particle potential and relocation due to mutual forces betIen particles within a large 3D space. The CUDA version of the LavaMD program defines the grid size as

```
dim3 blocks;
blocks.x = dim_cpu.number_boxes;
blocks.y = 1;
```

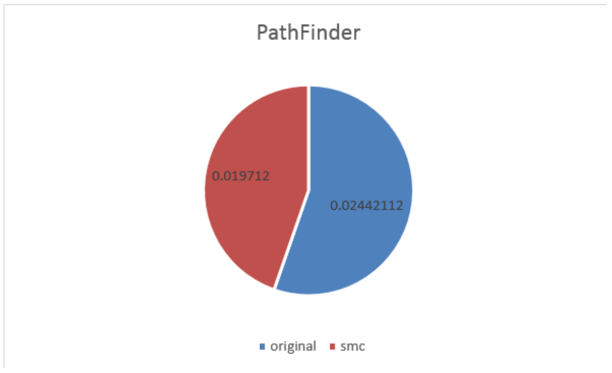


Figure 8. PathFinder

After the transformation, the compiler added an extra line of code in order to transform it to SM-centric form.

```
dim3 blocks;
blocks.x = dim_cpu.number_boxes;
blocks.y = 1;

// smc transformation grid size
dim3 __SMC_orgGridDim(dim_cpu.number_boxes, 1);
// launch kernel - all boxes
__SMC_init();
kernel_gpu_cuda<<<grid, threads>>>( par_cpu,
                        dim_cpu,
                        d_box_gpu,
                        d_rv_gpu,
                        d_qv_gpu,
                        d_fv_gpu, __SMC_orgGridDim,
                            __SMC_workersNeeded,
                            __SMC_workerCount,
                            __SMC_newChunkSeq,
                            __SMC_seqEnds);
```

This program records different states of execution times over the entire period of GPU execution, including the time spend for set device/driver initialization, GPU memory allocation, GPU memory copy time, GPU kernel execution time, time spend for copying memory back to CPU and the time of freeing the GPU memory. In this project, since we only care about the performance of the GPU kernel, therefore, I only collected the GPU kernel execution time. As you can see in figure 9, the original version took 0.1057556 seconds on average, whereas the smc version spent 0.1243388 seconds on average executing the kernel function.

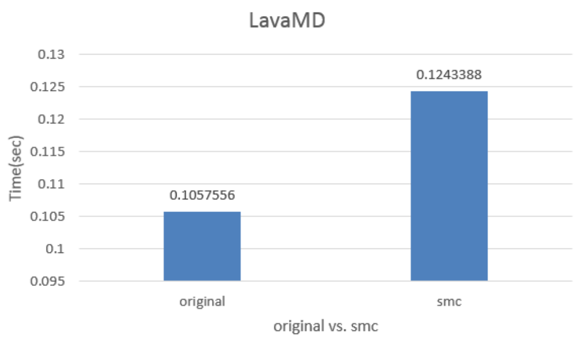


Figure 9. LavaMD

2.8 k-Nearest Neighbors

The NN benchmark finds the k-nearest neighbors from an unstructured data set. Similar to the LavaMd benchmark, NN performs

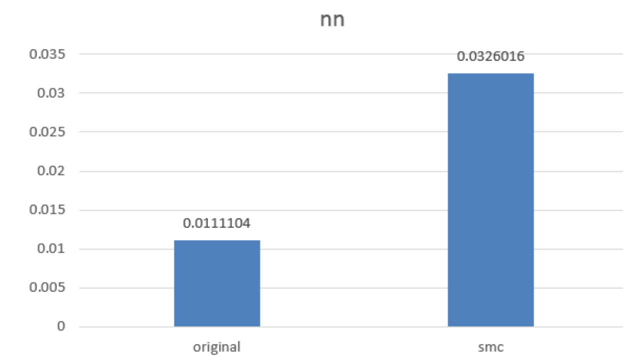


Figure 10. NN

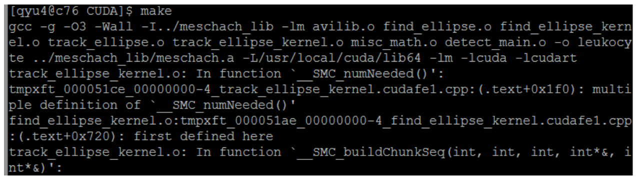
better with the original version. After smc transformation, it runs about 2 times slower.

3. Benchmarks Currently Not Working

After running the smc compiler on all benchmarks included in the Rodinia Benchmark Suite, only a small number of benchmarks worked. Even though the compiler itself can do the correct transformation on almost all benchmarks, some of them eventually turned out to be unable to run correctly. In this section, I will list all the benchmarks that are currently unable to run correctly, reasons that might contribute to the inability of them to run, and some possible solutions.

3.1 Leukocyte, B+ Tree, GPUDWT

The compiler works perfectly on the Leukocyte benchmark, all transformations were done correctly. However, after transforming the code to SM-centric form, I kept getting ”multiple definition of function” errors. I think this problem has to have something



```
[qyu4@c76 CUDA]$ make
gcc -g -O3 -Wall -I../meschach_lib -lm avilib.o find_ellipse.o find_ellipse_kern
el.o track_ellipse.o track_ellipse_kernel.o misc_math.o detect_main.o -o leukocy
te ../meschach_lib/meschach.a -L/usr/local/cuda/lib64 -lm -lcuda -lcudart
track_ellipse_kernel.o: In function `__SMC_numNeeded()':
tmpxft_000051ce_00000000-4_track_ellipse_kernel.cudafe1.cpp:(.text+0x1f0): multi
ple definition of `__SMC_numNeeded()'
find_ellipse_kernel.o:tmpxft_000051ae_00000000-4_find_ellipse_kernel.cudafe1.cpp
:(.text+0x720): first defined here
track_ellipse_kernel.o: In function `__SMC_buildChunkSeq(int, int, int, int*&, i
nt*&)':
```

Figure 11. Multiple definition error

to do with the #include ”smc.h” statement in both of the CUDA files, since the error message indicates that multiple definitions of the __SMC_numNeeded() function was declared. However, in order to perform the transformation, smc.h has to be included in both CUDA files because both files either have kernel function declaration or kernel function call. If we remove the the include statement, the transformation wouldn’t be performed correctly. I’ve also tried to modify the Makefile of this benchmark but still couldn’t able to make it work. B+ Tree has the same problem.

3.2 MUMmerGPU

Compiler works for MUMerGPU benchmark as well, but after transforming the code, I kept getting ”Kernel execution failed: an illegal memory access was encountered.” error right after it started executing round 1 every time I run the program. I double checked the code after performing the transformation, all CUDA files were transformed correctly, it doesn’t seem obvious about why do I keep getting the error messages.

3.3 Needleman-Wunsch, BFS

These two benchmarks have a common problem - the grid variable is assigned inside a loop and it’s value is proportional to the number of iterations of the loop. In order to make them work, we have to place the __SMC_init() and dim3 __SMC_orgGridDim inside the loop. However, if we do this, it will significantly slow down the performance, because every time the loop iterates, the program executes __SMC_init() and dim3 __SMC_orgGridDim, in which a huge amount of overhead will be added to the execution time.

3.4 SRAD, Gaussan Elimination

SRAD and Gaussan Elimination both have multiple grid variable defined within one function. A code snippet is shown below from the Gaussan Elimination benchmark:

```
dim3 dimBlock(block_size);
dim3 dimGrid(grid_size);
//dim3 dimGrid( (N/dimBlock.x) + (!(N%dimBlock.x)?0:1)
     );

int blockSize2d, gridSize2d;
blockSize2d = BLOCK_SIZE_XY;
gridSize2d = (Size/blockSize2d) +
     (!(Size%blockSize2d?0:1));

dim3 dimBlockXY(blockSize2d,blockSize2d);
dim3 dimGridXY(gridSize2d,gridSize2d);

// begin timing kernels
struct timeval time_start;
gettimeofday(&time_start, NULL);
for (t=0; t<(Size-1); t++) {
   Fan1<<<dimGrid,dimBlock>>>(m_cuda,a_cuda,Size,t);
   cudaThreadSynchronize();
   Fan2<<<dimGridXY,dimBlockXY>>>
      (m_cuda,a_cuda,b_cuda,Size,Size-t,t);
   cudaThreadSynchronize();
   checkCUDAError("Fan2");
}
```

As you can see, within one single function, there are two grid dimentions defined, the first one is called dimGrid and the second one is dimGridXY. The problem is that, due to the limitation of the smc.h file, only one grid variable can be rewritten (either dimGrid or dimGridXY) to __SMC_orgGridDim and then rewrite the kernel function call accordingly. However, without modifying the original smc.h file, it seems impossible to have both grid variables transformed to SM-centric form.

3.5 Myocyte

The program is only program that could not be compiled by my smc compiler. None of its CUDA files have a single line of include statement, but all of them depend on the macros defined a the define.c file. I have done a lot of research but still couldn’t find a way to include a C file just like including a .h header file.

3.6 Huffman, Hybrid Sort

These two benchmarks have similar problems. For the Huffman program, in the hist.cu file, all kernel launching statements have a constant as their grid size. For my current implementation of the compiler, I was not able to figure out a method to determine whether the grid variable for a specific kernel launching statement is a constant or a variable. It would be a great feature to add in the future, but due to limited time for this project, this feature has not been implemented yet.

The Hybrid Sort benchmark is not suitable for the SM-centric transformation by my compiler either. The reason being is that, some of the CUDA file has its grid variable changed between two kernel function launch. For example in the bucketsort.cu file:

```
dim3 threads(BUCKET_THREAD_N, 1);
int blocks = ((listsize - 1) / (threads.x *
     BUCKET_BAND)) + 1;
dim3 grid(blocks, 1);
// Find the new indice for all elements
bucketcount <<< grid, threads >>>(d_input, d_indice,
     d_prefixoffsets, listsize);
```

the grid variable was first defined as dim3 grid(blocks, 1), but after executing the kernel function bucketcount, the x dimension of the grid size was changed to another value.

```
grid.x = DIVISIONS / threads.x;
bucketprefixoffset <<< grid, threads
     >>>(d_prefixoffsets, d_offsets, blocks);
```

For a single function, my current implementation only adds __SMC_init() once, right before the very first kernel function call. However, since the program changed the x value of the grid variable, we also have to call __SMC_init() again. In addition, we also need to take loops and if statements into consideration, because we don't want to place __SMC_init() inside a loop. It would take a large amount of time to make the compiler work for this particular situation, which I hope I could be able to implement it in the future.

4. Limitations

There are two main limitations of the improved SM-centric compiler, both of them are related to the grid size initiations in the CUDA code. As I discussed in previous sections, different programmer defines the grid size in different ways. The compiler first only handles grid size defined as dim3 grid(x,y), but I was able to extend it to make it more general for some other variations. The tool currently doesn't work under these two circumstances.

First, if the grid size is a constant, the tool can not be able to do the transformation as expected. The reason behind this behavior is that, currently my tool only tries to find if there is any variable which has the same name as the first argument of the CUDA kernel launching statement. However, if the first argument is a constant, after three traversals of the source code, the tool still cannot find any variables that have the same name as the first arguments in the CUDA kernel launching statement. One possible way to solve this issue could be that, after the last traversal, if we still haven't found any grid variables, we run the ASTMatcher, as you can see in line 494 to 495 of the smc.cpp source code. The class which implemented the MatchFinder::MatchCallback is called GridHandler, and I think I could do something inside this class to add the dim3 __SMC_orgGridDim(whatever constant). I was not able to solve this issue due to the time constrain, but I am confident that it can be resolved in future improvements.

The second issue is that, the tool is not capable of performing the correct transformation is the grid size is initialized more than one time. To be more specific, consider the following pesudocode code:

```
dim3 grid;
grid.x = 5;
grid.y = 10;
kernel<<<grid, threads>>>();
grid.y = 20;
kernel2<<<grid,threads>>>();
```

The problem with this code snippet is that, the grid size was changed between two kernel function launches. Assume we didn't change the value of grid.y before the second kernel launch and we didn't even have the second kernel function launch, then the transformation would become

```
dim3 grid;
grid.x = 5;
grid.y = 10;
dim3 __SMC_orgGridDim(5, 10);
__SMC_init();
kernel<<<grid, threads>>>();
```

The main problem here is that we only want to call __SMC_init() once within the function, but if the grid size have changed after the first kernel launch, we have to add the dim 3__SMC_orgGridDim() and __SMC_init() again, and I am nore sure it is the right SM-centric form.

One final limitation is when running the on some of the benchmarks, some errors about undefined functions occurred. Even though the compiler could still perform the transformation correctly, it is not result that we could like to receive. For example, when I was running the tool on kmeans_cuda.cu, an error message saying "tex1Dfetch function not defined" was produced. I been trying to figure out what could be the possible flags that I could add to the command line, but still haven't found a solution for this. Nevertheless, the compiler was still able to perform the transformation.

5. Conclusion

This project focused on the improvement of my original SMC compiler, making it more adaptable for as many programs as possible, as well as the performance differences before and after the SMC transformation. After this project, the SMC compiler works for most of the cases, with some limitations on a few of the benchmarks. In terms of the performances, most benchmarks slowed down quite a bit after the SM-centic transformation. After doing some research, I think it might have something to do with the smc.h file. In smc.h, the __SMC_workersNeeded was assigned with a value of 2, this value could be more flexible and could be determined by the maximum number of CTAs that can be active. By changing its value from 2 to 50 in the LavaMD program, I see a big performance improvement for the smc version (Figure 12). The execution time shortened from 0.12 second down to 0.0918, with about 15% speedup than the original version

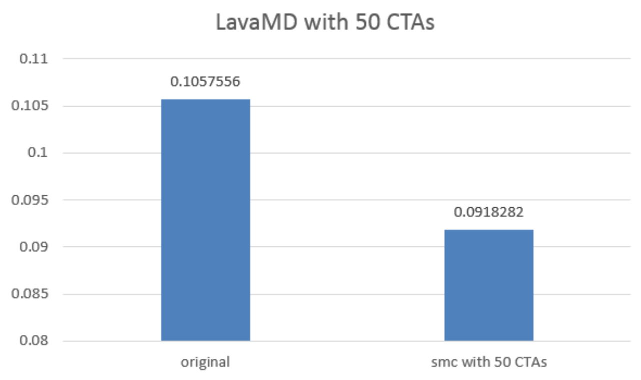


Figure 12. LavaMD with 50 CTAs

References

[1] Bo Wu , Guoyang Chen , Dong Li , Xipeng Shen , Jeffrey Vetter, Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations, Proceedings of the 29th ACM on International Conference on Supercomputing, June 08-11, 2015, Newport Beach, California, USA

[2] http://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators

[3] Bruno Cardoso Lopes , Rafael Auler, Getting Started with LLVM Core Libraries, Packt Publishing, 2014

[4] http://clang-developers.42468.n3.nabble.com/Matching-CUDA-code-td4046484.html

[5] https://github.com/eliben/llvm-clang-samples/blob/master /src_clang/rewritersample.cpp

[6] https://github.com/loarabia/Clang-tutorial

[7] https://stackoverflow.com/questions/27029313/whats-the-right-way-to-match-includes-or-defines-using-clangs-libtooling