저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)를 이해하기 쉽게 요약한 것입니다.

Disclaimer [↩]
Practical Optimizations for Conjugate Gradient Method Acceleration using CUDA

کوادا را به کار برده، عملکردی عملی برای بهبود سرعت کاهش توانایی کلاس استفاده می‌کنم

کودک را به کار برده، عملکردی عملی برای بهبود سرعت کاهش توانایی کلاس استفاده می‌کنم

2016년 8월

서울대학교 대학원
협동과정 계산과학전공

유 동 한
Abstract

Practical Optimizations for Conjugate Gradient Method Acceleration using CUDA

Donghan Ryu
Interdisciplinary Program in Computational Science and Technology
The Graduate School
Seoul National University

This dissertation presents a series of optimizations for preconditioned and non-preconditioned the Conjugate Gradient (henceforth, CG) method using CUDA. Each lines of CG algorithm has data dependency on adjacent lines but each step is parallelizable operation like matrix-vector multiplication, dot product, and axpy operation. Because each step is well-known parallelizable operation, overall CG algorithm speed can be accelerated by GPUs and meaningful speedup can be seen with the optimization methods presented in this dissertation.

First, we describe performance issues from naïve version of CUDA based CG implemented using an widely adopted CUDA library package: cuBLAS. This library provides generic low level algorithms that can be useful to implement high level algorithms without being focused on writing performant CUDA kernels. However, device-host synchronizations limit the performance gains from CUDA acceleration due to the data dependency of conjugate gradient algorithm steps if that is implemented without a care. GPUs could be
severely under-utilized between each step and GPUs cannot be run at full speed. We proposed a simple but practical optimization technique to avoid device and host synchronizations: Lazy residual evaluation.

In this thesis, the overall runtime performance gain by eliminating device-host synchronizations are explained one by one as the number of synchronizations per iteration is reduced. In the meantime, the changes on CPU and GPU pipeline are explained with illustration as well. Then, the performance gain from the proposed method, Lazy residual evaluation, and advantages or disadvantages are compared against other backend implementations with different level of device-host synchronizations.

Finally, importance of device and host synchronization minimization is expressed in details when accelerating iterative algorithms similar to CG using GPUs.

Keywords: lazy residual evaluation, conjugate gradient method, CUDA

Student Number: 2013-20407
## Contents

Abstract i

Contents iii

List of Figures iv

List of Tables v

1 Introduction 1
   1.1 Iterative Solvers .............................................. 1
   1.2 The Conjugate Gradient Method ............................... 2
   1.3 CUDA ......................................................... 4
   1.4 Performance Issues with the Naïve CUDA/cuBLAS CG .... 5
   1.5 Optimization Methods for CUDA/cuBLAS CG ............... 6

2 Previous Work 8

3 Background 11
   3.1 cuBLAS Device Pointer Mode ................................. 13
3.2 CUDA Dynamic Parallelism .................................. 15

4 Fully Asynchronous CG based on the Dynamic Parallelism 17

5 Lazy Residual Evaluation 19
  5.1 The Modified Secant Method ................................. 21

6 Experiment Results 24
  6.1 Testing Setup ............................................ 24
  6.2 Experiment Results of onepiece_256k ..................... 29
  6.3 Experiment Results of onepiece_118k ..................... 32
  6.4 Experiment Results of onepiece_42k ..................... 35
  6.5 Experiment Results of onepiece_10k ..................... 38

7 Conclusion 41

Bibliography 43

초 록 46
List of Figures

1.1 Illustration of performance issues from the naïve CUDA/cuBLAS CG ........................................... 7

3.1 Illustration of CUDA/cuBLAS based Conjugate gradient method with device pointer mode .................. 14

3.2 Illustration of CUDA/cuBLAS based Conjugate gradient method with Dynamic Parallelism .................. 15

4.1 Illustration of CUDA/cuBLAS based Conjugate gradient method with Dynamic Parallelism .................. 18

5.1 Illustration of CUDA/cuBLAS based CG method with Lazy Residual Evaluation (1) .......................... 20

5.2 Illustration of CUDA/cuBLAS based CG method with Lazy Residual Evaluation (2) .......................... 20

5.3 Illustration of the secant method ........................................ 21

5.4 Illustration of issues of applying the secant method ............ 22

5.5 Illustration of the modified secant method ....................... 23
6.1 (a) Rendered image of onepiece dataset before fitting. (b) Rendered image of onepiece dataset after fitting 28
6.2 onepiece_265k dataset with CG/single precision 30
6.3 onepiece_265k dataset with CG/double precision 30
6.4 onepiece_265k dataset with Jacobi PCG/single precision 31
6.5 onepiece_265k dataset with Jacobi PCG/double precision 31
6.6 onepiece_118k dataset with CG/single precision 32
6.7 onepiece_118k dataset with CG/double precision 33
6.8 onepiece_118k dataset with Jacobi PCG/single precision 33
6.9 onepiece_118k dataset with Jacobi PCG/double precision 34
6.10 onepiece_42k dataset with CG/single precision 35
6.11 onepiece_42k dataset with CG/double precision 36
6.12 onepiece_42k dataset with Jacobi PCG/single precision 36
6.13 onepiece_42k dataset with Jacobi PCG/double precision 37
6.14 onepiece_10k dataset with CG/single precision 38
6.15 onepiece_10k dataset with CG/double precision 39
6.16 onepiece_10k dataset with Jacobi PCG/single precision 39
6.17 onepiece_10k dataset with Jacobi PCG/double precision 40
List of Tables

6.1 Tested CPUs .............................................. 25
6.2 Tested GPUs ............................................... 26
6.3 List of tested solver backends ......................... 27
6.4 List of tested iterative methods ...................... 27
6.5 Tested 3x3 block CSR datasets ...................... 28
Chapter 1

Introduction

Accelerating computation speed of systems of linear equation has been a major research topics for a long time. Over the time, researchers has improved the speed of the system of linear equation solvers by employing three key concepts: (1) iterative methods, (2) sparse matrix, and (3) parallel computing. All three are important to get the best computational performance on modern computers. In this dissertation, our research’s focus is on the minimization of device-host synchronization for the CUDA [9] based conjugate gradient solver that falls into the categories of iterative methods and parallel computing.

1.1 Iterative Solvers

Many direct matrix solvers make use of the LU factorizations of the original matrix [11]. Direct methods produce robust and predictable outputs but the use of direct methods are often limited to small linear systems as the compu-
tional costs of direct methods are too expensive. On the other hands, iterative solvers are usually faster, but the convergence is not always guaranteed, and the performance varies by the condition of the input matrices.

There are two types of iterative solvers: stationary methods and non-stationary methods. The Jacobi method, the Gauss-Seidel method and the SOR methods are well-known stationary methods. These methods are simple but generally considered as less effective than non-stationary methods. Thus, the use of stationary methods are often limited to preconditioners of non-stationary methods. There are wide variety of non-stationary methods developed on mid-20th century and Krylov subspace based methods such as the conjugate gradient method (CG), The biconjugate gradient method (BiCG), and the generalized minimal residual (GMRES) method are the most popular iterative methods. In this thesis, our focus is only on preconditioned and non-preconditioned CG but the proposed technique is not limited to CG and the same principle can be applied to other iterative methods.

1.2 The Conjugate Gradient Method

The CG is an iterative algorithm to compute linear systems with real, symmetric, and positive-definite matrix (SPD) matrices. We choose conjugate gradient method in this thesis because of its simplicity which helps us to identify system level performance issue easier than other more complex iterative methods. The pseudo code of CG used in this thesis can be found from Algorithm 1.
The whole sequence of CG inside the main loop is sequential but the intermediate steps such as the sparse matrix-vector multiplication (SpMV), the constant times vector plus vector (Axpy), and dot product can be well-parallelized and they can be accelerated by massively parallel processors such as GPUs.

**Algorithm 1** Pseudo code of CG algorithm

```plaintext
1: procedure CG(A, b, x, tol)
2:    Ax ← sparse(A) * x           ▷ SpMV Operation
3:    r ← b − Ax                  ▷ Axpy Operation
4:    r2 ← dot(r, r)               ▷ Dot Product
5:    iter ← 1
6:    while r2 > tol * tol do
7:        if iter > 1 then
8:            p ← r + (r2 / r2old) * p    ▷ Axpy Operation
9:        else
10:            p ← r                       ▷ Vector-Vector Operation
11:        end if
12:    Ap ← sparse(A) * p           ▷ SpMV Operation
14:    alpha ← r2 / dot             ▷ Scalar Operation
15:    x ← x + alpha * p             ▷ Axpy Operation
16:    r ← r − alpha * Ap           ▷ Axpy Operation
17:    r2old ← r2                   ▷ Scalar Operation
18:    r2 ← dot(r, r)               ▷ Dot Product
19:    iter ← iter + 1             ▷
20: end while
21: end procedure
```
1.3 CUDA

CUDA is a general purpose parallel computing platform that allows massively parallel executions of programs with large data set. GPUs had been initially developed for graphics computations such as vertex transformations and fragment shading, but they has been evolved from fixed pipeline hardwares to more generalized compute cores to meet the needs from graphics programmers to write their own shaders for different visual effects. The generalization of GPU compute core opened up an oppotunity to use GPUs as an accelerator for non-graphics use cases and CUDA was unveiled on 2008 to meet the industry demands for GPU based parallel computing platform.

cuBLAS  CUDA has very large collections of library [10] and cuBLAS [8] is one of the most popular one that provides highly optimized Basic Linear Algebra Subroutines(BLAS) functionality accelerated by CUDA. cuBLAS allows programmers to implement their parallel algorithm without writing their own CUDA kernel which is often considered hard to write. Initially, cuBLAS routines are only callable from host side but device side cuBLAS call is also supported with CUDA 4.0 and higher. In this thesis, cuBLAS is used for efficient Axpy and dot product computations.

The Block Compressed Row Storage(BCSR)  There are many different sparse matrix representations and each of them has their own strength and weakness. The BCSR is a variant of the CSR format which further increase compression
ratio of the original CSR format by indexing each non-zero 2D block instead of indexing each non-zero element. We choose BCSR for testing as this format is widely used sparse matrix representation.

1.4 Performance Issues with the Naïve CUDA/cuBLAS CG

The naïve CUDA/cuBLAS CG already exhibits higher performance against single threaded and OpenMP based CPU implementations. However, the naïve version does not fully utilize all the power of GPUs because cuBLAS’s API semantics take pointers of input and output scalar as CPU side host pointers by default. For the operations like dot products, the output of a dot product of two vectors is a scalar value and cuBLAS returns the output scalar value to the address of CPU accesible host pointer before the control is returned to the application. This means that CPU thread are blocked in the dot product cuBLAS routine until the outcome of dot product of two vectors is computed from the device and the outcome value is copied over to the host. Because of this, no further execution is scheduled to the GPU and overall parallelism goes down as the system needs to pay the new stream kickoff latency and the memory transfer latency.
Chapter 1. Introduction

1.5 Optimization Methods for CUDA/cuBLAS CG

The latency of building, flushing and scheduling a new command buffer takes several hundreds micro seconds on modern desktop computer systems running on Windows operating systems. Furthermore, each iteration can have multiple GPU kickoffs due to the implicit synchronizations that exist on the naïve CUDA/cuBLAS based CG. Figure 1.1 shows that there are two implicit synchronizations at the end of dot products for one iteration of the non-preconditioned CG. Each synchronization causes runtime overheads, kickoff latency, and memory transfer latency. This makes only small fraction of the GPU pipelines are busy during one iteration of CG. Throughout this dissertation, we removes per-iteration device-host synchronizations one by one until the iteration become fully asynchronous. Then, we propose a technique called Lazy residual evaluation which is not fully asynchronous, but it achieves similar performance level as the fully asynchronous kernel.
Chapter 1. Introduction

Figure 1.1: Illustration of performance issues from the naïve CUDA/cuBLAS CG
Chapter 2

Previous Work

Researchers have investigated techniques to accelerate CG by using GPUs even before CUDA has not been widely adopted. Bolz et al. [2] first mapped a conjugate gradient solver onto the GPU along with a multigrid solver. They used OpenGL textures and fragment shaders to emulate the conjugate gradient method algorithm under the OpenGL graphics pipeline. However, their approach is not generalized and their contribution is limited to the specific implementation of the conjugate gradient method. Kruger et al. [6] then provided generalized linear algebra operators for GPU implementation of numerical algorithms on top of the OpenGL graphics pipeline. Up until CUDA was born, all the GPU based iterative solvers had to be implemented on the graphics pipeline which can easily make the implementation complex and counter intuitive.

After CUDA was introduced, several researchers wanted to create a general
purpose sparse matrix building blocks [5] before widely adopted cuBLAS was released.

Buatois et al. [3] implemented a conjugate gradient method using CUDA and cuBLAS. They also compared the performance characteristics of scalar processing GPU architecture and vector processing GPU architecture and showed that GPU based solvers can solve sparse systems described as block compressed row storage (BCSR) faster than CPU. They introduced CUDA and cuBLAS based conjugate gradient method, but they did not assess GPU utilization during the computation. Also, their study is only limited to only one sparse matrix format which is inefficient on GPU.

Then, Bell et al. [1] looked into a way to optimize sparse matrix-vector multiplication (SpMV) as SpMV is takes most of the time in the iterative solvers. In their study, the performance differences between different sparse matrix data structures were compared, and they discussed the strength and weakness of different sparse matrix formats. The covered sparse matrix formats are diagonal (DIA), ellpack (ELL), compressed row storage (CSR), coordinate (COO), hybrid (HYB), packet (PKT).

After several years later, Xu et al. [12] observed the performance issues with the hybrid format and proposed slightly improved format: the recursive hybrid format (HYB-R). HYB format’s performance is limited by a large number of non-zeros may be left in the COO portion of the HYB format. Also, they showed that HYB’s partition method may not be the best choice. HYB-R increases throughput by putting as many non-zeros possible in the ELL portion
through recursive partition.

Also, Li et al. [7] introduced a new sparse format called Jagged Diagonal (JAD) which can be viewed as a generalized Ellpack-Itpack format. Unlike Ellpack, JAD does not have the assumption of the fixed-length rows that allows JAD to be used on any sparse matrix. In their experimental results, JAD format outperformed scalar and vector CSR formats.

Choi et al. [4] tried to make SpMV faster on GPU by auto-tuning block parameters. As GPU’s occupancy has high impact on the runtime performance, they made a model to automatically the pick block parameters which has high GPU occupancy.

However, there has been little attention on maximizing GPU utilization by minimizing host-device synchronizations when the iterative methods are used. This thesis focuses on the minimization of host-device synchronization.
Chapter 3

Background

BSCR Sparse Matrix-Vector Multiplication  Although most of the vector operations are handled by cuBLAS library, a kernel for the heaviest computation, sparse matrix-vector multiplication(SpMV), needed to be written. Since BCSR format is chosen as our sparse matrix format in this thesis, a SpMV Kernel needed to be written. The actual CUDA codes for BSCR SpMV are shown on Algorithm 3.1. Bcsr read does not provide a nice way to have coalesced memory access on global memory and it is not the most efficient sparse matrix data structure for CUDA. However, it is one of the most widely used sparse matrix format in practice. One of the interesting point to note is the use of __restrict__ keyword on input and output pointers. without __restrict__, the compiler must assume that there could be a pointer aliasing and generate more conservative codes which leads to multiple reads of same global memory address to avoid potential pointer aliasing. By adding __restrict__, compiler
Algorithm 3.1: BSCR SpMV CUDA kernel

```c
__global__ void bscrmv_3x3 (int n, int nnz,
    const int* __restrict__ row_ptr,
    const int* __restrict__ col_idx,
    const float* __restrict__ val,
    const float* __restrict__ x,
    float* __restrict__ out)
{
    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float temp[3];
    float rhs[3];

    if (tid < n) {
        int row_start = row_ptr[tid];
        int row_end  = row_ptr[tid+1];

        temp[0] = 0.0; temp[1] = 0.0; temp[2] = 0.0;

        for (int j = row_start; j < row_end; j++) {
            rhs[0] = x[3*col_idx[j]+0];
            rhs[1] = x[3*col_idx[j]+1];

            temp[0] += val[9*j+0] * rhs[0] +
                       val[9*j+1] * rhs[1] +
                       val[9*j+2] * rhs[2];
            temp[1] += val[9*j+3] * rhs[0] +
                       val[9*j+5] * rhs[2];
                       val[9*j+7] * rhs[1] +
                       val[9*j+8] * rhs[2];
        }
        out[3*tid+0] = temp[0];
        out[3*tid+1] = temp[1];
        out[3*tid+2] = temp[2];
    }
}
```

Algorithm 3.1 is the only sparse matrix format dependent codes in our CG implementation while the rest of codes are sparse matrix format agnostic. That

tries to read the value once and saves the data into local register memory for later use whenever it is possible.
allows CG algorithm implementers to easily switch underlying sparse matrix formats as the rests of the codes are generic and only the SpMV kernel needs to be re-written.

### 3.1 cuBLAS Device Pointer Mode

cuBLAS is a powerful library that uses GPU to accelerate standard BLAS operations. Generally, cuBLAS expects pointers of vectors to be in device memory. However, it expects pointers of scalars to be in host memory. This become a performance issues as CPU thread must waits for GPU to complete certain operations as the output values must be copied back to the CPU accesable memory address before moving forward to the next operation. For CG, dot products must be synchronous calls because of this reason and devices and hosts must be synched each other here.

This performance issue was later resolved by the the release of CUDA 4.0 with a new header, `cublas_v2.h`. The new header provides a new API function, `cublasSetPointerMode(cublasHandle_t handle, cublasPointerMode_t mode)`, to specify which pointer mode for scalars to be used. Two available pointer modes are `CUBLAS.PointerMode.Device` and `CUBLAS.PointerMode.Host`. Because there are two dot products in CG algorithm, there are two places where host and device needs to be synched when host pointer mode is used. However, the first synchronization can be eliminated with the device pointer mode as the output value of the first dot product is streamlined to the next scalar division
Figure 3.1: Illustration of CUDA/cuBLAS based Conjugate gradient method with device pointer mode

kernel on GPU as the cuBLAS dot product routine just writes the result in the device memory which is subsequently read by the scalar division kernel. Figure 3.1 shows how the use of device pointer mode removes one extra synchronization between host and device.

After applying the device pointer mode, the synchronization right after the dot product between p and Ax disappears but the value of the second dot product is the residual and it still has to be copied over to CPU using cudaMemcpy to determine whether more iteration is needed. With this technique alone, CPU and GPU runs slightly more asynchronously and there is less GPU idle time. However, one synchronization is still remaining and this still prevents GPUs to run at full capacity.
CUDA Dynamic Parallelism lets a CUDA kernel to invoke another CUDA kernel. With this new hardware feature, the most of the CG logic can be transferred from the host C/C++ codes to the CUDA kernel codes. Because all the logics are moved to the GPU, there’s no device-host synchronization until the end of all the iterations. This allows GPUs to be almost 100 percent utilized while CPU is completely free to do other things. Since many interactive simulations are followed by realtime rendering code path, CPU can batching up rendering commands while GPU is computing the solution of a linear system. Figure 4.1 is the illustration of the CG using CUDA Dynamic Parallelism. With this feature, the conjugate gradient method can be fully asynchronous.

Although CUDA Dynamic Parallelism provides a way to eliminate per-iteration device-host synchronizations, this feature is only supported with the
hardware that has compute capability 3.5 or higher. Because those devices are relatively new, iterative solver algorithms based on CUDA Dynamic Parallelism cannot be universally used.
Chapter 4

Fully Asynchronous CG based on the Dynamic Parallelism

CUDA Dynamic Parallelism lets a CUDA kernel to invoke another CUDA kernel. With this new hardware feature, the most of the CG logics can be transferred from host C/C++ codes to CUDA kernel codes. Because all the logics are moved to GPU, there’s no device and host synchronization until the end of all the iterations. This allows GPUs to be almost 100 percent utilized while CPU is completely free to do other things. Since many interactive simulations are followed by realtime rendering code path, CPU can batching up rendering commands while GPU is computing the solution of a linear system. Figure 4.1 is the illustration of the CG using CUDA Dynamic Parallelism.

Although CUDA Dynamic Parallelism provides a way to eliminate per-iteration device-host synchronizations, this feature is only supported with the
Chapter 4. Fully Asynchronous CG based on the Dynamic Parallelism

Figure 4.1: Illustration of CUDA/cuBLAS based Conjugate gradient method with Dynamic Parallelism

hardware that has compute capability 3.5 or higher. Because those devices are relatively new, iterative solver algorithms based on CUDA Dynamic Parallelism cannot be universally used.
Chapter 5

Lazy Residual Evaluation

Residual values of all the systems of linear equations that can be solved with the iterative solvers eventually converge to zero although the convergence speeds may vary. Because they are supposed to converge, it can be redundant to evaluate residual every iteration. If per-iteration residual evaluation steps are skipped or done lazily, the computations for multiple iterations can be batched and submitted to GPU all together. That means lower synchronization costs due to residual download and higher GPU utilizations.

The easiest way to skip the residual evaluation is statically performing residual evaluation on every N iterations. For this static method, having too small N yields lower performance due to the frequent synchronizations between host and device. On the other hands, having too large N yields lower performance as well due to over iterations when residual values are very close to the target residual tolerance.
Figure 5.1: Illustration of CUDA/cuBLAS based CG method with Lazy Residual Evaluation (1)

Figure 5.2: Illustration of CUDA/cuBLAS based CG method with Lazy Residual Evaluation (2)

Figure 5.1 shows the CPU and GPU pipelines when residual evaluation is skipped. Then, Figure 5.2 shows the same pipeline when there’s residual evaluation at the end of iteration.
Chapter 5. Lazy Residual Evaluation

5.1 The Modified Secant Method

The secant method is a simple and widely used root-finding algorithm. We adopted the secant method with a modification to estimate when to perform next residual evaluation by knowing the current and previous residuals, and iteration numbers. With the secant method, very large amount of iteration can be executed on GPU without a synchronization by looking at the slope of the convergence rates. Figure ?? illustrates how the secant method is used to dynamically skip per-iteration residual downloading.

However, applying the secant method directly has two practical issues. The issues are illustrated on Figure ???. Because iterative solver’s the slope of the convergence rates at the beginning of the iteration are very steep, almost no iteration batching can be done when the slope is too steep. Thus the benefits of batching at the beginning of the iteration is very small or none. Also, the slope of the convergence rate at the end of the iterations are very flat, there
We mitigate this issue by setting minimum and maximum step size that help to avoid inefficiency happens on both beginning of the iterations and at the end of the iterations. This method is called the modified secant method. If the convergence rate is too high, the next target step size can be almost zero. In order to avoid this, a minimum step size of some small integer number is enforced. Conversely, if the next target step size can be too big, it can also be mitigated by enforcing a maximum step size. Minimum step size of 5 and maximum step size of 50 is used on our experiment. Figure ?? shows how the system behaves after applying the modified secant method on both ends.

By performing residual evaluation lazily with the modified secant method, the synchronization costs due to residual download at the end of the iteration is no longer a major performance issue. From our experiment, close to 100
percent GPU utilization was achieved even on the GPUs that do not support Dynamic Parallelism.

**Algorithm 2** Pseudo code to estimate next residual evaluation point

```
procedure NEXTRESEVALPOINT(prevRes2, prevIter, currRes2, currIter, tol2)
2: slope ← (prevRes2 − currRes2)/(prevIter − currIter)
    stepSize ← roundup((tol2 − currRes2)/slope)
4: if stepSize < minStepSize then
    stepSize ← minStepSize
6: end if
    if stepSize > maxStepSize then
    stepSize ← maxStepSize
8: end if
10: return currIter + stepSize
end procedure
```

The simple pseudo code on Figure 2 estimates next residual evaluation pointer by knowing the residual values of current and previous residual evaluations.
Chapter 6

Experiment Results

6.1 Testing Setup

**Hardwares** We measured the performance of the proposed algorithms and other algorithms on multiple different configurations. CPU backends are tested on 4 different high-end desktop CPUs with different generations. For GPU, 3 different GPUs were used for testing with different size and generations. All the testing was done on Microsoft Windows 10 64bit Operating systems with 12GB system memory.
## Experiment Results

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>i7-860</th>
<th>i7-3770K</th>
<th>i7-4770</th>
<th>i7-6700</th>
</tr>
</thead>
<tbody>
<tr>
<td># of cores</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Cache size</td>
<td>8 MB</td>
<td>8 MB</td>
<td>8 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Max SP Flops/cycle</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max DP Flops/cycle</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Base Frequency</td>
<td>2.8 GHz</td>
<td>3.5 GHz</td>
<td>3.4 GHz</td>
<td>3.4 GHz</td>
</tr>
<tr>
<td>Turbo Frequency</td>
<td>3.46 GHz</td>
<td>3.9 GHz</td>
<td>3.9 GHz</td>
<td>4.0 GHz</td>
</tr>
<tr>
<td>Max SP GFlops</td>
<td>22.4</td>
<td>56.0</td>
<td>108.8</td>
<td>108.8</td>
</tr>
<tr>
<td>Max DP GFlops</td>
<td>11.2</td>
<td>28.0</td>
<td>54.4</td>
<td>54.4</td>
</tr>
<tr>
<td>Max Mem B/W</td>
<td>21 GB/s</td>
<td>25.6 GB/s</td>
<td>25.6 GB/s</td>
<td>34.1 GB/s</td>
</tr>
<tr>
<td>TDP</td>
<td>95 W</td>
<td>77 W</td>
<td>84 W</td>
<td>65 W</td>
</tr>
</tbody>
</table>

Table 6.1: Tested CPUs
## Chapter 6. Experiment Results

<table>
<thead>
<tr>
<th></th>
<th>GTX 780 Ti</th>
<th>GTX 970</th>
<th>GTX Titan X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microarchitecture</strong></td>
<td>Kepler</td>
<td>Maxwell</td>
<td>Maxwell</td>
</tr>
<tr>
<td><strong>Chipset</strong></td>
<td>GK110-400-A1</td>
<td>GM204-200</td>
<td>GM200-400</td>
</tr>
<tr>
<td><strong>Compute capability</strong></td>
<td>3.5</td>
<td>5.2</td>
<td>5.2</td>
</tr>
<tr>
<td><strong># of SMX</strong></td>
<td>15</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td><strong>DP:SP/SM</strong></td>
<td>192:8</td>
<td>128:4</td>
<td>128:4</td>
</tr>
<tr>
<td><strong># of SP processors</strong></td>
<td>2880</td>
<td>1664</td>
<td>3072</td>
</tr>
<tr>
<td><strong># of DP processors</strong></td>
<td>120</td>
<td>52</td>
<td>96</td>
</tr>
<tr>
<td><strong>Base frequency</strong></td>
<td>0.876 GHz</td>
<td>1.050 GHz</td>
<td>1.000 GHz</td>
</tr>
<tr>
<td><strong>Boost frequency</strong></td>
<td>0.928 GHz</td>
<td>1.178 GHz</td>
<td>1.075 GHz</td>
</tr>
<tr>
<td><strong>Max SP GFlops</strong></td>
<td>5045.7</td>
<td>3494</td>
<td>6144</td>
</tr>
<tr>
<td><strong>Max DP GFlops</strong></td>
<td>210.2</td>
<td>109</td>
<td>192</td>
</tr>
<tr>
<td><strong>Max Mem B/W</strong></td>
<td>336.5 GB/s</td>
<td>196+28 GB/s</td>
<td>336 GB/s</td>
</tr>
<tr>
<td><strong>TDP</strong></td>
<td>250 W</td>
<td>145 W</td>
<td>250 W</td>
</tr>
</tbody>
</table>

**Table 6.2: Tested GPUs**
Chapter 6. Experiment Results

Tested Algorithms and Dataset  The testing was done on two different CPU based backends and four different GPU based backends. The list of tested solver backends are listed on the Table 6.3. Also, we used pre-conditioned and non-pre-conditioned conjugate gradient methods for testing and tested against 5 matrices with different size. The smallest size of dataset has matrix block dimension of 10,769x10,769 and the largest size of dataset has matrix block dimension of 265,345x265,345 and each block contains 3x3 elements. The detailed information on the tested matrix size is on Table 6.5.

<table>
<thead>
<tr>
<th>Backend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(omp off)</td>
<td>Single-threaded CPU based solver</td>
</tr>
<tr>
<td>CPU(omp on)</td>
<td>Multi-threaded CPU based solver based on OpenMP</td>
</tr>
<tr>
<td>CUDA(naïve)</td>
<td>GPU based solver which uses CUDA and cuBLAS operated in host pointer mode</td>
</tr>
<tr>
<td>CUDA(device-pointer)</td>
<td>GPU based solver which uses CUDA and cuBLAS operated in device pointer mode</td>
</tr>
<tr>
<td>CUDA(dynamic-parallelism)</td>
<td>GPU based solver which iterative loop run on GPU using Dynamic Parallelism hardware feature</td>
</tr>
<tr>
<td>CUDA(lazy-residual)</td>
<td>GPU based solver which performs residual evaluation lazily</td>
</tr>
</tbody>
</table>

Table 6.3: List of tested solver backends

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Conjugate gradient method without a preconditioner</td>
</tr>
<tr>
<td>PCG(Jacobi)</td>
<td>Conjugate gradient method with Jacobi preconditioner</td>
</tr>
</tbody>
</table>

Table 6.4: List of tested iterative methods
Chapter 6. Experiment Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Size(block)</th>
<th>Size(elem)</th>
<th>NNZ(block)</th>
<th>NNZ(elem)</th>
<th>Tol</th>
</tr>
</thead>
<tbody>
<tr>
<td>onepiece_10k</td>
<td>10,769</td>
<td>32,307</td>
<td>135,801</td>
<td>1,222,209</td>
<td>1e-6</td>
</tr>
<tr>
<td>onepiece_42k</td>
<td>42,709</td>
<td>128,127</td>
<td>545,591</td>
<td>4,910,319</td>
<td>1e-6</td>
</tr>
<tr>
<td>onepiece_118k</td>
<td>118,163</td>
<td>354,489</td>
<td>1,517,129</td>
<td>13,654,161</td>
<td>1e-6</td>
</tr>
<tr>
<td>onepiece_265k</td>
<td>265,345</td>
<td>796,035</td>
<td>3,416,133</td>
<td>30,745,197</td>
<td>1e-6</td>
</tr>
</tbody>
</table>

Table 6.5: Tested 3x3 block CSR datasets

Figure 6.1: (a) Rendered image of onepiece dataset before fitting. (b) Rendered image of onepiece dataset after fitting

All onepiece datasets are the first frame of the same one piece model with different vertex sizes and Figure ?? shows the rendered image of onepiece dataset before and after fitting.
6.2 Experiment Results of onepiece_256k

onepiece_256k dataset is the biggest size dataset among other test data. CPU results shows that the newer CPUs in same performance category shows 20-30 percent of gain on each generation. However, CPU result does not scale well when computation is split among 4 different threads. GPU results shows that the performance of GPU scales as the number of CUDA core goes up. Comparing to the single thread CPU’s performance on Core i7-860 result, GTX-Titan X shows 1,225 percent better performance even without any optimization. With device pointer mode, there were a marginal gain over naïve version. Dynamic Parallelism and Lazy Residual shows 1,484 percent and 1,410 percent performance gain, respectively. The performance impact on device-host synchronization is not very big on onepiece_256k dataset as it takes very long time to go through one iteration on GPU and the cost of device-host synchronization is not big enough comparing to the actual compute cost. Also, GTX-Titan X results shows more performance impact from device-host synchronization than other GPUs as GTX-Titan X is faster but the overall gain is very small.
Chapter 6. Experiment Results

Figure 6.2: onepice_265k dataset with CG/single precision

Figure 6.3: onepice_265k dataset with CG/double precision
Chapter 6. Experiment Results

Figure 6.4: onepice_265k dataset with Jacobi PCG/single precision

Figure 6.5: onepice_265k dataset with Jacobi PCG/double precision
Figure 6.6: onepiece_118k dataset with CG/single precision

6.3 Experiment Results of onepiece_118k

onepiece_118k starts showing the performance difference when device-host synchronization is removed. On the results from CG with single precision, there are around 30 percent difference comparing to the naïve version. Double precision performance gains are much less than single precision gains on this dataset as the ratio of single and double precision CUDA cores are 192:8 on GTX-780 Ti and 128:4 on GTX-970 and GTX-Titan X and the performance is limited by the number of double precision CUDA cores.
Figure 6.7: onepice_118k dataset with CG/double precision

Figure 6.8: onepice_118k dataset with Jacobi PCG/single precision
Figure 6.9: onepice_118k dataset with Jacobi PCG/double precision
Chapter 6. Experiment Results

6.4 Experiment Results of onepiece_42k

As matrix size goes down, the overall performance differences between CPU and GPU become narrower and the performance differences with naïve version and Lazy residual evaluation version become wider. On this dataset, the performance on Lazy residual evaluation mode is noticeably faster than Dynamic Parallelism version.
Figure 6.11: onepice_42k dataset with CG/double precision

Figure 6.12: onepice_42k dataset with Jacobi PCG/single precision
Figure 6.13: onepice_42k dataset with Jacobi PCG/double precision
6.5 Experiment Results of onepiece_10k

When the number of blocks in one dimension goes down to 10,769, the performance of GPUs become similar to the performance of CPUs. However, the trend on the performance different between GPU backend remains similar. Actually, there are almost 2x performance difference from the naïve version. This shows that minimization of device-host synchronization is important when the matrix size is small.
## Chapter 6. Experiment Results

### Figure 6.15: onepice_10k dataset with CG/double precision

![Graph](image)

### Figure 6.16: onepice_10k dataset with Jacobi PCG/single precision

![Graph](image)
Figure 6.17: onepice_10k dataset with Jacobi PCG/double precision
Chapter 7

Conclusion

This dissertation focuses on the importance of minimization of CPU and GPU synchronization for iterative algorithms such as Conjugate gradient method. Using the proposed method, Lazy residual evaluation, we were able to accomplish very high level of GPU utilizations and good runtime performance from wide variety of the GPUs that supports CUDA.

Use of device pointer in cuBLAS assures that there’s no implicit host and device synchronizations between cuBLAS and other kernels caused by the access to the scalar values. This mode allows one iteration of CUDA stream batch to be executed in one kickoffs without being blocked until the point where residual value should be transferred to GPU at the end of an iteration.

However, CUDA stream submission from CPU to GPU still has high latency and reading residual value from CPU on every iteration creates unavoidable synchronization as well. We alleviated this seemingly unavoidable syn-
chronization by checking residual value from CPU on every four iterations. This is possible because residual is supposed to be smaller as more iterations are performed. In real-world interactive use cases, the cost of synchronization is far too much comparing to the execution time of extra iterations. In order to automatically know how many iteration should be executed without a device-host synchronization, we used the modified Secant method. The modified Secant method that we use allows large amount of iteration steps to be taken at once by looping at the slope of the residual squares. With this simple optimization algorithm, we were able to get similar performance level as Conjugate gradient method running solely on GPU using new hardware feature called CUDA Dynamic Parallelism.

As our experimental results showed, The runtime performance of Conjugate gradient method increases as GPU runs more asynchronously from CPU by removing device-host synchronization. Both Lazy residual evaluation and CUDA Dynamic Parallelism achieved almost 100 percent GPU utilization but Lazy residual evaluation can be used on all the GPUs but Dynamic Parallelism based kernel can only be run from latest GPUs.

Since GPU utilization reaches to almost 100 percent, further speedup is possible by optimizing SpMV CUDA kernel as that is the heaviest operations in Conjugate gradient method. In fact, i7-6770’s theoretical single precision throughput is 108.8 GFlops while GTX-Titan X’s theoretical single precision throughput is 6144 GFlops. There are almost 60 times differences in theoretical throughput numbers but only 14 times differences are seen the onepiece_265k
test case. We were not able to get 60 times differences as the sparse matrix format that we use, block CSR, has multiple divergent memory access that prevents coalesced memory access from the threads in a warp. Also, the block CSR format does not allows a nice way to utilize shared memory. Therefore, memory access latency on block CSR SpMV kernel becomes an overall performance bottleneck.
Bibliography


초 록

본 논문은 쿠다(CUDA)를 사용하여 프리컨디셔닝(preconditioning) 및 비 프리컨디셔닝(non-precondition) 된 결레기율기법 방법에 대한 여러가지 최적화 방법들을 제시한다. 결레기율기법 알고리즘의 각각의 단계는 이전 단계에 대한 데이터 종속성이 있지만 대부분 병렬화가 가능한 행렬-벡터 곱셈, 벡터 내적 계산 및 엑스피(axpy) 계산 작업이다. 결레기율기법 계산 단계들내는 일반적으로 병렬화가 잘 된다고 알려져있고, 이 논문에 소개된 최적화 과정을 통해서 씨피유(CPU) 와 비최적화 된 지피유 (GPU) 결레기율기법 대비 상당한 속도 향상을 얻을 수 있다.

이 논문에서는 널리 사용되는 쿠다 라이브러리인 쿠블러스(cuBLAS)를 이용하여 구현한 쿠다 기반 결레기율기법 방법의 성능상 문제에 대하여 설명한다. 쿠블러스 라이브러리는 프로그래머가 최적의 쿠다커널(CUDA Kernel)을 작성하는데 시간을 쓰지 않고 제공된 API를 이용해서 고차원의 알고리즘을 빠르게 구현하는 데 유용하다. 그러나 사용상 주의를 기울여야 하며 정보가 공유될 경우 최적화의 단계에서 완전히 자동화의 제한으로 인해 발생할 수 있다. 장치와 호스트의 동기화에 의한 성능 저하가 결레기율기법 알고리즘 내에서의 데이터 종속성으로 인해 발생할 수 있다. 장치와 호스트간의 동기화는 알고리즘의 각각의 단계에서 지피유 활용도를 저하시키고 전반적인 지피유 가속에서 오는 성능 향상을 저하시킨다. 우리는 본 논문에서 동기화를 피하기 위해 게으른 잔여값 평가(Lazy Residual Evaluation)라는 간단하지만 유용한 방법을 제안한다.

이 논문에서 비최적화 된 지피유 결레기율기법 대비 제안된 게으른 잔
여값 평가 방법의 성능향상에 대하여 단계별로 설명한다. 단계별 설명 중 써피유와 지피유 파이프라인의 변화에 대하여 설명한다. 그리고나서 게으른 잔여값 평가의 성능과 쿠다 하드웨어 능력 3.5 이상에서만 가능한 쿠다 동적 병렬 처리 (CUDA Dynamic Parallelism) 대비 비슷한 성능을 내는 것을 보여준다.

이 논문에서 장치와 호스트간의 동기화를 한계씩 줄여나가면서 전체 실행 속도의 변화에 대하여 설명한다. 그리고 여기서 소개된 게으른 잔여값 평가의 성능과 다른 수준의 동기화 문제를 가지고 있는 구현방법들과의 성능 및 장단점을 비교한다.

마지막으로 쿠데기울기법과 같은 반복 방법을 지피유로 가속시킬 때 장치와 호스트간의 동기화 줄이기의 중요성에 대하여 설명한다.

주요어: 게으른 잔여값 평가, 쿠데기울기법, 쿠다

Keywords: lazy residual evaluation, conjugate gradient method, CUDA

학번: 2013-20407