Implementation of Adaptive Coarsening Algorithm on GPU using CUDA

Putt Sakdhnagool, Tassapol Athiapinya
<psakdhnagool@cs.ucsd.edu>,<tathiapinya@cs.ucsd.edu>

1. Introduction

In scientific computing today, the high-performance computers grow rapidly along with the data size. The adaptive coarsening algorithm described in [1][2][3] is one of the techniques to deal with this problem. It can compress the data while allowing the data to be manipulated in memory without costly decompression. Its strategy is to produce a non-progressive multiresolution representation, subdividing the dataset into fixed size regions and compressing each region independently. In each region, the complexity of data is analyzed. Then recursively subsample it, upsampling the data and compute the error and continue to subsample it until the error violates the accuracy threshold.

In our project, we implement this algorithm in GPU using CUDA to explore the performance capability of the GPU in terms of both compression factor and the performance throughput. While both are important, we tend to emphasize more on the speed-up because it contributes more to the class and the high-performance computing research lab.

The report is organized in the following order: The AC algorithm, Interpolation, Implementation, Result, Analysis and Discussion, and Conclusion.

2. Adaptive Coarsening Algorithm

Consider the data from Fig.1 which is a solution from Navier Stokes equation, some part of data is smoother than other parts. One type of lossy compression technique called subsampling can apply on this mesh.

![Figure 1: Data from Navier Stokes Equation](image)

Normally, subsampling technique can apply to this mesh by dividing mesh into uniformly pre-defined block size and do subsampling over the mesh. Then try to make subsampling as coarse as possible while maintaining the accuracy of the data. This approach is called Fixed interval subsampling.

With this layout of data where some parts are smoother than other parts, it can be compressed efficiently by applying finer subsampling in the complex area and coarser subsampling in smoother area. The step to determine how coarse the data can be is performed by comparing the error between interpolated data and original data. If the error falls within the acceptable range, it can be compressed. After compressing, we can re-construct the approximated value of original data by interpolation. The pseudo-code of the AC algorithm, taken from [2], appears in Fig. 2

![Figure 2: Left: the AC algorithm. Right: AC divides the mesh into uniform patches of constant resolution. Patches need not all have the same resolution.](image)

### 3. Interpolation

We use two interpolation methods to approximate values during AC algorithm. One is bilinear interpolation which is suitable for diagonal line. Another is bicubic interpolation for curved line. In order to achieve high compression factor, we need to choose the appropriate interpolation method for the particular data. In our CUDA implementation, we try both methods to compare the result.

Bilinear Interpolation

Bilinear interpolation is an extension of linear interpolation for interpolating functions of two variables on a regular grid. The formula for linear interpolation is

\[ S(\delta) = (\delta - 1)f(p_i) + (\delta)f(p_{i+1}) \]

\[ \delta \in [0,1] \]

Where

- \( f(p) \) is a function that returns a value of point \( p \)
- \( \delta \) is the position between point \( p_i \) and \( p_{i+1} \)

The way bilinear interpolation is interpolating is to perform linear interpolation first in one direction, and then again in the other direction.

The formula for bilinear interpolation for finding the value of point \((x, y)\) is

\[ f(x, y) = b_1 + b_2x + b_3y + b_4xy \]

Where

- \( f(m,n) \) is the value of function \( f \) given point \((m,n)\)
- \( b_1 = f(0,0) \)
- \( b_2 = f(1,0) - f(0,0) \)
- \( b_3 = f(0,1) - f(0,0) \)
- \( b_4 = f(0,0) - f(1,0) - f(0,1) + f(1,1) \)

Bicubic Interpolation

Bicubic interpolation is an extension of cubic interpolation for interpolating data points of two dimensional regular grids. For cubic interpolation, it requires 4 points to interpolate the curved line with bicubic interpolation. The estimated curved line is illustrated here.

---

3 http://upload.wikimedia.org/wikipedia/commons/e/e7/Bilinear_interpolation.png
In this implementation we use B-Spline for our bicubic interpolation. The formula for B-Spline cubic interpolation is

\[
S_i(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{bmatrix} \quad t \in [0,1]
\]

Where

\( t \) is position between point \( p_i \) and \( p_{i+1} \)

The bicubic interpolation can be done by performing cubic interpolation for five times. We use four points in the same y-axis to interpolate \( y \). Then, interpolate \( y_{-1}, y_0, y_0 \) and \( y_1 \) to get the interested point. This is the depiction of how to interpolate point \( P \) in 2-D grid.

4. Implementation

In this work, we implement 2 versions of 2D Adaptive Coarsening on GPU. The first version we called “Normal Version”. In this version, the main objective is to implement AC algorithm on CUDA and make sure that it runs correctly. The second version we called “Optimized version”. Because the normal does not give us much speedup as we expected, so we optimize it in this version. The detail implementation

\[ \text{http://www.lsgi.polyu.edu.hk/staff/Eric.Guilbert/images/envconv.bmp} \]
for each version is described in the following section. For each version, we implement 2 interpolation methods, bilinear and bicubic, each method is running separately from other. In this work, we refer to any implementation with bilinear interpolation as “bilinear method” and with bicubic interpolation, we refer as “bicubic method”. For bicubic interpolation, we use b-spine for interpolation model.

4.1 Design Decision

We faced some design issues when implementing AC on CUDA and we made some decision on those issues. The details for each issue and decision are listed below.

Relative Error

We use relative error for determine that patch is acceptable or not. The equation for relative error is

$$Err_r = \frac{|Val_i - Val|}{Val}$$

Where

$Err_r$ is relative error
$Val_i$ is interpolated data
Val is original data

The problems with this issue are

- Val can be 0, which leads to divide by zero problem
- When Val is very small e.g. 10e-7, the relative error is very high. Error of 10e-6 with Val of 10e-7 can leads to relative error of 10 or more, while this error is relatively small compare to acceptance threshold of 10e-3.

These problems can greatly reduce the number of acceptable patch. Our solution to this problem is that we use absolute error ($(|Val_i - Val|)$) instead of relative error for these cases

- Val is 0
- Val is less than 10e-5

Our decision for this issue is that, for Val is 0, there is no other option of error calculation to deal with this problem. For Val is less than 10e-5, because we use the acceptance threshold of 10e-3, the absolute error that is less than 10e-3 should not has much effect on the accuracy of the result.
**Compression Factor Calculation**

In this work, we do not create an output file as our result; the compression factor we got is the calculation from AC result and compared to real file. Our abstract file structure for calculation is:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>First 16 bytes</td>
<td>store dimension of original data (2 long integers)</td>
</tr>
<tr>
<td>Next 16 bytes</td>
<td>store dimension of patch (2 long integers)</td>
</tr>
<tr>
<td>For each patch</td>
<td></td>
</tr>
<tr>
<td>First 16 bytes</td>
<td>is coarsening level on patch (2 long integers)</td>
</tr>
<tr>
<td>Next 8 bytes</td>
<td>is number of stored data N in patch (integer)</td>
</tr>
<tr>
<td>Next N*8 bytes</td>
<td>is stored data (double)</td>
</tr>
</tbody>
</table>

With this abstract structure, we use for calculate compressed file size and use this value to compute compression factor. Note that we do not optimize our file structure because our main objective is not the compression factor.

**Result Data from CUDA kernel**

We try to use less global memory as possible, so our algorithm can accept as large data as possible. Our decision is to load only original data on global memory. The problem occurred when we try to send the result from GPU back to host. Our solution is that, for data that do not presented in file, we replace that data with value -0.0f. Our reasons to this solution are:

- With this method, host code can easily rebuild patch by examine all data in array, so our implementation can actually use in real application.
- Value -0.0f should not appeal in original data because this is not real life value. This value appears because of the structure of floating number format. At first, we have 4 other value to use, NaN, -NaN, Inf, and -Inf but these values need extra calculation, so we decide to use -0.0f which need only assignment statement.

**4.2 Normal Version**

This version tends to implement AC algorithm that runs correctly on GPU, however we make some assumption for implementation. Our assumptions are:

- Use as much shared memory as possible, so the number of time to access global memory is as less as possible. Our decision is we assign each thread block to compute a 32x32 patch, each block contains 16x16 threads, and each thread computes 4 data.
- Branching can greatly reduce performance of the kernel, so we try to reduce number of branching by apply loop unrolling (using #pragma unroll) on every loop that is able to apply.
- Instead of start compression from the finest level, our implementation starts from coarsest level and terminate as soon as the patch is accepted, so we can reduce number of unnecessary computation (compute fine lever while coarser level is acceptable).
Pseudo Code

```c
__shared__ float s_Data[1024]
__shared__ float s_InterData[1024]
For n := 0 to 16; foreach step n += 16
    For m := 0 to 16; foreach step m += 16
        int shared_data_pos = (threadIdx.y + n) * blockDim.x + threadIdx.x + m
        int original_data_pos = (((blockIdx.y * blockDim.y) + threadIdx.y + n) * original_dim) + ((blockIdx.x * blockDim.x) + threadIdx.x + m);
        For i := Max_Sampling_Step downto 2; for each step i = i / 2
            For j := Max_Sampling_Step downto 2; for each step j = j / 2
                For n := 0 to 16; foreach step n += 16
                    For m := 0 to 16; foreach step m += 16
                        int shared_data_pos = (threadIdx.y + n) * blockDim.x + threadIdx.x + m
                        int original_data_pos = (((blockIdx.y * blockDim.y) + threadIdx.y + n) * original_dim) + ((blockIdx.x * blockDim.x) + threadIdx.x + m);
                        //Interpolate data
                        s_InterData[shared_data_pos] = interp2(s_Data,shared_data_pos);
                        calculate error by comparing s_InterData with s_Data
                        __syncthreads()
                        Thread[0,0] check that every error is less than accuracy threshold.
                        if True: write data back to global memory and break
                        else continue running
                Load data from global memory Data[original_data_pos] to s_Data[shared_data_pos] using coalesced loading.
```
Figure 6 Occupancy Information for Normal Version with Bilinear Interpolation
Figure 7 Occupancy Information for Normal Version with Bicubic Interpolation
Profiling Information

Table 1 shows the data from cudaprof, CUDA profiler provided by Nvidia. With this data, we can make sure that all access to global memory is coalesced. We use the data of Shared Memory per Block and Register per Thread field as input for calculates occupancy using Nvidia Occupancy Calculator.

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory Per Block</th>
<th>Register Per Thread</th>
<th>gld_incoherent</th>
<th>gld_coherent</th>
<th>gst_incoherent</th>
<th>gst_coherent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilinear</td>
<td>8744</td>
<td>29</td>
<td>0</td>
<td>16384</td>
<td>0</td>
<td>43264</td>
</tr>
<tr>
<td>Bicubic</td>
<td>8744</td>
<td>55</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 1 Data from cudaprof for normal version

Fig. 6 and Fig. 7 show Occupancy data calculated from Nvidia Occupancy Calculator for bilinear and bicubic interpolation. For both result, the data shows that the program occupancy is limited by number of registers per thread and shared memory usage (Note that, for varying register count chart for bilinear interpolation, there is no triangle mark in the chart because number of register used for this interpolation is 55. It is exceeded the range of the chart provided by Occupancy Calculator). We get only 25% occupancy on both method.

4.3 Optimized Version

In this version we optimize normal version to increase occupancy for our implementation. Our guidelines for this implementation are

- Reduce the registers usage, so we can get better utilization for multiprocessor.
- Reduce the number of branching by removing removable loop from kernel. Our decision is each thread should do less work, so we can remove loop which iterate data from the kernel.
- Use compiler flag –use_fast_math for using fast math library in the kernel.

The result is that we reduce the number of data processed for each thread, from 4 to 1, so the result is

- We use same thread block size (16x16) but for each block, it processes a patch of size 16x16 instead of 32x32.
- Loop that is used for iterating data in each thread is removed.
- Shared memory needed for each thread block is reduce from 8192 bytes to 2048 bytes.

Pseudo Code

```c
__shared__ float s_Data[256]
__shared__ float s_InterData[256]
int shared_data_pos = threadIdx.y * blockDim.x + threadIdx.x
int original_data_pos = (((blockIdx.y * blockDim.y) + threadIdx.y) * original_dim) +
                      ((blockIdx.x * blockDim.x) + threadIdx.x);
```
Load data from global memory Data[original_data_pos] to s_Data[shared_data_pos] using coalesced loading.

For i := Max_Sampling_Step downto 2; for each step i = i / 2
    For j := Max_Sampling_Step downto 2; for each step j = j / 2
        s_InterData[shared_data_pos] = interp2(s_Data,shared_data_pos);
        calculate error by comparing s_InterData with s_Data

__syncthreads()
Thread[0,0] check that every error is less than accuracy threshold.
if True: write data back to global memory and break
else continue running

Profiling Information

Table 2 shows the data from cudaprof running optimized version. We use the data of Shared Memory per Block and Register per Thread field as input for calculates occupancy using Nvidia Occupancy Calculator

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory Per Block</th>
<th>Register Per Thread</th>
<th>gld_incoherent</th>
<th>gld_coherent</th>
<th>gst_incoherent</th>
<th>gst_coherent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilinear</td>
<td>2080</td>
<td>16</td>
<td>0</td>
<td>16384</td>
<td>0</td>
<td>43264</td>
</tr>
<tr>
<td>Bicubic</td>
<td>2080</td>
<td>29</td>
<td>0</td>
<td>16384</td>
<td>0</td>
<td>47424</td>
</tr>
</tbody>
</table>

Table 2 Data from cudaprof for optimized version

Fig. 8 and Fig. 9 show Occupancy data calculated from Nvidia Occupancy Calculator for bilinear and bicubic interpolation. For bilinear method, we can increase occupancy to maximum occupancy and for bicubic method; we can increase to 50% occupancy. The number of used registers is still the occupancy bound of the bicubic method, however we can reduces the number of registers used from 55 to 29, about 47% reduction for used registers. For bilinear method, we can also reduce number of registers used from 29 to 16, about 45% reduction of used register.
Figure 8 Occupancy Information for Optimized Version with Bilinear Interpolation
**Figure 9 Occupancy Information for Optimized Version with Bicubic Interpolation**

- **Varying Block Size**
- **Varying Register Count**
- **Varying Shared Memory Usage**
5. Result

We run our implementation on Lincoln server on Teragrid (See. Appendix A for machine specification). Our sample data is generating from Navier Stokes equation with size of 1024x1024. The result shows in the table below.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Compression Time (seconds)</th>
<th>Speedup</th>
<th>Compression Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>0.2510</td>
<td>1.00x</td>
<td>4.814</td>
</tr>
<tr>
<td>CUDA Implementation (bilinear)</td>
<td>0.0269</td>
<td>9.33x</td>
<td>3.387</td>
</tr>
<tr>
<td>CUDA Implementation (bicubic)</td>
<td>0.0310</td>
<td>8.09x</td>
<td>2.969</td>
</tr>
<tr>
<td>CUDA Implementation (bilinear) – Optimized</td>
<td>0.0032</td>
<td>78.44x</td>
<td>3.884</td>
</tr>
<tr>
<td>CUDA Implementation (bicubic) – Optimized</td>
<td>0.0062</td>
<td>40.48x</td>
<td>3.760</td>
</tr>
</tbody>
</table>

Our optimized implementation gains 78.44x speedup when using bilinear interpolation and gains 40.48x when using bicubic interpolation compared to serial version.

6. Analysis and Discussion

6.1 Speedup

For bilinear interpolation, we got 78.44x speed up compared to serial code and 8.41x speedup compared to normal version. The different between normal version and optimized version are

- The number of data processed per thread is reduced by the factor of 4, so each thread’s work is less by factor of 4, while the number of thread block is increased by the factor of 4.
- In optimized version, we achieve 100% occupancy compared to 25% from normal version; we can more threads in a multiprocessor by a factor of 4.
- We reduce number of branching in optimized version so CUDA kernel can work more efficiency.
- Using fast math can reduce running time but from our test the different of using fast math affect only 10% of running time (without fast math, the running time is 0.0036 second for bilinear interpolation).

From point of occupancy alone, assume that there is no overhead for creating and initializing CUDA thread, we should gain about 4x speedup, so our expectation is that we should gain around 4-5x.
speedup for our optimized version. The extra speedup from 4x speedup is from reduction of branching and using fast math for calculation. From the result, we gain more speedup than we expected.

Unfortunately, by the fact that changing one factor will affect other factor (e.g. removing loop will affect number of data processed per thread) we cannot explain how each factor affects the overall speedup of the implementation. The reasons why we gain more speedup than we expected should be

- We underestimate the speedup for removing branching.
- We missed some speedup factor to compute in our expected speedup calculation.
- We use the different from profiling data and source code, but do not use from compiled code which some optimization from compiler is applied to.

The solution for this need further work, and mostly is out of this work scope. Anyway, one solution we suggested is that compile the kernel code for each version into ptx bytecode, so the different between two versions become clearer.

### 6.2 Compression Factor

We assume that the compression factor of serial code is the best compression factor the mesh should have. For bilinear interpolation, we can nearly achieve the compression factor of serial code; however, as our goal is performance, we think that this compression factor we got is acceptable. The reasons why we have less compression factor than serial version are

- Serial code use more than 1 interpolation to find the best compression for the file, but for our implementation we use only an interpolation method at a time.
- B-Spline has less accuracy than linear interpolation. Even at the subsampled point, the interpolated data does not have the same value as original data.

The different of compression factor between normal version and optimized version is because both versions use different patch size. Optimized version used finer patch size so the probability that each patch is accepted is higher but comes with more overhead, so optimized version get better compression but the different is not too much.

### 6.3 Effect of using –use_fast_math

We use –use_fast_math flag in the optimized version and we compare the effect of using this flag in for bilinear method. The result from profiling shows that without –use_fast_math, the number of registers used for each thread is 18 compared to 16 when using. We can reduce number of register used by 2. Our explanation is that, traditional math library will use extra register for calculation so it can deliver more accurate result, while fast math library does not use those extra register and deliver less accurate result.
The effect on running time is about 10% reduction (using fast math library reduces running time about 4.3e-4 on bilinear method of optimized version). Compression factor is reduced by 0.01, so from our point of view, this is a good trade of between running time and compression.

7. Conclusion

The performance of CUDA implementation can be widely varied depending on the program design. As we have seen in this report, the reduction of number of registers used and branching have dramatic effects on the speedup. The problem is that it is not an easy task to code and optimize the CUDA program. First, it starts from being unable to know the state and variable values of running program in case of verifying the program correctness. Next, to optimize, it requires both skills and time to identify the bottleneck. It could be the global memory access, the bank conflict in shared memory access, the number of registers or the branching. After the bottleneck has been found, the section can be rewritten but it is not obvious to predict the speedup change as one modification sometimes has an impact on many issues.

Although the AC algorithm is suitable to implement in CUDA because the mesh can be divided into equal-sized patches and we can compute each patch individually without side-effect, we have spent a lot of time doing the optimization part for this speedup factor. At last, our work can achieve the reasonable GPU performance with 78x speedup compared to the serial version.

8. Reference


Appendix A: System Information

CPU Information [1]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Intel Xeon Processor E5345</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>4</td>
</tr>
<tr>
<td>Bus Speed</td>
<td>1333 MHz</td>
</tr>
<tr>
<td>L1 Instruction cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L1 Data cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>8MB</td>
</tr>
</tbody>
</table>

GPU Information [2][3]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU processor</td>
<td>Tesla C1060</td>
</tr>
<tr>
<td>Number of GPUs</td>
<td>2</td>
</tr>
<tr>
<td># of Streaming processor cores</td>
<td>240</td>
</tr>
<tr>
<td>Frequency of processor cores</td>
<td>1.3 GHz</td>
</tr>
<tr>
<td>Single precision floating point performance (peak)</td>
<td>933</td>
</tr>
<tr>
<td>Double precision floating point performance (peak)</td>
<td>78</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>102 GB/sec</td>
</tr>
<tr>
<td>Compute capability</td>
<td>1.3</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Multiprocessors</td>
<td>30</td>
</tr>
<tr>
<td>Number of registers per multiprocessor</td>
<td>16384</td>
</tr>
<tr>
<td>Shared memory size</td>
<td>16384 bytes</td>
</tr>
<tr>
<td>Maximum threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum sizes of the x-,y- and z-dimension of a thread block</td>
<td>512,512,64</td>
</tr>
<tr>
<td>Maximum size of each dimension of a grid</td>
<td>65535</td>
</tr>
</tbody>
</table>

Reference


Team Self Evaluation Form

If you are working in a team on a project, then submit one copy of this self-evaluation form. The members of each team should discuss how they worked together and what to write for the evaluation.

(1) List the names of your team members:
A: Putt Sakdhnagool <psakdhnagool@cs.ucsd.edu>
B: Tassapol Athiapinya <tathiapinya@cs.ucsd.edu>

(2) Estimate how much time each team member devoted to this project.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meetings</td>
<td>10 hours</td>
<td>10 hours</td>
</tr>
<tr>
<td>Coding &amp; Optimization</td>
<td>30 hours</td>
<td>25 hours</td>
</tr>
<tr>
<td>Writeup</td>
<td>5 hours</td>
<td>5 hours</td>
</tr>
<tr>
<td>planning (alone)</td>
<td>8 hours</td>
<td>8 hours</td>
</tr>
<tr>
<td>total (including meetings)</td>
<td>53 hours</td>
<td>48 hours</td>
</tr>
</tbody>
</table>

(3) On an attached page, describe (a) what the major responsibilities of each team member were, (b) whether or not you met your milestones (c) the major strengths and weaknesses in how your team worked together, (d) the lessons learned from these events, (e) whether and how you plan to modify your milestones and how your team is organized, and (f) anything else that comes to mind.

a) Division of labor

We spent the time study and discuss adaptive coarsening algorithm. Then, we studied how to implement bilinear and bicubic interpolations because we cannot use GSL library in CUDA. For 2D implementation, at first Putt wrote the fundamental function and using 1D implementation as based code while Tassapol focused on implement bicubic interpolation. Combine both code together we got normal version of code. Before we nearly got the normal version, we contacted Didem to get the serial version. Tassapol worked on compiling the serial code while Putt started doing the optimization. During this step, we shared the idea that could help speeding it up. Later, we worked on the report and the presentation.

b) We met the milestone stated in the progress report.

c) Strength and Weakness

Strength – We have programming experience in C. We also have the CUDA experience in homework #3.

Weakness – We have no experience in Adaptive Coarsening Algorithm and the interpolation methods.
d) We learned the AC algorithm, interpolation methods, program with shared memory in CUDA and optimization in CUDA.

e) We changed one milestone stated in progress report. We do not implement 2D adaptive coarsening using 1D approach because we realized that it cannot generate a correct result. However, we use that time to optimize our implementation instead.