University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

2014

INTRODUCTION TO PARALLEL COMPUTATION

Clinton McKay

Follow this and additional works at: https://scholarworks.umt.edu/etd

Part of the Higher Education Commons, Numerical Analysis and Scientific Computing Commons, OS and Networks Commons, Other Computer Sciences Commons, Programming Languages and Compilers Commons, and the Science and Mathematics Education Commons

Let us know how access to this document benefits you.

Recommended Citation

McKay, Clinton, "INTRODUCTION TO PARALLEL COMPUTATION" (2014). *Graduate Student Theses, Dissertations, & Professional Papers*. 4366.
https://scholarworks.umt.edu/etd/4366

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.

INTRODUCTION TO PARALLEL COMPUTATION

By

CLINTON ROBERT MCKAY

B.S. Computer Science, University of Montana, Missoula, MT, 2014

Professional Paper
presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

December 2014

Approved by:

Sandy Ross, Dean of The Graduate School
Graduate School

Joel Henry, PhD
Dept. of Computer Sciences

Travis Wheeler, PhD
Dept. of Computer Sciences

Bharath Sriraman, PhD
Dept. of Mathematical Sciences

COPYRIGHT

by

Clinton Robert McKay

2014

All Rights Reserved

McKay, Clinton, M.S, 2014 Computer Science

Introduction to Parallel Computing

Chairperson: Joel Henry

Co-Chairperson: Travis Wheeler

Introduction to Parallel Computing is a course designed to educate students on how to use the parallel libraries and tools provided by modern operating systems and massively parallel computer graphics hardware. Using a series of lectures and hands-on exercises. Students will learn about parallel algorithms and concepts that will aid them in analyzing a problem and constructing a parallel solution, if possible, using the tools available to their disposal.

The course consists of lectures, projects, quizzes, and homework. The combination of these components will deliver the necessary domain knowledge to students, test them, and in the process train them to break a problem down and construct a concurrent solution. The design and layout of the deliverables will follow Blooms Taxonomy [3] and Magers Content Reference Instruction [5] (CRI) model to maximize student retention of the materials.

Delivering the course will be achieved via the iterative development model, often used in software development, but effective in other domains as well. Using the iterative method will aid in the development of robust deliverables that can be extended, replaced, and modified depending on future course requirements.

Contents

1 Overview **1**

2 Requirements **1**
2.1 End User Requirements . . . 1
2.1.1 Content Lectures . . . 1
2.1.2 Testing Components . . . 2
2.2 Developer Requirements . . . 3

3 Design Tasks **4**
3.1 Objectives . . . 4
3.2 Content . . . 5
3.3 Measures . . . 6

4 Implementation Tasks **6**
4.1 Lectures . . . 6
4.2 Quizzes . . . 6
4.3 Homework . . . 7
4.4 Projects . . . 7

5 Deliverables **7**

6 Process of Execution **8**
6.1 Development Process . . . 8
6.2 Method of Implementation . . . 9
6.2.1 Research (Requirements Analysis) . . . 9
6.2.2 Draft (Design & Implementation) . . . 9
6.2.3 Testing . . . 9
6.2.4 Evaluation . . . 10
6.3 Revisions . . . 10

7 Lessons Learned **10**
7.1 Project Deadlines & Deadline Difficulty Assessment . . . 10
7.2 Resources . . . 11
7.3 Documentation . . . 11
7.4 Lectures . . . 11
7.5 Iterations . . . 11
7.6 Development Tools . . . 12

8 Conclusions **12**

9 Appendicies **13**
9.1 Slides . . . 13
9.1.1 GPGPU Lectures . . . 13
9.1.2 Concurrency Lectures . . . 55

9.2 Projects . . . 78
9.2.1 Matrix Multiplication . . . 78
9.2.2 Radix Sort . . . 98
9.2.3 Parallel Logger . . . 128
9.3 Homework . . . 148
9.3.1 Intro to C++11 Threads . . . 148
9.3.2 Inter-Process Communication . . . 155
9.3.3 CUDA Radix Sort . . . 161
9.3.4 CUDA Scanning . . . 186
9.3.5 CUDA Reducing . . . 205
9.4 Quizzes . . . 218
9.4.1 CUDA Quiz Questions . . . 218
9.4.2 Concurrency Quiz Questions . . . 228

10 References 234

1 Overview

Introduction to Parallel Processing is a course designed to aid students in comprehending parallel computation models and to utilize them to their advantage while at the same time provide a general toolset for selecting and using concurrency models to solve real-world problems. The course covers two computational models: General Purpose Graphics Processing Unit (GPGPU) computing and classic threading. The GPGPU computing section teaches students the architecture of the GPU, what types of problems it best solves, and how to make use of the tools available. The second component covers threading models and parallel processing using utilities provided by Linux operating system. Overall, the intent of the course is to provide domain knowledge, use cases, and the implementation skills to students such that they can analyze a problem, select the appropriate model, and then implement a solution.

2 Requirements

2.1 End User Requirements

The end user requirements will contain most of the components required for a functioning course. The components include: lectures, reading materials, and testing materials. Each item contains a simple description of how it will be applied. Individual instructors may add to or revise course materials as needed.

2.1.1 Content Lectures

The lectures provide content associated with each parallel model. Content includes but is not limited to; explaining the design of the model, applying the model to the appropriate problem, domain knowledge of the model, and use cases.

Reading Material: If necessary there will be reading sources associated with the

content taught in the course. Reading materials will be mentioned in the lecture slides.

2.1.2 Testing Components

To properly evaluate student performance, Bloom's Taxonomy model [3] will be used to monitor progress in four categories of the cognitive domain. Given the scope of the course the *Synthesis* and *Evaluation* domains will be left to the discretion of the professor.

Knowledge: Quizzes test students in the knowledge category. Relative to this course quizzes test the students' retention of lecture materials and their associated reading materials.

Comprehension: Homework problems cover lecture materials so that reinforce the students' comprehension of the topic and aid them in solving the projects. A subset of the quiz questions will also fall into this category; depending on the difficulty of the questions.

Application: Projects will test the students' ability to implement a computational model. Implementing the model will require the students to make use of the tools within the parallel computation category.

Analysis: Projects also represent the highest level of comprehension. They will test the students' knowledge of advanced concepts as they must draw from the components learned from the homework and lectures, and then assemble them to construct algorithms or larger systems with complicated interactions.

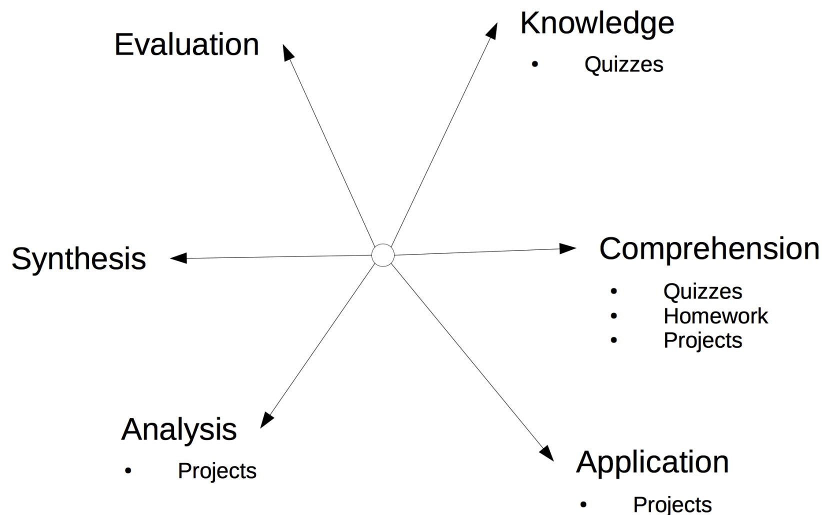


Figure 1: Blooms Taxonomy

2.2 Developer Requirements

1. Build a set of lectures for the class. The lectures themselves will be written in a pandoc-compatable markdown syntax.

2. List a set of reading materials that complement the class presentation materials.

3. Construct a set of questions for each lecture that can be easily transferred to a testing medium such as Moodle. The questions will be formatted in the General Import Format Template (GIFT) file format.

4. Specify a set of homework questions that will contain smaller scoped problems.

5. Construct a set of questions such that there are at least 70 questions for the GPGPU and threading sections of the course combined.

6. Construct projects that allow students to get hands-on experience with the different tools associated with each parallel computation paradigm. The projects contain instructions, input data, and desired output data.

3 Design Tasks

The course has been designed so that the instructor has a fast and efficient means to teach the material. Reaching this goal has required implementing to Robert Mager's Criterion (CRI) framework [5]. The CRI framework categorizes a course into three main components; objectives, content, and measure.

3.1 Objectives

The are two sets of objectives for the students.

1. The first group of objectives consist of the non-terminal objectives for the students. The objectives have been designed so that students gain a broad perspective of the different parallel programming models.

    (a) Understand the domain knowledge and terminology used in each model. Students should be able to use the vocabulary and understand how each model works such that they can effectively communicate among peers.

    (b) Learn what each model is best suited for and know which problems it can be applied to.

    (c) Learn about the architecture of each model and how it works.

    (d) Learn about real world use cases.

    (e) Become familiar with a set of the parallel libraries available.

2. The terminal objectives have been designed to ensure that students are capable of implementing a program using at least one of the parallel models. These objectives for the students include being able to:

   (a) Determine which model is the most effective when solving a particular problem.

   (b) Implement a program that follows the chosen model.

   (c) Determine if a library is needed and then include the library infrastructure in their program.

   (d) Understand the executable behavior of the application and then performing a short analysis of the strengths and weaknesses of their program.

3.2 Content

Content delivery occurs through the lectures and the assigned reading material. The lectures have been designed to present the domain knowledge of each parallel model, which includes: terminology, use cases, algorithm explanations, and architecture details. Reading materials serve as reference and exploratory tools to aid in learning course objectives presented in the lectures. The combination of the two delivery mediums ensures that the non-terminal objectives in group 1 are satisfied.

Content related to the terminal objectives (objective group 2) consists of the projects and homework. The projects and homework have been designed so that students have to satisfy at least one of the terminal objectives. Students are required to analyze a problem, implement a solution, or design a solution. Designing a solution would consist of a verbal or written statement that explains a potential implementation path.

3.3 Measures

Unlike the lectures, homework and projects test the students' progress. Besides implementing a program, the structure, output, and completion of the program measure students' performance and understanding of the terminal objectives. Feedback can then be used to modify the course structure to improve the students' attainment of course objectives. Homework assignments are similar to projects but their primary focus is testing objectives 2a, 2b, and 2c using a more open ended question approach that requires the student to analyze problems and explain their solutions. Quizzes have been designed to test non-terminal objectives 1a, 1d, and 1e. Quizzes consist of multiple choice, matching, and true/false questions that are designed to test the students' retention of the domain language and the architecture.

4 Implementation Tasks

4.1 Lectures

The lectures consist of a set of text files written in the pandoc-compatible markdown that can be compiled into PDF slides. Each lecture covers an important subtopic of each of the parallel computation paradigms. Lectures had to be delivered so that they cover the topics at a high level and they establish a common relationship between the projects and objectives presented.

4.2 Quizzes

The quizzes for the course are delivered as a set of text files conforming to the General Import Format Template (GIFT) format. The GIFT format allows questions to be imported into Moodle and is supported by other online educational systems. Another advantage is that the GIFT format is easy to read without special software, thereby

enabling portability.

4.3 Homework

Homework assignments require no more than a couple of hours to complete and prepare students for the projects. Homework will contain a set of instructions, at least a single file with incomplete source code, and tools to build it.

4.4 Projects

Projects take a more active approach to learning the paradigms within the course. Projects consist of instructions, incomplete source code that compiles without errors, and testing utilities. Students must utilize the parallel programming concepts learned in the course to complete the assignment. The score the student receives will be based on the output of their solution, code implementation, and documentation.

5 Deliverables

1. Two sets of lecture materials each consisting of at least 60 to 120 slides. One set of the lectures will be for CUDA while the other will be for general threading.
2. A set of 70 to 100 quiz questions.
3. Three to four homework assignments.
4. Three projects.
5. List of complementing materials (books, journal articles, and API documentation).
6. A Git repository of all of the deliverables and course components.

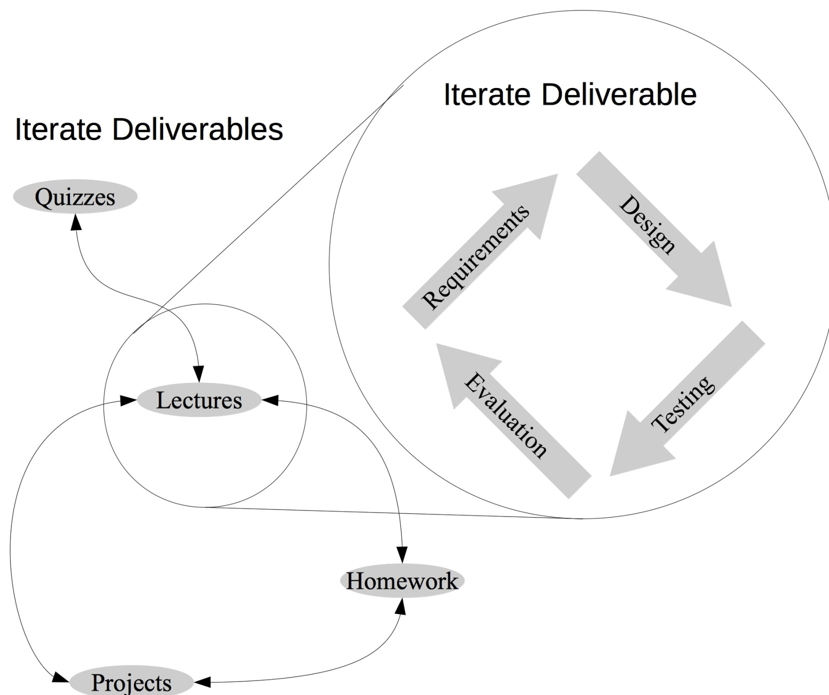


Figure 2: Iterative Design Process

6 Process of Execution

6.1 Development Process

To stay on schedule and produce high quality deliverables. The iterative development process was used to analyze the requirements, design and implement, test, and then evaluate the deliverable. The complexity of the deliverable (and its dependencies) would affect the iteration count. In conjunction to the iterative process a development hierarchy was also utilized. The hierarchy is as follows: slides, homework, projects, and quizzes (see figure 2).

6.2 Method of Implementation

6.2.1 Research (Requirements Analysis)

The first of the three steps involved researching and determining the course objectives the materials should cover. Once the course objectives were identified, they were laid out in a logical manner and then further subdivided.

6.2.2 Draft (Design & Implementation)

Once the objectives had been constructed, they were organized sequentially and expanded to create the lecture materials. During the expansion phase, development would shift to the other deliverables, iterate, and then shift back to the objective expansion. This constant shifting ensured completeness, consistency, and enforced a common relationship amongst the deliverables.

6.2.3 Testing

Test the deliverables and check for program errors, spelling mistakes, and other areas that could be improved. Testing varied based on the type of deliverable. Testing lectures and quizzes involved looking for spelling mistakes and invalid explanations that would need revision or objectives that had been missed. The testing involved with homework and projects was considerably more involved. To ensure that the concepts were explained properly a solution to each homework had to be constructed in order to test the mechanics of the parallel paradigm and update the slides accordingly. To ensure program accuracy for both homework and projects, cuda-gdb and gdb was used to debug and analyze the behavior of the program. Projects were then tested further using Perl scripts that would compare the expected output to the program's output and report errors. These same scripts will also be used to provide students with a base to validate the accuracy of their program.

6.2.4 Evaluation

Evaluation was treated similar to testing. The major difference was presenting the deliverable to my advisors and obtaining feedback, where it was further recorded into the GitHub project management system. The evaluation of the deliverable also served as a decision criteria to either reiterate over the deliverable or mark it as completed.

6.3 Revisions

Project revisions were tracked with Git. Git is a software versioning system that allows all changes to be documented. In conjunction to documenting changes, Git's branching system allowed the development of deliverables to be iterated and tracked independently, thereby allowing the continuous revisioning and advisor collaboration.

7 Lessons Learned

7.1 Project Deadlines & Deadline Difficulty Assessment

Managing project deadlines and deliverables proved to be a challenging task. It is crucial to find a tool that permits the organization of project deliverables, deadlines, and issues. Relative to this project I decided to use GitHub's project management features which provides the facilities for milestone management and issue tracking. Creating issues and associating them with a milestone allowed the assessment of project deliverables and estimation of deadlines.

Using the proper tools will always help when assessing deliverables and deadlines, but when I combined it with feedback from my advisors it proved to be much more effective. Properly managing a project requires constant milestone evaluation and communication amongst peers.

7.2 Resources

Effective project management also requires efficient resource tracking. First, keep track of all of the resources that have been used. Second, make sure that all of the resources needed are available. Failure to do so can cause project deadline miscalculations resulting in unfavorable delays.

7.3 Documentation

Documentation is crucial. Throughout the project I documented my progress via Git commits, lecture notes, README files, and kept track of all of the resources that I used. Doing so has saved time on numerous occasions. For example, when I had to go back and revise components or realized that a source found earlier could be of use, such documentation proved pivotal. Whenever working on a project, make sure to always make note of any sources or ideas that may prove resourceful later.

7.4 Lectures

While not directly related to the project's management, designing lectures that were dense in content but did not overwhelm the students was a challenge. Lectures must start out simple and then build upon previous slides in later slides so that there are no knowledge gaps and more complicated concepts can be explained. With this in mind, future projects or deliverables that require training will make use of the same concepts to maximize student learning.

7.5 Iterations

Iterating is crucial for any design process. Designing a project without iterating over it and revising it will result in deliverables that are of poor quality and defective. The development of my project helped me understand the importance of the research,

implementation, testing, and evaluation cycle. During this project, I learned how to effectively iterate the deliverables and improve them until completion. Any medium to large scale project should include some form of iteration.

7.6 Development Tools

When learning a new framework or language, it is critical to understand how the development tools function and learn to use them correctly. Relative to the projects, CUDA's cuda-gdb and gdb proved to be versatile when testing and debugging the homework and project deliverables. CUDA required remote programming via SSH since I didn't have dedicated hardware to run cuda-gdb. Understanding the basics of compiling debuggable code and running it through the debugger saved countless hours; Learn how tools function without the help of an IDE.

8 Conclusions

The development of this project has served as a learning platform to project management and most importantly parallel computing concepts. During the development lifecycle of the project I was able to expand my knowledge of parallel computing using modern hardware. The concepts behind concurrency are well founded but at the same time can be applied to bleeding edge technology such as the GPGPU. I was able to grasp a better understanding of parallel computing and still be able to learn modern technologies. In conjunction to learning new computing techniques and technology. The project served as a basis to expand my knowledge of project management and development using advanced tools such as Git, GitHub, gdb, and Pandoc.

I would also like to provide special thanks to Dr. Joel Henry, Dr. Travis Wheeler, and Dr. Douglas Raiford for support and feedback during the entire development of the project.

9 Appendicies

9.1 Slides

9.1.1 GPGPU Lectures

Massively Parallel Hardware

January 8, 2015

Introduction

An introduction leveraging the computing capabilities of the graphics processing unit to parallelize a task.

Problem

- Why would we want to use a GPU?
- For example lets say an assignment required that you write a function that cubes each element in a array X.
  - $f(X, p) = Y$ such that $X = [x_0, x_1, ...x_{n-1}]$ and $Y = [x_0^3, x_1^3, ...x_{n-1}^3]$.
- X contains at least a billion integers and performance is a requirement.

Serial Example

- Up to this point most of your assignments involved writing code that would execute in a serial fashion.

```
#define LENGTH 1000000000 // 1,000,000,000
int main()
{
  int ary[LENGTH] = {2, ..., 2}; // assume that array is initialized with 2s.

  int tmp = 0;

  // At least ten CPU cycles per iteration.
  for(int i = 0; i < LENGTH; i++) {
    tmp = ary[i];
    ary[i] = tmp * tmp * tmp;
  }

  // About 1,000,000,000 * 10 cycles. ~3 seconds on a 3GHz processor.
}
```

Serial Example

Advantages

- Easy to write
- Portable

Disadvantages

- Even though it is a simple operation. At least 3 seconds is required complete the work on a 3GHz processor.
- What if the operation on each element required more work?

Parallelization Example

```
__on_gpu__ void gpu_cube_array() {
  for(all a_i in array in parallel) {
    tmp = a_i;
    a_i = tmp * tmp * tmp;
  }
}

#define LENGTH 1000000000 // 1,000,000,000
#define CHUNKS 8 // Eight chunks to send to GPU.
int main() {
  int ary[LENGTH] = {2, ..., 2}; // assume that array is initialized with 2s.

  for(int i = 0; i < CHUNKS; i++) {
    int start_index = i * LENGTH / CHUNKS;
    int stop_index = start_index + LENGTH / CHUNKS;
    copy_to_gpu(ary[start_index ... stop_index]) // About 120,000,000 cycles.
    gpu_cube_array(); // About 25,000,000 cycles.
    copy_from_gpu(ary[start_index ... stop_index]) // about 120,000,000 cycles.
  } // 25,000,000 + 2 * 120,000,000 = 265,000,000 cycles

  // About 2,120,000,000 cycles total. ~1.5 seconds on a 1.5GHz processor.
}
```


Parallel Example

Advantages

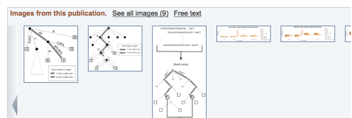
- If the GPGPU runs at 1.5GHz then it will take at least 1.41 seconds.
- Takes less time than the serial example.

Disadvantages

- More difficult to read.
- More overhead is required to copy the data.
- Make sure that the GPU memory is not exhausted.

Use Case

- GPGPUs are already being used in the scientific community.


Parallel Comput. 2009 Aug 1;35(8):429-440.
Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.
Trapnell C, Schatz MC.
Author information
Abstract
MUMmerGPU uses highly-parallel commodity graphics processing units (GPU) to accelerate the data-intensive generation DNA sequence data to a reference sequence for use in diverse applications such as disease genoty MUMmerGPU 2.0 features a new stackless depth-first-search print kernel and is 13× faster than the serial CPU nearly 4× faster in total computation time than MUMmerGPU 1.0. We exhaustively examined 128 GPU data lay footprint and running time and conclude higher occupancy has greater impact than reduced latency. MUMmerGI http://mummergpu.sourceforge.net.
PMID: 20161021 [PubMed] PMCID: PMC2749273 Free PMC Article
Images from this publication. See all images (9) Free text

GPGPU-accelerated Interesting In other Computations on GeoSpatial of Results

Sushil K. Prasad, Georgia State University, Atlanta, GA, sprasad@gsu.edu
Shashi Shekhar, University of Minnesota, Minneapolis, MN, shekhar@cs.umn.edu
Xun Zhou, University of Minnesota, Minneapolis, MN, xun@cs.umn.edu
Michael Evans, University of Minnesota, Minneapolis, MN, mevans@cs.umn.edu

ABSTRACT
It is imperative that for scalable solutions of GIS computations the modern hybrid architecture comprising a CPU-GPU pair is exploited fully. The existing parallel algorithms and data structures port reasonably well to multi-core CPUs, but poorly to GPGPUs because of latter's atypical fine-grained, single-instruction multiple-thread (SIMT) architecture, extreme memory hierarchy and coalesced access requirements, and delicate CPU-GPU coordination. Recently, our parallelization of the state-of-art interesting sequence discovery algorithms calculates one-dimensional interesting intervals over an image representing the normalized difference vegetation indices of Africa within 31 ms on an nVidia 480GTX. To our knowledge, this paper reports the first parallelization of these algorithms. This allowed us to process 612 images representing biweekly data from July 1981 through Dec 2006 within 22 seconds. We were also able to pipe the output to a display in almost real-time, which would interest climate scientists. We have also undertaken parallelization of two key tree-based data structures, namely R-tree and heap, and have employed parallel R-tree in polygon overlay system. These data structure parallelization are hard because of the underlying tree topology and the fine-grained computation leading to frequent access to such data structures severely stifling parallel efficiency.

History

ENIAC

- Weight 30 tons.
- Consumed 200kW.
- Required 19,000 vacuum tubes to operate.
- Slow (~100 kHz)

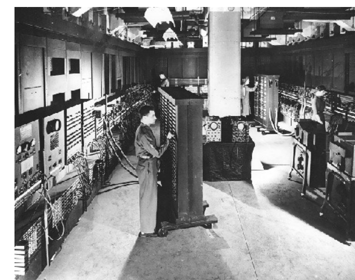


Figure 1: Unidentified Photographer [photo]. Retrieved from http://en.wikipedia.org/wiki/ENIAC

Colossus Mark II

- Used a process similar to systolic array to parallelize the cryptanalysis of the enigma.
- Systolic arrays weren't described until 1978 by H. T. Kung and Charles E. Leiserson.
  - A grid of processors designed to process instructions and data in parallel.
- Modern graphics processing units are similar in design to a systolic array.

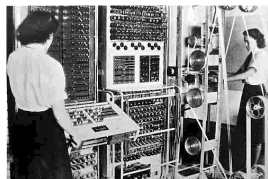


Figure 2: Retrieved from http://en.wikipedia.org/wiki/Colossus_Mark_II

Gaming Consoles and Workstations

- Beginning in the 70's gaming consoles started to pioneer the use of graphics processing units (GPU) to speed up computations so that an image could be rendered quickly on a display.
- It became common to have a dedicated processor to handle system graphics and offload work from the CPU.
- Later workstations started to come with dedicated graphics processors.
- GPUs were still dumb and nonprogrammable. Therefore using the GPU to perform computations required an understanding of the hardware and strange hacks.

Dedicated Graphics

- 1990's
  - Market fills with dedicated graphics chips makers.
  - S3 Graphics
  - 3dfx
  - Nvidia
  - ATI (Array Technology Inc.)
  - Silicone Graphics
- Post 2000's
  - Dedicated processing units are a standard in laptops and Desktops.
  - Programmable

Graphics Processing Unit vs CPU

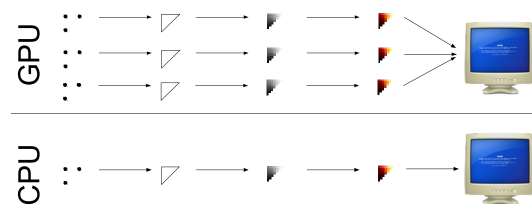


Figure 3: Simple example of the GPU vs CPU when rendering an image.

Fixed Graphics Pipeline (1980's - 1990's)

- Configurable but not programmable.
- Fixed Pipeline
  1. Send image as a set of vertexes and colors to the GPU when then performs steps 2 - 6.
  2. In parallel, determine if each point is visible or not. (*Vertex Stage*)
  3. In parallel, assemble the points into triangles. (*Primitive Stage*)
  4. In parallel, convert the triangles into pixels. (*Raster Stage*)
  5. In parallel, color each pixel based on the color of its neighbors. (*Shader Stage*)
  6. Display the grid of pixels on the screen.

Programmable Pipeline (2000+)

- Why not let the programmer customize steps in the pipeline?
- As a result the programmable pipeline was born.
  - Allow customization of the vertex processing step.
  - Allow customization of the shading step.
  - GLSL (OpenGL shading language)
- The game designer could now modify how the GPU processed the data in parallel.
- More components of the pipeline were converted to keep up with user demands.
- Still limited to graphics. Performing other computations required the user to restructure the problem in a form the GPU could understand (as an image).

Unified Pipeline (Nvidia)

- Programmable pipeline feature-creep.
- Instead of adding more hardware to perform different tasks. Generalize the pipeline such that it can perform each step in the fixed pipeline.
- Use an array of unified processors and perform three passes on the data before sending it to the framebuffer.
  - vertex processing: Take the vertex data, transforms it
  - geometry processing
  - pixel processing
- GPU can now perform load balancing at each step.
- Precursor to the general purpose graphics processing unit (GPGPU).

Unified Pipeline (Nvidia)

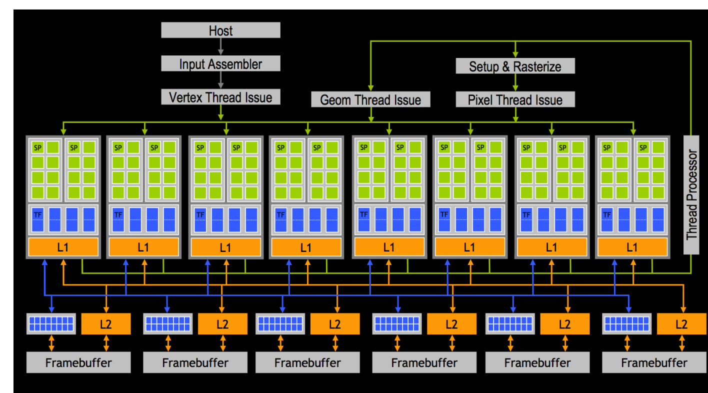


Figure 4: Nvidia's Unified Pipeline: Adopted from *Programming Massively Parallel Processors*.

General Purpose Graphics Processing Unit

- With the birth of the unified pipeline and more advanced shading algorithms. The GPU began to resemble a CPU. Hence the term GPGPU (General Purpose Graphics Processing Unit) was born.
- This caught the interest of researchers who used the shading language to solve computational problems. [1]

GPU implementation of neural networks

Kyoung-Su Oh, Keechul Jung

Show more

DOI: 10.1016/j.patcog.2004.01.013 Get rights and content

Abstract

Graphics processing unit (GPU) is used for a faster artificial neural network. It is used to implement the matrix multiplication of a neural network to enhance the time performance of a text detection system. Preliminary results produced a 20-fold performance enhancement using an ATI RADEON 9700 PRO board. The parallelism of a GPU is fully utilized by accumulating a lot of input feature vectors and weight vectors, then converting the *many* inner-product operations into *one* matrix operation. Further research areas include benchmarking the performance with various hardware and GPU-aware learning algorithms.

[1] Kyoung-Su Oh, Keechul Jung, GPU implementation of neural networks, Pattern Recognition, Volume 37, Issue 6, June 2004, Pages 1311-1314, ISSN 0031-3203, http://dx.doi.org/10.1016/j.patcog.2004.01.013.

New APIs

Figure 5: GPGPU APIs

- In response to the research community Nvidia and Khronos created APIs that allow users to repurpose their GPUs by writing their own programs.
- Provide interface to program GPGPU.
  - CUDA (Compute Unified Device Architecture) a proprietary API to Nvidia GPUs.
  - OpenCL (Open Computing Language) an open standard by the Khronos group.

Speed Evolution

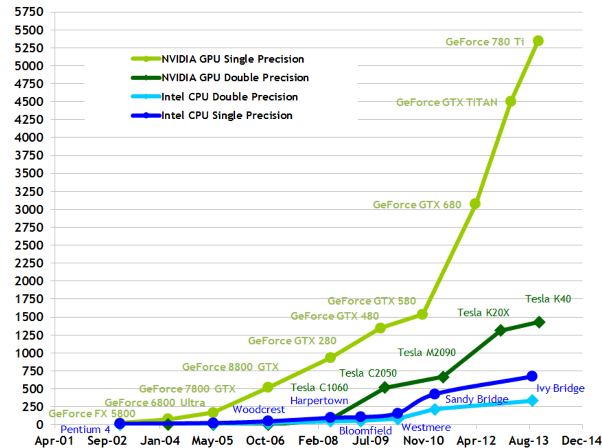


Figure 6: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

Parallelism

Parallel Processing

- Bit-level parallelism
  - increase word size to reduce the number of passes a processor needs to perform in order to perform arithmetic.
  - e.g. 8-bit processor must perform two passes when adding two 16-bit numbers.
- Systolic Array
  - Data centric processing. It is a data-stream-driven by data counters unlike the Von-Neumann architecture is an instruction-steam-driven program counter.
- Flynn's Taxonomy
  - General categorization of parallel processing paradigms.

Flynn's Taxonomy

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Flynn's Taxonomy

Single Instruction Single Data

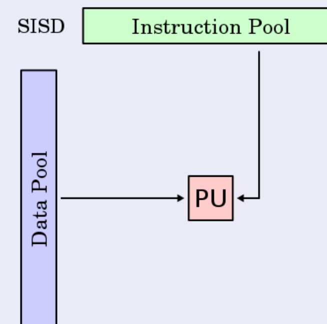


Figure 7: Wikipedia: SISD. PU stands for processing unit.

Flynn's Taxonomy

Single Instruction Multiple Data

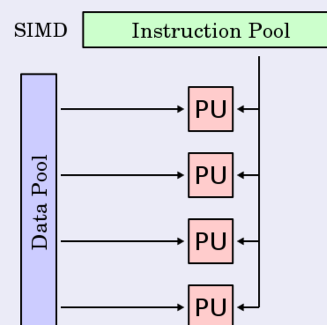


Figure 8: Wikipedia: SIMD. Keep this diagram in mind. GPU parallelism falls into this category

Flynn's Taxonomy

Multiple Instruction Single Data

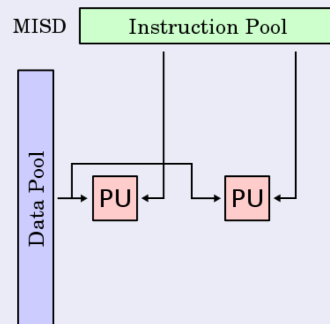


Figure 9: Wikipedia: MISD

Flynn's Taxonomy

Multiple Instruction Multiple Data

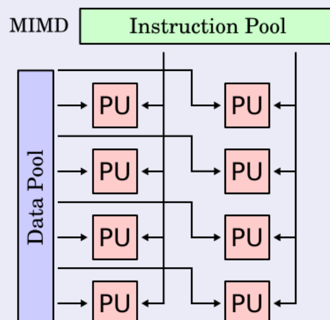


Figure 10: Wikipedia: MIMD

Embarrassingly Parallel Problem

- A problem that can be subdivided into smaller problems that are independent of each other.
  - Matrix multiplication. Each cell in the new matrix is independent of the other cells.
  - Processing vertexes and pixels on the GPU is embarrassingly parallel.
  - Genetic Algorithms
  - Bioinformatics: BLAST (Basic Local Alignment Tool) searches through a genome.

Architecture Comparison: CPU vs GPGPU

CPU Parallelism Structure

- Multiple CPUs per motherboard.
- Multiple cores per CPU.
- 2 or more CPUs per motherboard.
- Clustered hardware or distributed hardware.
- Memory is shared amongst computer peripherals.

CPU Parallelism Structure

Advantages

- Low latency
- Interrupts
- Asynchronous events
- Branch prediction
- Out-of-order execution
- Speculative execution
- Scalable with minimal hardware

CPU Parallelism Structure

Disadvantages

- Low throughput
- CPU's have evolved to do generalized work.
- They are not optimized to do a single task extreamly well.
- Limited number of threads (more complex but limited)
- May not be the best solution for math intensive scientific computations.

GPGPU Parallelism Structure

Structure

- Single Die with hundreds of massively parallel processors.
- Each processor is simpler than a standard CPU core.
  - handle large amounts of concurrent threads.
- Dedicated memory for the GPU.

GPGPU Parallelism Structure

Advantages

- High throughput
  - threads hide memory latencies.
- Efficient at solving math intensive problems.
- Handles larger amounts of data far quicker than the CPU.
- No need to deal with system interrupts.
- Free the CPU of unnecessary work.

GPGPU Parallelism Structure

Disadvantages

- High latency
- Needs to process large amounts of data. (not good for small tasks)
- Processes running in the GPU can only access resources available on the GPU card.
- Can only handle math intensive tasks. Cannot take advantage, of network communication, system calls,...
- Requires base hardware for the GPU (motherboard, CPU, storage) cannot run alone.

GPGPU Architecture

Nvidia GPGPU Architecture

- Streaming Multiprocessors (SM)
  - CUDA (Compute Unified Device Architecture) cores. Also called streaming processors (SP)
  - CUDA cores are an Nvidia branded ALU (Arithmetic Logic Unit).
  - Control units
  - Registers (local memory)
  - Execution pipelines
  - Caches (shared memory)

Nvidia GPGPU Architecture

Fermi Architecture

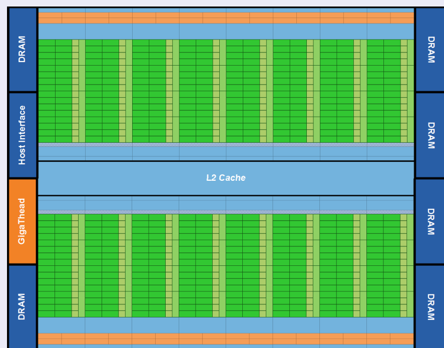


Figure 11: Retrieved from Nvidia's Fermi Whitepaper

Nvidia GPGPU Architecture

Fermi Streaming Multiprocessor

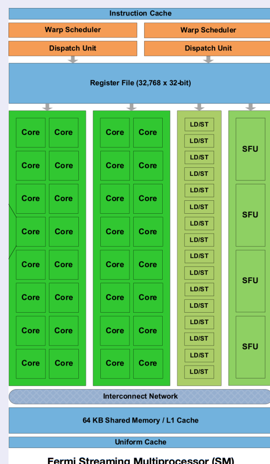


Figure 12: Retrieved from Nvidia's Fermi Whitepaper

Nvidia GPGPU Architecture

CUDA Core (ALU)

- Floating point (FP) unit
  - Single Precision
  - Double Precision (far slower than single precision)
- Integer (INT) unit
  - Boolean, Move, Compare

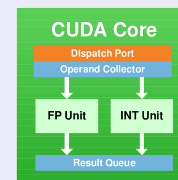


Figure 13: Retrieved from Nvidia's Fermi Whitepaper

Programming with CUDA

Overview

- Minimal code example.
- Step through the code and explain CUDA concepts.
- Provide minimal kernel example.
- Step through kernel code and explain CUDA concepts.

Minimal Working Example

- Minimal program to transfer data to the GPU, do some work, and transfer it back.

```
#include<vector>
#include<cuda_runtime.h>
#define LENGTH 1024
__host__ int main() {
  std::vector<int> host_array(LENGTH, 2);

  int * dev_array = NULL; // Points to the memory located on the device.
  cudaMalloc((void **)&dev_array, LENGTH * sizeof(int));
  cudaMemcpy(dev_array, &host_array[0], LENGTH * sizeof(int),
             cudaMemcpyHostToDevice);

  dim3 blockDims(LENGTH, 1, 1); // dim3 is struct supplied by cuda_runtime.h
  dim3 gridDims(1, 1, 1);
  kernel_cube_array<<<gridDims, blockDims>>>(dev_array, LENGTH);
  cudaDeviceSynchronize();

  cudaMemcpy(&host_array[0], dev_array, LENGTH * sizeof(int),
             cudaMemcpyDeviceToHost);
  cudaFree(dev_array);
}
```

Transfer Data to the Device

- Most problems involve the manipulation of a dataset.
- Before the GPU can perform any work the data must be transfered over to its memory.

```
std::vector<int> host_array(LENGTH, 2);

int * dev_array = NULL; // Points to the memory located on the device.
cudaMalloc((void **)&dev_array, LENGTH * sizeof(int));
cudaMemcpy(dev_array, &host_array[0], LENGTH * sizeof(int),
          cudaMemcpyHostToDevice);
```

- `cudaMalloc` initializes a contiguous set addresses in the GPU's *global* memory.
- `cudaMemcpy` copies the data from the `host_array` which resides in the computer's main memory to the `dev_array` on the GPU's global memory.

Thread Hierarchy

- The next three lines involve setting up the thread partitioning scheme.

```
dim3 blockDims(LENGTH, 1, 1);
dim3 gridDims(1, 1, 1);
kernel_cube_array<<<gridDims, blockDims>>>(dev_array, LENGTH);
```

- Threads can be considered as the *currency* of the GPU. They are the smallest execution unit that is defined by the kernel function `kernel_cube_array`
- `dim3 blockDims(LENGTH, 1, 1)` defines the block dimensions. Every thread *belongs* to a block.
- `dim3 gridDims(1, 1, 1)` defines the dimension of the grid. Every block *belongs* to a grid.
- `kernel_cube_array<<<gridDims, blockDims>>>(dev_array, LENGTH);` tells the GPU to execute a single block of size `LENGTH`.

Thread Hierarchy

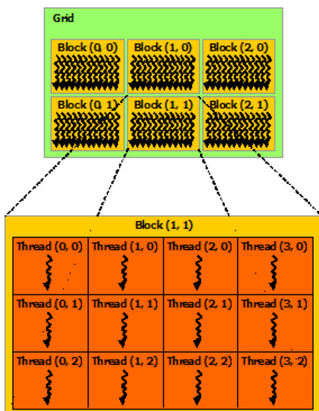


Figure 14: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

Blocks

- Blocks of threads can be partitioned amongst the streaming multiprocessors since they are independent.

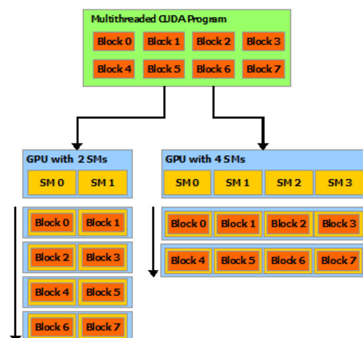


Figure 15: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

Transferring Data to the Host

- Now that the work has been completed we must transfer the data back to the host.

```
cudaDeviceSynchronize();
cudaMemcpy( &host_array[0], dev_array, LENGTH * sizeof(int),
            cudaMemcpyDeviceToHost);
cudaFree(dev_array);
```

- Wait for all of the threads to complete their work `cudaDeviceSynchronize`.
- Then transfer the data back to the host device with `cudaMemcpy`.
- All memory that was allocated must be freed with `cudaFree`.

Kernel Implementation

- Implementation of `kernel_cube_array`.

```
__device__ int cube(int a) {
  return a * a * a;
}

__global__ void kernel_cube_array(int * dev_array, int length) {
  int tidx = threadIdx.x + blockIdx.x * blockDim.x;

  __shared__ int shared_mem[LENGTH];
  shared_mem[tidx] = dev_array[tidx];
  __syncthreads();

  int b = shared_mem[tidx];
  shared_mem[tidx] = cube(b);
  __syncthreads();

  dev_array[tidx] = shared_mem[tidx];
}
```


Kernel Functions

- Kernels are the definitions of thread instances.
- Special functions that are designed to run on the GPU.
- Written using an ANSI C syntax with additional extensions.
  - `__device__` kernels are only callable on the GPU through other kernels.
  - `__global__` kernels are only callable from the host but not from the device.
  - `__host__` kernels are simply classic C functions. If a function doesn't have any declarations it defaults to this one.
  - <<<>>> to define how a kernel executes (as seen in the earlier slides).

Thread Indexes

- The first line in the kernel builds an index for the thread to use.

```
int tidx = threadIdx.x + blockIdx.x * blockDim.x
```

- CUDA provides a set of ANSI C extensions that allow the programmer to access thread, block, and grid properties.
  - `threadIdx` is the index of the *thread relative to the block*.
  - `blockIdx` is the index of the *block relative to the grid*.
  - `blockDim` is the dimensions of the block.
  - `gridDim` is the dimensions of the grid.
- You can combine use these extensions to determine the ID the GPU uses for the thread.
  - `tid = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y`
- Custom indexing schemes can also be implemented.

Thread Indexes

- How can thread indexes be used?
  - Custom indexes can be used to access memory in different ways.
  - *map*: Each thread index corresponds to a single cell in memory. A *one-to-one* mapping.
  - *gather*: Each thread index corresponds to multiple locations in memory. A *many-to-one* mapping.
  - *scatter*: Each thread index can be used to write to multiple locations. A *one-to-many* mapping.
  - *stencil*: Similar to a *gather* a stencil can be used calculate a new value from many values. Stencils are primarily used for image manipulation and simulation operations.
  - Control execution paths of individual threads in a block.
  - Other forms of thread communication.

Thread Indexing

Memory Access Patterns

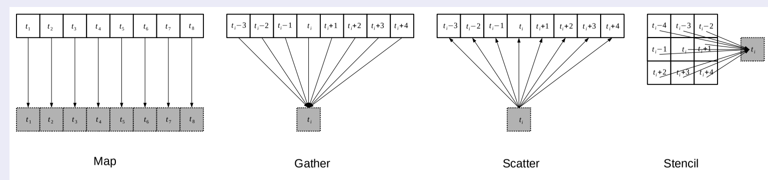


Figure 16: White boxes represent memory addresses. Grey boxes represent cells. The arrows represent the direction of communication (read / write).

Shared Memory.

- The next couple of lines load the data from global memory into the shared memory (L2 cache).

```
__shared__ int shared_mem[LENGTH];
shared_mem[tidx] = dev_array[tidx];
```

- The `__shared__` keyword defines a variable that is visible to all threads in a block.
- The next line tells each thread to copy its corresponding data from *global* memory to *shared* memory.

Memory Hierarchy

- Global
  - Visible amongst all blocks.
  - Main memory of the device.
  - Large capacity 1GB to 6GB (continuing to grow).
  - Slow and high latency.
- Shared
  - Shared amongst threads in a block.
  - L2 cache that is shared amongst the SMs
  - Small capacity (several megabytes)
- Local
  - Visible only to executing thread.
  - Stored on the registers partitioned to the active thread.
  - Stored in *global memory*.

Memory Hierarchy

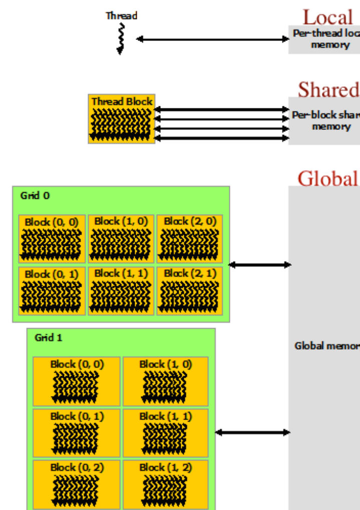


Figure 17: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit

Memory Hierarchy

- `int b = shared_mem[tidx];`.
- b is what you call a local variable. Local variables are only visible to individual threads.
- Terminology is confusing. Local memory is **not** guaranteed to be fast.
- The compiler determines where to store local variables. According to Nvidia's documentation the following rules will determine if a local variable is stored in global memory.
  - Arrays for which the compiler cannot determine that they are indexed with constant quantities.
  - Large structures or arrays that would consume too much register space.
  - Any variable if the kernel uses more registers than available (this is also known as *register spilling*).

Memory Hierarchy

Scopes Overview

Memory	Scope	Lifetime
Register	Thread	Kernel
Local	Thread	Kernel
Shared	Block	Kernel
Global	Grid	Application
Constant	Grid	Application

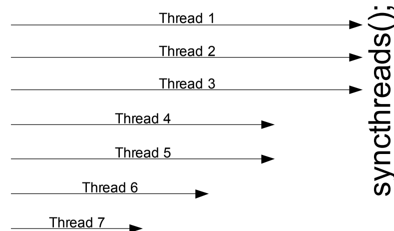
Part of Table 5.1 in *Programming Massively Parallel Processors*.

Device Functions

- Besides writing kernels it is also possible to write device functions that are callable from a kernel.
- `shared_mem[tidx] = cube(b);` makes a call to the device function `__device__ cube(int a)`.
- Device functions are only callable from the GPU. You cannot have a `__host__` function such as `main` call a device functions.
- Useful if there is a need to modularize GPU code.

Thread Synchronization

- The kernel function `kernel_cube_array` contains two calls to the function `__syncthreads()`
- `__syncthreads()` causes all thread instances in a **block** to wait for other threads in the same block to catch up before continuing.



-
- `__threadfence()` is similar in functionality except that it signals all threads in all blocks to catch up.

Kernel Function

- Recall that we are working with shared memory.
- Need to transfer results back to global memory.
- All that is needed is to map the results from the shared array to the global array.
- `dev_array[tidx] = shared_mem[tidx];`

Overview of Learned Objectives

- Learned to initialize memory.
- Transfer data to and from the GPU.
- Divide the problem into blocks of threads and a grid of blocks.
- Call a kernel function to invoke the thread instances.
- Implement kernel and device functions.
- Introduced to thread hierarchies.
- Introduced to memory hierarchies.
- Synchronizing thread actions.

Extra Information on Threads

- When blocks are being executed by an SM. Not all threads are executing concurrently.
- SMs execute warps of threads at a time. A warp usually consists of 32 threads.
- GPU that has a compute capability of 3.0 supports at maximum:
  - 64 concurrent warps per SM.
  - Each warp consists of 32 threads executing in parallel.
- The Quadro K2000 has 2 SMs. Therefore it can execute $2 * 64 * 32 = 4096$ threads concurrently.
- The Tesla S2050 has 14 SMs. Therefere it can execute $14 * 64 * 48 = 28762$ threads concurrently!

Compilation

- General compilation procedure when calling `nvcc` on source file (ends in the `.cu` extension) with existing kernel functions.

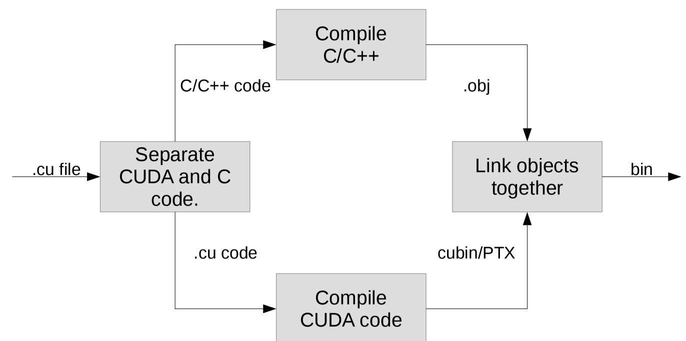


Figure 18: Simplified version of Nvidia's compilation procedure.

Matrix Multiplication

Matrix Multiplication Algorithm

- Let A, B, and C be three matrices. Such that A is $m \times n$, B is $n \times o$, and C is the product of A and B (it's dimensions are $m \times o$).
- The classic algorithm states that every cell in C is the dot product of the corresponding row and column in A and B respectively.
- $c_{ij} = \mathbf{a_i} \bullet \mathbf{b_j}$ where $\mathbf{a_i}$ is row i A and $\mathbf{b_j}$ is column j in B.
  - $1 <= i <= m$ and $1 <= j <= o$
- Iterate through all values of i and j to calculate the values of C.
- Implementation consists of a double loop through all cells in C.

Matrix Multiplication Algorithm

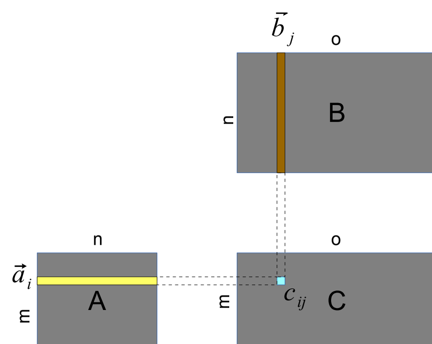


Figure 19: Reproduced from Nvidia's CUDA Toolkit Documentation

Matrix Data Structure.

- 1D or 2D array of doubles?
  - A 2D structure requires more work to transfer to the GPU. `cudaMalloc` and `cudaMemcpy` required for each row in the matrix.
  - 1D structure requires single `cudaMalloc` and `cudaMemcpy`.
- Access row 4 and col 2 in a 6×6 row-major matrix.
  - If a is 2D then `a[4][2]`
  - If a is 1D then `a[4 * 6 + 2]`
- Access row i and col j in a $m \times n$ row-major matrix.
  - `a[i * n + j]`

Project 1: Matrix Multiplication

- Implement the classic matrix multiplication algorithm twice.
  - Once for the CPU.
  - Once for the GPU by writing your own kernel function.
- Implement necessary code to manage memory and transfer data between the GPU and CPU.

Advanced Thread Management

Thread Management Objectives

- Atomic Operations
  - add, divide, and multiply.
- Thread Syncronization
  - `__syncthreads()`
  - `__threadfence()`

Race Conditions

- When two or more threads are accessing and modifying a value in memory (global or shared) yet the order in which they do it is important. If that order isn't followed and it causes undefined behavior then we have a **race condition**
- E.X. A group of CUDA threads are binning the values in a large dataset. Incrementing counts in the bins can result in a race condition when the value changes before it is stored back in memory. Thereby creating inconsistencies.

Race Conditions

```
/// rseq and bins are pointers to an array of ints in global memory.
__global__ void bin_kernel_simple(int * rseq, int * bins,
                                  int bin_width, int rseq_len)
{
    const int index = threadIdx.x + blockIdx.x * blockDim.x;
    if ( index < rseq_len) bins[rseq[index] / bin_width]++;
}
```

- Simple program that bins up values from a random sequence of integers (`rseq`).
- `rseq` resides in global memory. When `bins[rseq[index] / bin_width]++` executed then there will be a race condition.
  - Why? Each SM is executing at most w threads concurrently. Therefore, it is possible to have $w * s$ (s is the count of SMs on the device) threads overwriting a single value in `bins`.

Atomic Operations

- A simple solution to a race condition is to make use of an atomic operation which are supported by CUDA. Here are a few that may be usefull.
  - `atomicAdd`, `atomicSub`, `atomicMin`, and `atomicMax`.
  - Additional operators can be found in `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions`
- Updating the `if` statement will fix the race condition. `atomicAdd` requires that we pass the address of the bin in `bins`.

```
if (index < rseq_len) atomicAdd( &bins[rseq[index] / bin_width], 1);
```


Thread Syncronization

- What if an algorithm consists of multiple substeps and the order in which the substeps are executed is important?
- Stop execution of threads until all threads reach a desired state.
  - `__syncthreads()` threads in a thread block will wait for other threads in the same block to catch up.
  - `__threadfence()` threads in all thread blocks will wait for other threads to catch up.

Advanced Memory Management

Shared Memory

- Global memory is slow. If possible use shared memory to decrease the count of global memory accesses.
- Shared memory is visible amongst all threads in a block, and the lifetime is to that of the block.
- Simple use cases:
  - Shared memory to prevent global memory locks due to using the `atomicAdd` operation.
  - Prefetch data before performing memory intensive calculations.

Shared Memory Allocation

- There are two ways of using shared memory to reduce the count of global memory locks.

Static Allocation

- Define the variable or array at the beginning of the kernel using the `__shared__` keyword.
- If sharing an array of data the size must be known at compile time.
- Example:

```
#define DEFAULT_SHARED_SIZE 1024
__global__ void kernel_shared() {
    __shared__ int array_of_ints[DEFAULT_SHARED_SIZE];
}
```


Shared Memory Allocation

Dynamic Allocation

- Only a single variable can be dynamically allocated.
- The size of the shared memory must be known at the kernel invocation.
- `extern` keyword required so that `nvcc` recognizes it.
- Example:

```
#define DEFAULT_SHARED_SIZE 1024
__global__ void kernel_shared() {
    extern __shared__ int dynamic_array_of_ints[];
}

void kernel_invoking_function() {
  ...
  size_t shared_size = 1024 * sizeof(int);

  kernel_shared<<<1024,1024,shared_size>>>();
  ...
}
```

Shared Memory Question

- Will this kernel allocate the shared memory in global memory or L2 memory?

```
#define DEFAULT_SHARED_SIZE 1024
__global__ void kernel_shared()
{
    __shared__ int * shared_bins;

    if(threadIdx.x == 0) { // first allocate the shared memory.
        size_t shrd_sz = DEFAULT_SHARED_SIZE * sizeof(int);
        shared_bins = (int *)malloc(shrd_sz);
        memset(shared_bins, 0, shrd_sz);
    }
    __syncthreads(); // Wait for the memory allocation.

    ... // other computations
}
```


Parallel Algorithms

Complexity of a Parallel Algorithm

- Step Complexity
  - The number of iterations a parallel device needs to do before completing an algorithm.
- Work Complexity
  - The amount of work a parallel device does when running a algorithm.
- Work Efficient
  - A parallel algorithm that is asymptotically the same as its serial implementation.

Reduce

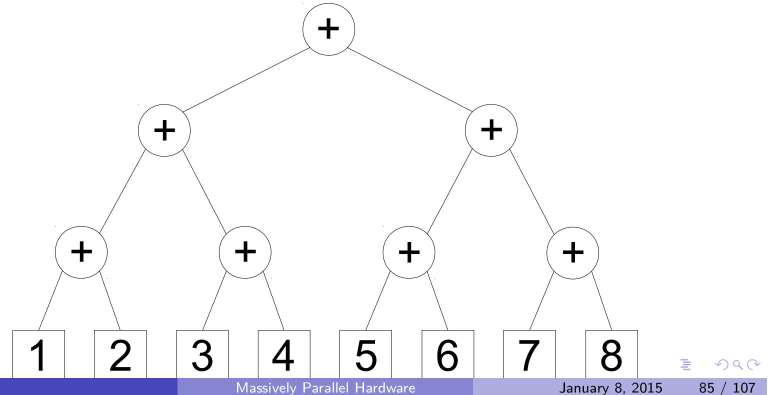
- Matrix multiplication project was a simple mapping since each operation was independent of other operations.
- A reduction is dependent of other components but can still be parallelized.
  - A reduction can be applied if the operator $\oplus$ satisfies the following conditions.

1. Binary
2. Associative
3. Identity element

- Take a sequence of values and apply a binary operator over each pair of elements to get an overall sum.

Reduction as a Tree

- Given the properties of $\oplus$ the reduction can be represented as a tree where each branch is calculated independently.
- For example lets sum reduce $[1, 2, 3, 4, 5, 6, 7, 8]$ then we would have the resulting tree.



Reduction as a Tree Complexity

- The work complexity of the tree reduction is $O(n)$ since the device will still need to perform $n - 1$ operations to reduce a list of numbers.
- The step complexity of the tree reduction is $O(\log(n))$ since each level on the tree is independent and the height of a binary tree is $\log(n)$ where n is the number of leaves.

Reduce Serial

- Serial implementation of a reduction.

```
int sum_reduce(std::vector< int > & sequence) {
  int sum = 0;
  for(int i = 0; i < sequence.size(); i++) {
    sum += sequence[i];
  }

  return sum;
}
```


Reduce Kernel

- Simple example a reduction implemented in CUDA.

```
__shared__float partialSum[];

unsigned int t = threadIdx.x;
for( unsigned int stride = 1; stride < blockDim.x, strid *= 2)
  __syncthreads();
  if(t % (2 * stride) == 0) {
    partialSum[t] += partialSum[t + stride]
  }
}
```

Scan

- Scan applies concepts similar to reduce.
- Take a sequence of values, apply a binary operator while keeping a cumulative output.
- Example:
  - INPUT: $S = [a_0, a_1, a_2, a_3, ...a_n], \quad i = I, \quad \oplus$ where S is the sequence and i is the identity for the operator $\oplus$.
  - OUTPUT:
  - Exclusive: $[I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), (I \oplus a_0 \oplus a_1 \oplus a_2), ...(I \oplus ... \oplus a_{n-1})]$

Scan Serial

- Serial implementation of scan.

```
// 'in' and 'out' have the same length.
void sum_scan(std::vector< int > & in, std::vector< int > & out) {
  int sum = 0;
  for( int i = 0; i < int.size(); i++ ) {
    out[i] = sum;
    sum += in[i];
  }
}
```


Scan Algorithms

Hillis-Steele

- Simple to implement, but lacks efficiency when scanning large arrays.

Blelloch

- More work efficient than the Willis-Steel scan. Has the same efficiency as the serial scan.

Others

- Kogge–Stone
- Brent Kung

Hillis-Steele

- Takes a simple approach to parallelizing the scan. Treats the summation operations as a binary tree with no attempt to minimize the number of operations.

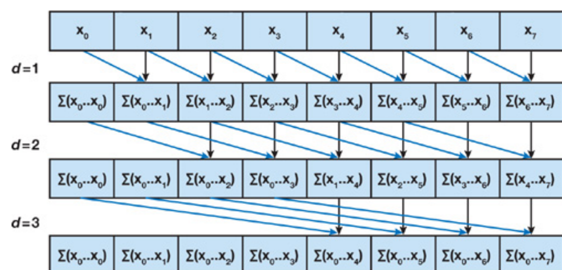


Figure 21: Hillis-Steele Scan: Adopted from *GPU Gems 3*

Hillis-Steel Complexity

- What is the complexity of the Hillis-Steele algorithm?
  - Work complexity?
  - Step complexity?
- The work efficiency is $O(n \log(n))$.
- The step efficiency is $O(\log(n))$.

Hillis-Steele Implementation

- Hillis-steele scan using a single block of threads.

```
#define SEQ_SIZE 1024
#define BLOCK_SIZE 1024
__global__ void hs_scan_kernel(int * in, int * out) {
  int tid = threadIdx.x;

  __shared__ int shared[BLOCK_SIZE];

  shared[tid + 1] = in[tid];
  __syncthreads();

  for( int d = 1; d <= log2(SEQ_SIZE); ++d ) {
    if( tid >= (1 << (d - 1))) {
      shared[tid] += shared[tid - (1 << (d - 1))];
    }
    __syncthreads();
  }

  out[tid] = shared[tid];
}
```


Blelloch

- Splits the scan operation into two phases, a down-sweep and up-sweep phase.

Down-Sweep

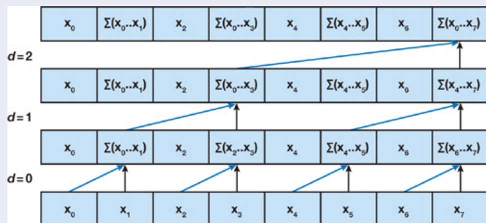


Figure 22: Blelloch Down-Sweep: Adopted from *GPU Gems 3*

Blelloch

Down-Sweep Complexity

- What is the complexity of the down-sweep portion?
  - Work complexity?
  - Step complexity?

Blelloch

Down-sweep Implementation

```
__device__ void bl_sweep_down(int * to_sweep, int size) {
  to_sweep[size - 1] = 0;
  int t = threadIdx.x;
  for( int d = log2(size) - 1; d >= 0; --d) {
    __syncthreads();
    if( t < size && !((t + 1) % (1 << (d + 1)))) {
      int tp = t - (1 << d);
      int tmp = to_sweep[t];
      to_sweep[t] += to_sweep[tp];
      to_sweep[tp] = tmp;
    }
    __syncthreads();
  }
}
```


Blelloch

Up-Sweep

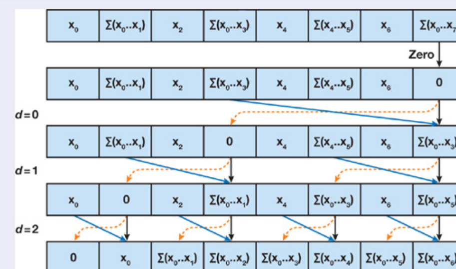


Figure 23: Blelloch Up-Sweep: Adopted from *GPU Gems 3*

Blelloch

Up-Sweep Complexity

- What is the complexity of the down-sweep portion?
  - Work complexity?
  - Step complexity?

Blelloch

Up-sweep implementation

```
__device__ void bl_sweep_up(int * to_sweep, int size) {
  int t = threadIdx.x;
  for( int d =  0; d < log2(size); ++d) {
    __syncthreads();
    if( t < size && !((t + 1) % (1 << (d + 1)))) {
        int tp = t - (1 << d);
        to_sweep[t] = to_sweep[t] + to_sweep[tp];
    }
    __syncthreads();
  }
}
```

Blelloch

Complexity

- What is the complexity of the Blelloch algorithm?

Kernel Implementation

- What would the kernel need to contain in order to make use of the up-sweep and down-sweep functions?

Compact

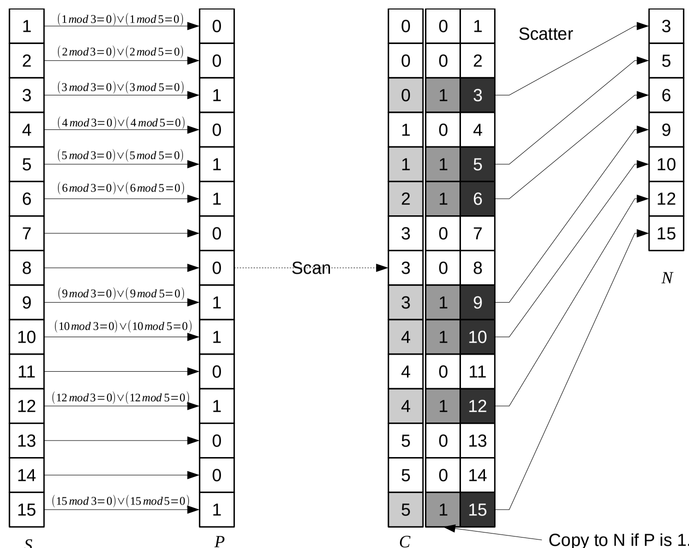
- What is compacting?
  - The process of extracting a subset from a set of items given a condition
  - Items in a set are compared against a condition. (usually an array)
  - The output of the comparison is then stored in a dataset called the predicate.
  - Using the predicate and scan operator the desired items can then be mapped into a new set.

Compact

Example

- Find elements that are either a multiple of three or five.
- S is the set to compact.
  - $S = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
- The condition to check on each element is $s \in S$ is $(s \mod 3 = 0)|(s \mod 5 = 0)$
- Resulting in the predicate set P.
  - $P = [0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1]$
- Applying the scan operator on P and output C
  - $C = [0, 0, 0, 1, 1, 2, 3, 3, 3, 4, 5, 5, 6, 6, 6]$
- Now map the items from S where each item P is 1 to the corresponding index in C into a new set N.
  - $N = [3, 5, 6, 9, 10, 12, 15]$

Compact



Projects & Homework

Homework: Scan Algorithms

- Implement the Blelloch and Hillis-Steele algorithms.
- CUDA program should only be a single file.
- Scan only a single block of 1024 threads.

Project 2: Sorting

- Implement the radix sort algorithm.
- Radix sort requires that you make use of the compact algorithm. Recall that compacting requires you:
  - Scan
  - Scatter
- Refer to `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html` for more information.

Parallel Example

Advantages

- If the GPGPU runs at 1.5GHz then it will take at least 1.41 seconds.
- Takes less time than the serial example.

Disadvantages

- More difficult to read.
- More overhead is required to copy the data.
- Make sure that the GPU memory is not exhausted.

- Cubing a value is a simple operation. So there wasn't much of a performance gain.
- A complex math operation would give the GPU more of an advantage.

Introduction

An introduction leveraging the computing capabilities of the graphics processing unit to parallelize a task.

NOTE: To whom ever is reading this document. Pandoc processes divs that inherit from the 'notes' class as a note. These will not show up in your slides. To compile slides with notes run `make notes`. You will need to be running pandoc –version >= 1.12.

Colossus Mark II

- Used a process similar to systolic array to parallelize the cryptanalysis of the enigma.
- Systolic arrays weren't described until 1978 by H. T. Kung and Charles E. Leiserson.
  - A grid of processors designed to process instructions and data in parallel.
- Modern graphics processing units are similar in design to a systolic array.

Figure 2: Retrieved from http://en.wikipedia.org/wiki/Colossus_Mark_II

- Why are systolic arrays important? They demonstrate that using multiple processors to solve tasks in parallel has been around before GPUs became commonplace.
- Even though they were not described until 1978 by H. T. Kung and Charles E. Leiserson the Colossus implemented such a method in order to decrease the time to decrypt Enigma. In reality though, it was never used to decipher Enigma that was the job of Turing's machine the electromechanical Bombe. Colossus computer.

ENIAC

- Weight 30 tons.
- Consumed 200kW.
- Required 19,000 vacuum tubes to operate.
- Slow (~100 kHz)

]. Retrieved from

Parallel Comput. 2009 Aug 1;35(8):429-440.

Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.

Trapnell C[1], Schatz MC.

Author information

Abstract

MUMmerGPU uses highly-parallel commodity graphics processing units (GPU) to accelerate the data-intensive generation DNA sequence data to a reference sequence for use in diverse applications such as disease genotyp MUMmerGPU 2.0 features a new stackless depth-first-search print kernel and is 13× faster than the serial CPU nearly 4× faster in total computation time than MUMmerGPU 1.0. We exhaustively examined 128 GPU data layo footprint and running time and conclude higher occupancy has greater impact than reduced latency. MUMmerGP http://mummergpu.sourceforge.net.

PMID: 20161021 [PubMed] PMCID: PMC2749273 Free PMC Article

Images from this publication. See all images (9) Free text

tegra

GPGPU-accelerated Interesting In other Computations on GeoSpatial of Results * †

Sushil K. Prasad, Georgia State University, Atlanta, GA, sprasad@gsu.edu
Shashi Shekhar, University of Minnesota, Minneapolis, MN, shekhar@cs.umn.edu
Xun Zhou, University of Minnesota, Minneapolis, MN, xun@cs.umn.edu
Michael Evans, University of Minnesota, Minneapolis, MN, mevans@cs.umn.edu

ABSTRACT

It is imperative that for scalable solutions of GIS computations the modern hybrid architecture comprising a CPU-GPU pair is exploited fully. The existing parallel algorithms and data structures port reasonably well to multicore CPUs, but poorly to GPGPUs because of latter's atypical fine-grained, single-instruction multiple-thread (SIMT) architecture, extreme memory hierarchy and coalesced access requirements, and delicate CPU-GPU coordination. Recently, our parallelization of the state-of-art interesting sequence discovery algorithms calculates one-dimensional interesting intervals over an image representing the normalized difference vegetation indices of Africa within 31 ms on an nVidia 480GTX. To our knowledge, this paper reports the first parallelization of these algorithms. This allowed us to process 612 images representing biweekly data from July 1981 through Dec 2006 within 22 seconds. We were also able to pipe the output to a display in almost real-time, which would interest climate scientists. We have also undertaken parallelization of two key tree-based data structures, namely R-tree and heap, and have employed parallel R-tree in polygon overlay system. These data structure parallelization are hard because of the underlying tree topology and the fine-grained computation leading to frequent access to such data structures severely stifling parallel efficiency.

Dedicated Graphics

- 1990's
  - Market fills with dedicated graphics chips makers.
  - S3 Graphics
  - 3dfx
  - Nvidia
  - ATI (Array Technology Inc.)
  - Silicone Graphics
- Post 2000's
  - Dedicated processing units are a standard in laptops and Desktops.
  - Programmable

iSBX 275 Info

- Silicone Graphics (SGI) is important. They are accredited to creating OpenGL, OpenCl, and the Khronos group to monitor the evolution of the APIs.
- Khronos oversees the evolution of these open frameworks. `https://www.khronos.org/opencl/`
  - OpenGL stands for *Open Graphics Library*
  - OpenCL stands for *Open Computing Language*

Gaming Consoles and Workstations

- Beginning in the 70's gaming consoles started to pioneer the use of graphics processing units (GPU) to speed up computations so that an image could be rendered quickly on a display.
- It became common to have a dedicated processor to handle system graphics and offload work from the CPU.
- Later workstations started to come with dedicated graphics processors.
- GPUs were still dumb and nonprogrammable. Therefore using the GPU to perform computations required an understanding of the hardware and strange hacks.

- Atari is famous for splitting the graphics rendering from other computations.
- The ANTIC processor generated the text and bitmap graphics.
- The CTIA/GTIA processor takes the output from ANTIC and adds coloring to the image before sending it to the screen.
- `http://en.wikipedia.org/wiki/Atari_8-bit_family`
- Sega Dreamcast had a PowerVR2 CLX2 dedicated GPU
- Nintendo 64 had the 62.5 MHz SGI RCP made by Silicone Graphics (SGI)
- XBox had a custom Nvidia GeForce 3.

Programmable Pipeline (2000+)

- Why not let the programmer customize steps in the pipeline?
- As a result the programmable pipeline was born.
  - Allow customization of the vertex processing step.
  - Allow customization of the shading step.
  - GLSL (OpenGL shading language)
- The game designer could now modify how the GPU processed the data in parallel.
- More components of the pipeline were converted to keep up with user demands.
- Still limited to graphics. Performing other computations required the user to restructure the problem in a form the GPU could understand (as an image).

- Customization of the vertex and shader pipelines by using GLSL (OpenGL shading language) which had a syntax similar to C.
- The programmer would write the shader code which would get executed on either each vertex or pixel during the GPU render process.
- `http://en.wikipedia.org/wiki/GLSL`

Fixed Graphics Pipeline (1980's - 1990's)

- Configurable but not programmable.
- Fixed Pipeline
  1. Send image as a set of vertexes and colors to the GPU when then performs steps 2 - 6.
  2. In parallel, determine if each point is visible or not. (*Vertex Stage*)
  3. In parallel, assemble the points into triangles. (*Primitive Stage*)
  4. In parallel, convert the triangles into pixels. (*Raster Stage*)
  5. In parallel, color each pixel based on the color of its neighbors. (*Shader Stage*)
  6. Display the grid of pixels on the screen.

- Refer to `http://cg.informatik.uni-freiburg.de/course_notes/graphics_01_pipeline.pdf`. I find that they do a good job breaking down the steps.
- A simple breakdown of a fixed pipeline in the GPU.
  - Send the data to the GPU via OpenGL or DirectX
  - **Vertex Stage:** Operates on each vertex independently. Therefore this step is embarrassingly parallel.
  - **Primitive Stage:** Primitives (vertexes) are assembled into lines an triangles. The assembly of each primitive is also parallel.
  - **Raster Stage:** Convert all of the geometry data into pixel data. Since this involves a lot of transformation operations then this stage can also be parallelized for each shape.
  - **Shader Stage:** Interpolate the colors of the pixels based on the colors of the vertexes. Each pixel is independent of the other pixels.

2015-01-08

Massively Parallel Hardware
└─History
└─Unified Pipeline (Nvidia)

Unified Pipeline (Nvidia)

Figure 4: Nvidia's Unified Pipeline. Adopted from *Programming Massively Parallel Processors*

- SP stands for streaming processor. SP is also referred to as a CUDA core.
- The blocks of SPs in the images are streaming multiprocessors (SMs). NVidia's unified programmable processor array of the GeForce 8800 GTX graphics pipeline.
- First send the data in through the *Host* interface, next the data is passed to the input assembler. Once the assembler has completed its task, the GPU will invoke threads for the vertex stage and distribute them amongst the SPs (Streaming Processors). The SPs will perform the three rounds as mentioned in the *Unified Pipeline (Nvidia)* slide.

The advantage to having this setup is that the SPs can distribute the workload depending on the graphics demands. For example if more time is required in the pixel processing state (for image processing) then the GPU will be able to devote more time to that task alone.

Essentially the GPU is beginning to resemble a processor with multiple cores.

2015-01-08

Massively Parallel Hardware
└─History
└─Unified Pipeline (Nvidia)

Unified Pipeline (Nvidia)

- Programmable pipeline feature-creep.
- Instead of adding more hardware to perform different tasks. Generalize the pipeline such that it can perform each step in the fixed pipeline.
- Use an array of unified processors and perform three passes on the data before sending it to the framebuffer.
  - vertex processing: Take the vertex data, transforms it
  - geometry processing
  - pixel processing
- GPU can now perform load balancing at each step.
- Precursor to the general purpose graphics processing unit (GPGPU).

The unified processor description is relative to Nvidia's hardware design. ATI may perform a similar method but it may differ.

- Vertex Processing: Convert the vertex data and color data into a form that can be used by the GPU.
- Geometry Processing: Similar to vertex processing except each primitive is processed. By primative I mean vertexes that belong to triangles or quadrilaterals (it depends on the GPU, Nvidia deals with triangle primitives)
- Pixel Processing: This is essentially the shading step from the fixed pipeline. (perhaps also the raster step)

2015-01-08

Massively Parallel Hardware
└─History
└─Speed Evolution

Speed Evolution

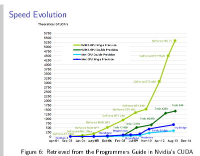


Figure 6: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

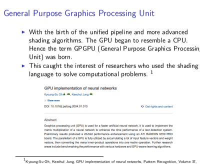
This plot was located in Nvidia's on line CUDA Toolkit Documentation (`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz36uP30F2s`). It's relative to Nvidia GPU's but it's clear that there is a trend which can be applied to GPGPUs in general (ATI GPUs, and mobile GPUs).

2015-01-08

Massively Parallel Hardware
└─History
└─General Purpose Graphics Processing Unit

General Purpose Graphics Processing Unit

- With the birth of the unified pipeline and more advanced shading algorithms. The GPU began to resemble a CPU. Hence the term GPGPU (General Purpose Graphics Processing Unit) was born.
- This caught the interest of researchers who used the shading language to solve computational problems. [1]



- Understanding the efficiency of GPU algorithms for matrix-matrix multiplication [2]

2015-01-08

Massively Parallel Hardware
└─Parallelism
└─Flynn's Taxonomy

Flynn's Taxonomy

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Flynn's taxonomy basically states that programs fall into one of the four categories. Each category may have more than one subcategory.

2015-01-08

Massively Parallel Hardware
└─Parallelism
└─Parallel Processing

Parallel Processing

- Bit-level parallelism
  - increase word size to reduce the number of passes a processor needs to perform in order to perform arithmetic.
  - e.g. 8-bit processor must perform two passes when adding two 16-bit numbers.
- Systolic Array
  - Data centric processing. It is a data-stream-driven by data counters unlike the Von-Neumann architecture is an instruction-steam-driven program counter.
- Flynn's Taxonomy
  - General categorization of parallel processing paradigms.

Recall, the first couple

2015-01-08

Massively Parallel Hardware
└─Parallelism
└─Flynn's Taxonomy

Flynn's Taxonomy
Single Instruction Multiple Data

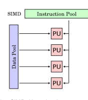


Figure 8: Wikipedia: SIMD. Keep this diagram in mind. GPU parallelism falls into this category

Single instruction multiple data is when there exist multiple processing units that can access multiple data points simultaneously. I haven't confirmed this but SQL also falls into this category since it can scale the number of PU's to access the data pool.

2015-01-08

Massively Parallel Hardware
└─Parallelism
└─Flynn's Taxonomy

Flynn's Taxonomy
Single Instruction Single Data

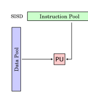


Figure 7: Wikipedia: SISD. PU stands for processing unit.

Single instruction single data is what most students in the department deal with on a daily basis. Even though we have laptops with multiple cores all of the assignments only require execution on a single core with a single set of data.

Massively Parallel Hardware
└Parallelism
└Flynn's Taxonomy

2015-01-08

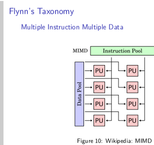


Figure 10: Wikipedia: MIMD

- Super computers apply this technique.
- `https://computing.llnl.gov/tutorials/parallel_comp/`

Massively Parallel Hardware
└Parallelism
└Flynn's Taxonomy

2015-01-08

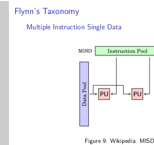
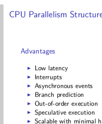


Figure 9: Wikipedia: MISD

Multiple instruction streams work on a single stream of data. A systolic array falls into this category.

Massively Parallel Hardware
└Architecture Comparison: CPU vs GPGPU
└CPU Parallelism Structure

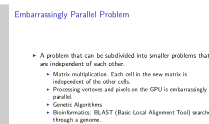
2015-01-08



- CPUs are optimized (once their pipeline is full) minimize latency at a cost of capacity.
  - Think of a CPU as a race car. It can get you somewhere in less time.
- CPUs are designed to handle system interrupts, such as from the mouse and keyboard.
- CPUs can handle multiple threads with different instructions each.
- CPUs may try to predict how the program executes in order to increase performance.
- Instructions my be reordered to increase performance.
- All that is needed at a minimum is a motherboard, ram, and a CPU.

Massively Parallel Hardware
└Parallelism
└Embarrassingly Parallel Problem

2015-01-08



- Later in the lectures we will mention the matrix multiplication algorithm.
- The GPU pipelines in the previous slides evolved to handle embarrassingly parallel problems.
- A genetic algorithm works by taking a population applying mutations and killing off the members. Since each member is independent of the other members then it is embarrassingly parallel.
- A divide and conquer approach can be taken by BLAST when searching through a genome.

GPGPU Parallelism Structure

Advantages

- High throughput
  - threads hide memory latencies.
- Efficient at solving math intensive problems.
- Handles larger amounts of data far quicker than the CPU.
- No need to deal with system interrupts.
- Free the CPU of unnecessary work.

- The GPU is able to process large datasets in parallel. Think of it as a bus that can carry 50 people at a time. If you needed to transport 200 people from Missoula to Portland it would be better to use a bus than a sports car. If needed provide a simple math breakdown of the problem.
- The GPU is optimized for algebraic functions and basic math.
- Since the GPU has a lot of threads it can hide RAM accesses.
- The GPGPU has been designed in mind for dedicated graphics and computations.

CPU Parallelism Structure

Disadvantages

- Low throughput
- CPU's have evolved to do generalized work.
- They are not optimized to do a single task extreamly well.
- Limited number of threads (more complex but limited)
- May not be the best solution for math intensive scientific computations.

- Perhaps wait time isn't an issue. Maybe the program cares more about processing data in bulk at the cost of latency.
- CPUs are setup to perform generalized work, ranging from security, to managing and communicating with all of the devices attached to the motherboard (including the GPU).
- A CPU on average can execute 2 * n threads where n is the number of cores. Any value greater than that will have diminishing returns.
- If a problem is math oriented then perhaps it is best to use a math oriented processor?

Nvidia GPGPU Architecture

- Streaming Multiprocessors (SM)
  - CUDA (Compute Unified Device Architecture) cores. Also called streaming processors (SP)
  - CUDA cores are an Nvidia branded ALU (Arithmetic Logic Unit).
  - Control units
  - Registers (local memory)
  - Execution pipelines
  - Caches (shared memory)

- The number of CUDA cores per SM depends on the *compute capability* of the GPU. Right now there are 32 cuda cores per SM. Refer to `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`.

GPGPU Parallelism Structure

Disadvantages

- High latency
- Needs to process large amounts of data. (not good for small tasks)
- Processes running in the GPU can only access resources available on the GPU card.
- Can only handle math intensive tasks. Cannot take advantage, of network communication, system calls,...
- Requires base hardware for the GPU (motherboard, CPU, storage) cannot run alone.

- GPU's have high latency because:
  1. The CPU needs to transfer the data to the GPU.
  2. The GPU has smaller cache sizes than the CPU therefore more cache misses.
  3. The GPU usually has a slower clock speed.
  4. Data needs to be transfered back to the system once the computations have completed.
- Hiding this latency requires that large datasets be used or computed.
- The GPU cannot communicate with peripherals on the machine. For example you cannot establish a network connection through the GPU to a remote machine.

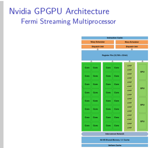


Figure 12: Retrieved from Nvidia's Fermi Whitepaper

- An exploded diagram of on of the SMs in a Fermi GPU. Students don't need to know this by heart, but it will aid them greatly if they are able to recognize the relationship between a CUDA core and an SM.
- A CUDA core is essentially a specialized ALU (Arithmetic Logic Unit) that contains both an ALU and FPU (Floating Point Unit).
- Other components that are not that important given the context of the course.
  - LD/ST stands for *Loading and Storing*. Each LD/ST unit is able to handle a single thread request per clock.
  - SFT stands for *Special Function Units*. They execute transcendental instructions such as sin, cosine, reciprocal, and square root. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

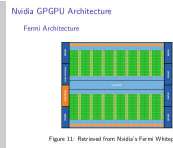
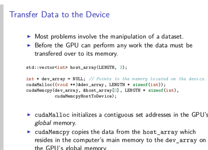


Figure 11: Retrieved from Nvidia's Fermi Whitepaper

- The Fermi white papers can be located at `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`
- The Fermi architecture is not the newest iteration, but newer architectures follow a similar model. They simply have more CUDA cores and features.
- L2 Cache is shared amongst the SMs. This is also referred to as *shared memory* when developing in CUDA.
- The DRAM on the sides in the image are referred to as *global memory*
- The *GigaThread* a proprietary Nvidia module on their GPU. It helps manage all of the thread being executed. There are probably similar components on other GPUs from different manufacturers. `http://www.nvidia.co.uk/page/8800_faq.html`



- Refer to section 3.4 (page 68) in *Programming Massively Parallel Processors* for some nice visuals of the process.

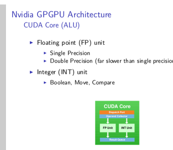


Figure 13: Retrieved from Nvidia's Fermi Whitepaper

- Double precision is slower on the GPU. The speed depends on the architecture, in some cases it can be up to 8x slower. - FPUs have been optimized to perform single precision floating point operations. - The INT unit is designed to handle all of the regular operations with integers that we are familiar with. - boolean values - bit shifting - comparing - bit reversing (change the order of the bits.

2015-01-08

Massively Parallel Hardware
└─Programming with CUDA
  └─Blocks

Blocks

- Blocks of threads can be partitioned amongst the streaming multiprocessors since they are independent.

Figure 15: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

- Blocks can be efficiently distributed amongst SMs since each block is independent.
- Since blocks are independent the GPU can maximize the throughput of the blocks.
- Also Section 4.3 (page 68) in the book *Programming Massively Parallel Processors*

2015-01-08

Massively Parallel Hardware
└─Programming with CUDA
  └─Thread Hierarchy

Thread Hierarchy

- The next three lines involve setting up the thread partitioning scheme.

```
dim3 blockDims(LENGTH, 1, 1);
dim3 gridDims(1, 1, 1);
kernel_cube_array<<<gridDims, blockDims>>>(dev_array, LENGTH);
```

- Threads can be considered as the *currency* of the GPU. They are the smallest execution unit that is defined by the kernel function `kernel_cube_array`
- `dim3 blockDims(LENGTH, 1, 1)` defines the block dimensions. Every thread *belongs* to a block.
- `dim3 gridDims(1, 1, 1)` defines the dimension of the grid. Every block *belongs* to a grid.
- `kernel_cube_array<<<gridDims, blockDims>>>(dev_array, LENGTH);` tells the GPU to execute a single block of size `LENGTH`.

- The class shouldn't worry about the kernel implementation right now.
- The trip chevrons are an extension to the ANSI C language that the `nvcc` compiler
- Chapter 4 in *Programming Massively Parallel Processors* talks about threads in an easy fausion.
- Also `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy` is a good reference which the book is based off of.

2015-01-08

Massively Parallel Hardware
└─Programming with CUDA
  └─Kernel Functions

Kernel Functions

- Kernels are the definitions of thread instances.
- Special functions that are designed to run on the GPU.
- Written using an ANSI C syntax with additional extensions.
  - `__device__` kernels are only callable on the GPU through other kernels.
  - `__global__` kernels are only callable from the host but not from the device.
  - `__host__` kernels are simply classic C functions. If a function doesn't have any declarations it defaults to this one.
  - <<<>>> to define how a kernel executes (as seen in the earlier slides).

- Refer to page 51 in the book *Programming Massively Parallel Processors* form more information about kernel functions.
- guest only kernels can be only called from other threads running on the kernel.

2015-01-08

Massively Parallel Hardware
└─Programming with CUDA
  └─Kernel Implementation

Kernel Implementation

- Implementation of `kernel_cube_array`.

```
__device__ int cube(int a) {
  return a * a * a;
}

__global__ void kernel_cube_array(int * dev_array, int length) {
  int tidx = threadIdx.x + blockIdx.x * blockDim.x;

  __shared__ int shared_mem[LENGTH];
  shared_mem[tidx] = dev_array[tidx];
  __syncthreads();

  int b = shared_mem[tidx];
  shared_mem[tidx] = cube(b);
  __syncthreads();

  dev_array[tidx] = shared_mem[tidx];
}
```

- The kernel implementation was left out on purpose for simplicity.

Thread Indexing

Memory Access Patterns

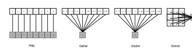


Figure 16: White boxes represent memory addresses. Grey boxes represent cells. The arrows represent the direction of communication (read / write).

- Understanding how to calculate the global thread index will aid a great deal when calculating indexes into global memory arrays.
  - For example lets say that a matrix is stored as a 1D array in memory. Calculating the thread index can be used to construct an index into that array.
- More information about thread hierarchies is available at `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`
- Refer to chapter 4 of *Programming Massively Parallel Processors* for more information.

Thread Indexes

- How can thread indexes be used?
  - Custom indexes can be used to access memory in different ways.
  - *map*: Each thread index corresponds to a single cell in memory. A *one-to-one* mapping.
  - *gather*: Each thread index corresponds to multiple locations in memory. A *many-to-one* mapping.
  - *scatter*: Each thread index can be used to write to multiple locations. A *one-to-many* mapping.
  - *stencil*: Similar to a *gather* a stencil can be used calculate a new value from many values. Stencils are primarily used for image manipulation and simulation operations.
  - Control execution paths of individual threads in a block.
  - Other forms of thread communication.

- Udacity's Parallel Programming course covers memory access patterns in section 2.
- The next slide will contain a graphical representation of each indexing scheme.

Memory Hierarchy

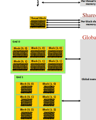


Figure 17: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

- Basic breakdown of the GPGPU memory hierarchy.

Memory Hierarchy

- Global
  - Visible amongst all blocks.
  - Main memory of the device.
  - Large capacity 1GB to 6GB (continuing to grow).
  - Slow and high latency.
- Shared
  - Shared amongst threads in a block.
  - L2 cache that is shared amongst the SMs
  - Small capacity (several megabytes)
- Local
  - Visible only to executing thread.
  - Stored on the registers partitioned to the active thread.
  - Stored in *global memory*.

- Cache sizes are dependent on the compute capability of the device. Higher compute capabilities correlates to higher cache sizes.

Compilation

- General compilation procedure when calling `nvcc` on source file (ends in the `.cu` extension) with existing kernel functions.

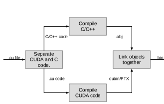


Figure 18: Simplified version of Nvidia's compilation procedure.

- `nvcc` is the Nvidia CUDA compiler. It is an extension of the standard C/C++ compiler.
- There are other extensions but we'll stick with `.cu` for consistency.
- Compilation procedure explained.
  1. The `nvcc` compiler first separates CUDA code from C/C++ code in the passed in `.cu` file.
  2. The C/C++ and CUDA code are compiled.
  - C/C++ is converted into the classic object form.
  - Depending on the `nvcc` compiler options. CUDA code is compiled into a cubin or PTX.
    - *cubin* is specific to the target GPU architecture.
    - PTX is an intermediate code that can be further compiled by the GPU driver of the target device.
  3. The objects are then linked together into an executable.

Extra Information on Threads

- When blocks are being executed by an SM. Not all threads are executing concurrently.
- SMs execute warps of threads at a time. A warp usually consists of 32 threads.
- GPU that has a compute capability of 3.0 supports at maximum:
  - 64 concurrent warps per SM.
  - Each warp consists of 32 threads executing in parallel.
- The Quadro K2000 has 2 SMs. Therefore it can execute $2 * 64 * 32 = 4096$ threads concurrently.
- The Tesla S2050 has 14 SMs. Therefere it can execute $14 * 64 * 48 = 28762$ threads concurrently!

- To figure out the GPU specs given a compute capability visit the following links:
  - `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities`
  - `https://developer.nvidia.com/cuda-gpus`
- The instruction set is the kernel function that the programmer wrote.
- If a thread goes to sleep while waiting for a load/store then the stall can by hidden by executing another thread.
- It isn't essential that students know about how SM warp size or the maximum amount of threads an SM can handle. Instead treat this slide as a precursor to explain thread hierarchies which is important for CUDA programming.
- Also knowing about warps can allow the student to better understand how the GPU works.

Matrix Multiplication Algorithm

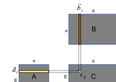


Figure 19: Reproduced from Nvidia's CUDA Toolkit Documentation

- This figure appears *Programming Massively Parallel Processors* page 65 and in `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory`.

Matrix Multiplication Algorithm

- Let A, B, and C be three matrices. Such that A is $m \times n$, B is $n \times o$, and C is the product of A and B (it's dimensions are $m \times o$).
- The classic algorithm states that every cell in C is the dot product of the corresponding row and column in A and B respectively.
- $c_{ij} = \mathbf{a_i} \bullet \mathbf{b_j}$ where $\mathbf{a_i}$ is row i A and $\mathbf{b_j}$ is column j in B.
  - $1 <= i <= m$ and $1 <= j <= o$
- Iterate through all values of i and j to calculate the values of C.
- Implementation consists of a double loop through all cells in C.

- Quick refresh for students. Matrix dimensions are specified as row $\times$ column.
- Classic method is to assume a 2D array. We'll talk about the 1-D implementation of a matrix.

Race Conditions

- When two or more threads are accessing and modifying a value in memory (global or shared) yet the order in which they do it is important. If that order isn't followed and it causes undefined behavior then we have a **race condition**
- E.X. A group of CUDA threads are binning the values in a large dataset. Incrementing counts in the bins can result in a race condition when the value changes before it is stored back in memory. Thereby creating inconsistencies.

- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-fence-functions` contains information about threadfence.
- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#synchronization-functions` contains information about thread synchronization.
- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions` contains information about atomic operations.

Matrix Data Structure.

- 1D or 2D array of doubles?
  - A 2D structure requires more work to transfer to the GPU. `cudaMalloc` and `cudaMemcpy` required for each row in the matrix.
  - 1D structure requires single `cudaMalloc` and `cudaMemcpy`.
- Access row 4 and col 2 in a 6×6 row-major matrix.
  - If a is 2D then `a[4][2]`
  - If a is 1D then `a[4 * 6 + 2]`
- Access row i and col j in a $m \times n$ row-major matrix.
  - `a[i * n + j]`

- For now lets assume that our matrix simply stores double precision values.
- Assumes that we start at index 0 instead of 1.
- TODO: Find descriptive image.

Atomic Operations

- A simple solution to a race condition is to make use of an atomic operation which are supported by CUDA. Here are a few that may be usefull.
  - `atomicAdd`, `atomicSub`, `atomicMin`, and `atomicMax`.
  - Additional operators can be found in `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions`
- Updating the `if` statement will fix the race condition. `atomicAdd` requires that we pass the address of the bin in `bins`.

```
if (index < rseq_len) atomicAdd( &bins[rseq[index] / bin_width], 1);
```

- Using atomic operations are slow since all threads writing the locked memory location have to wait.

Race Conditions

```
/// rseq and bins are pointers to an array of ints in global memory.
__global__ void bin_kernel_simple(int * rseq, int * bins,
                                  int bin_width, int rseq_len)
{
    const int index = threadIdx.x + blockIdx.x * blockDim.x;
    if ( index < rseq_len) bins[rseq[index] / bin_width]++;
}
```

- Simple program that bins up values from a random sequence of integers (`rseq`).
- `rseq` resides in global memory. When `bins[rseq[index] / bin_width]++` executed then there will be a race condition.
  - Why? Each SM is executing at most w threads concurrently. Therefore, it is possible to have $w * s$ (s is the count of SMs on the device) threads overwriting a single value in `bins`.

- A simple explanation of the binning program. First we calculate the access index into the random sequence `rseq`. If the calculated index is within the bounds of `rseq` then we update the corresponding bin in `bins` depending on the value from `rseq`.
- Since multiple threads may be accessing a single element `bins` then its value will become inconsistant.
- The insperation for this example was from Udacity's Parallel Programming Course. To view their orinal code go to `https://github.com/udacity/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%203%20Code%20Snippets/histo.cu`

2015-01-08

Massively Parallel Hardware
└─Advanced Memory Management
└─Shared Memory Allocation

- Since only a single variable can be dynamically allocated then if you want to manage more than one dynamic variable you will have to partition the data manually.

2015-01-08

Massively Parallel Hardware
└─Advanced Thread Management
└─Thread Syncronization

- Thread syncronization becomes a lot more important when making use of shared memory.

2015-01-08

Massively Parallel Hardware
└─Parallel Algorithms
└─Reduce

- Reduction can take an array of numbers and sum them up in parallel.
- Lesson 3 on the Udacity Intro to Parallel Programming Course.
- `https://www.youtube.com/watch?v=N1eQowSCdlw`

2015-01-08

Massively Parallel Hardware
└─Advanced Memory Management
└─Shared Memory Question

- Using `malloc` inside of a kernel will allocate the shared memory within global memory. Only the pointer resides in shared memory.
- This is bad considering the block performance is hindered due to the global memory allocation. To make matters worse the shared memory will be as slow as global memory. We do not want that.
- Always use the two previous methods to allocate onto the SM's L2 cache.

Reduce Kernel

- Simple example a reduction implemented in CUDA.

```
__shared__float partialSum[];

unsigned int t = threadIdx.x;
for( unsigned int stride = 1; stride < blockDim.x, strid *= 2)
  __syncthreads();
  if(t % (2 * stride) == 0) {
    partialSum[t] += partialSum[t + stride]
  }
}
```

- Example from the book **Programming Massively Parallel Processors** on page 100.
- The book provides a better implementation of a reduction on page 103. Perhaps open a discussion on how to further improve the kernel function.

Reduction as a Tree Complexity

- The work complexity of the tree reduction is $O(n)$ since the device will still need to perform $n-1$ operations to reduce a list of numbers.
- The step complexity of the tree reduction is $O(\log(n))$ since each level on the tree is independent and the height of a binary tree is $\log(n)$ where n is the number of leaves.

- Refer to *Lesson 3: Steps and Work* in Udacity's *Intro to Parallel Programming* course.
  - https://www.youtube.com/watch?v=V8TTrUdfpIY

Hillis-Steele Implementation

- Hillis-steele scan using a single block of threads.

```
#define SEQ_SIZE 1024
#define BLOCK_SIZE 1024
__global__ void hs_scan_kernel(int * in, int * out) {
  int tid = threadIdx.x;

  __shared__ int shared[BLOCK_SIZE];

  shared[tid + 1] = in[tid];
  __syncthreads();

  for( int d = 1; d <= log2(SEQ_SIZE); ++d ) {
    if( tid >= (1 << (d - 1))) {
      shared[tid] += shared[tid - (1 << (d - 1))];
    }
    __syncthreads();
  }

  out[tid] = shared[tid];
}
```

- Hillis-Steele algorithm implementation from section 39.2.1 from GPU Gems 3.

Scan

- Scan applies concepts similar to reduce.
- Take a sequence of values, apply a binary operator while keeping a cumulative output.
- Example:
  - INPUT: $S = [a_0, a_1, a_2, a_3, ...a_n], \quad i = I, \quad \oplus$ where S is the sequence and i is the identity for the operator $\oplus$.
  - OUTPUT:
  - Exclusive: $[I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), (I \oplus a_0 \oplus a_1 \oplus a_2), ...(I \oplus ... \oplus a_{n-1})]$

- A scan operation takes a sequence of values and provides a cumulative output given an operator.
- More information about the scan operator can be found at GPU Gems 3 `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html`.
- More information is also available at Udacity's online course **Intro to Parallel Programming** section three. Refer to the inclusive and exclusive scan videos.

2015-01-08

Massively Parallel Hardware
└─Parallel Algorithms
└─Blelloch

Blelloch

Up-Sweep Complexity

- What is the complexity of the down-sweep portion?
  - Work complexity?
  - Step complexity?

- The amount of work performed is $\log(n)$.
- The number of steps required is $\log(n)$.
- More information about Blelloch can be found in *Lesson 3* on Udacity's *Intro to Parallel Programming Course*

2015-01-08

Massively Parallel Hardware
└─Parallel Algorithms
└─Blelloch

Blelloch

Down-Sweep Complexity

- What is the complexity of the down-sweep portion?
  - Work complexity?
  - Step complexity?

- The amount of work performed is $\log(n)$.
- The number of steps required is $\log(n)$.
- More information about Blelloch can be found in *Lesson 3* on Udacity's *Intro to Parallel Programming Course*

2015-01-08

Massively Parallel Hardware
└─Parallel Algorithms
└─Compact

Compact

Example

- Find elements that are either a multiple of three or five.
- S is the set to compact.
  - $S = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$
- The condition to check on each element is $s \in S$ is $(s \mod 3 = 0)|(s \mod 5 = 0)$
- Resulting in the predicate set P.
  - $P = [0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1]$
- Applying the scan operator on P and output C
  - $C = [0, 0, 0, 1, 1, 2, 3, 3, 3, 4, 5, 5, 6, 6, 6]$
- Now map the items from S where each item P is 1 to the corresponding index in C into a new set N.
  - $N = [3, 5, 6, 9, 10, 12, 15]$

- Refer to Udacity's *Intro to Parallel Programing: Lesson 4* for more information on the compact operation.
  - `https://www.youtube.com/watch?v=GyYfg3ywONQ`

9.1.2 Concurrency Lectures

Concurrent Programming

January 8, 2015

Introduction

Introduction to using threads and thread management techniques.

Problem

- Why would we want to use threads?
- Lets say we have a service that performs a simple task. Take in sets of number sequences and sorts them using quicksort. $f(S) = S'$ such that $S = \{s_i | s_i = \{a_1, ...a_n\}, 1 <= i <= m; m, n \in \mathbb{N}\}$ and $S = \{S' | s_i \text{ is sorted}\}$. $f(S)$ will then call $Q(s_i)$ on each sequence m times.

Serial Example

- A simple program may contain a single loop in f that calls the quicksort function at each iteration.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    sort(*s_i);
  }
}

void sort(std::vector< int > & to_sort) { /* .... */ }
```

Serial Example

Advantages

- Easy to write.

Disadvantages

- Slow, takes about 2.2 seconds to complete the operation on a 3GHz Core 2 Duo.
- Most computers have at least two cores.
- Only single core is being used.
- Other cores wasted.

Parallel Example

- A program that utilizes threads to speed up the process would spawn multiple threads.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  std::vector< std::thread > threads;
  //Spawn thread to sort each subvector.
  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    threads.push_back(std::thread(sort, std::ref((*s_i))));
  }

  // Wait for each thread to complete its work.
  for(std::vector< std::thread >::iterator t = threads.begin();
      t != threads.end(); ++t)
  {
    (*t).join(); // or t->join();
  }
}
```


Parallel Example

Advantages

- Takes less time to execute. 1.3 seconds on a 3GHz Core 2 Duo.

Disadvantages

- More complex.
- Keep track of threads.
- New errors may arise.

Use Cases

- Operating Systems: Linux, Windows, and Unix.
- Graphical interfaces use event driven multithreading to preserve responsiveness.
- Games, separation of input, physics, and rendering.
- Web server technologies such as databases, search engines, and web servers.
- HMMER
- Bioinformatics.

History

Early Multithreading & Multitasking Systems

- Early Machines
  - Single process model.
  - Batch processing.
- Berkeley Timesharing System
  - Give processes **time-slots** of execution.
  - Memory is shared.
  - Computer remains usable for other operators.
- Unix
  - Processes now have dedicated memory.
  - Later, threading support added. Subprocesses that share memory with the processes.

Threading Architecture

Software

- Operating systems have built in thread management.
  - Distributed operating systems.
- Processes run separately.
- Pipes and sockets used for process communication.
- Subprocess support (threads) that share memory with processes.

Threading Models

- kernel threads (most kernel threads on Linux are processes).
- user threads (threads that processes spawn).

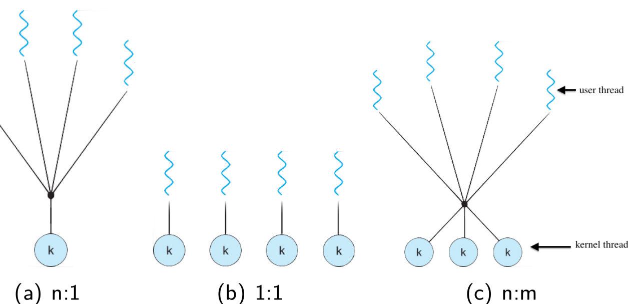
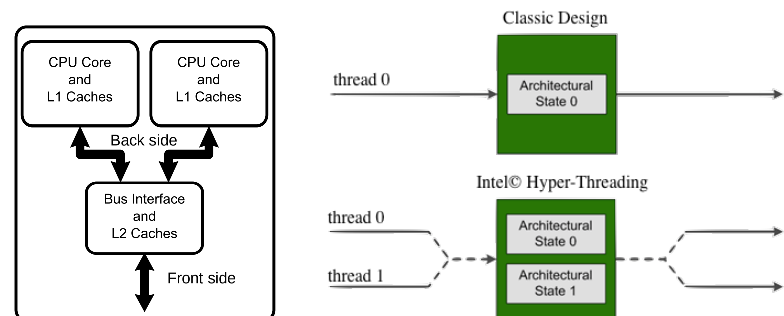


Figure 1: Threading Models: Retrieved from *Operating System Concepts*

Hardware

- Duplication of registers to store multiple states (Intel Hyper-threading).
  - Threads can still lock while waiting for CPU resources.
- Intel Hyper-Threading.
- Multiple cores.
- Multiple sockets.



Multithreaded Programming

Overview

- Minimal code example.
- Spawning a thread from a function.
- Terminating a thread (join, detach).
- Atomic Operation.
- Mutex (Locks).
- Semaphore.
- Lock-free data structure.

Minimal Working Example

```
void sort(std::vector< int > & to_sort) {
  std::sort(to_sort.begin(), to_sort.end());
}

void parallel_sorting(std::vector< std::vector<int> > & T) {
  std::vector< std::thread > threads;
  for(std::vector< std::vector< int > >::iterator s_i = T.begin();
        s_i != T.end(); ++s_i)
  {
    threads.push_back(std::thread(sort, std::ref(*s_i)));
  }

  for(std::vector< std::thread >::iterator t = threads.begin();
        t != threads.end(); ++t)
  {
    (*t).join();
  }
}

int main(int argc, char * argv[]) {
    std::vector< std::vector<int> > T;
    fill_with_vectors(T, 10000, 1000);
    parallel_sorting(T);
    return 0;
}
```


Threads in C++11

- After we have created `T` with vectors of random integers. We pass `T` to the `parallel_sorting` function.
- The first line of code `std::vector< std::thread > threads` is a vector that contains thread objects. If thread needs to be terminated with later then it is necessary to keep track of it.
- Second we iterate through the vectors in `T` and spawn a thread to sort each vector with:
  - `threads.push_back(std::thread(sort, std::ref(*s_i)));`
- `sort` is the function that defines the instructions for the thread instance.
- `std::ref(*s_i)` tells the thread constructor that `sort` requires `std::vector<int> &` as a parameter.

Defining a Thread Function

- Threads require a function to execute since they are subprocess.
- The thread function `sort` is a wrapper for `std::sort` from the C++ Standard Library (stdlib)

```
void sort(std::vector< int > & to_sort) {
    std::sort(to_sort.begin(), to_sort.end());
}
```


Terminating a Thread

- Going back to the `parallel_sorting` lets take a look at the section of code where the `join` method is called.
- There are two methods dealing with thread termination. First we can **join** a thread.
  - Joining a thread will cause the thread calling the join method to block until the thread completes its task.
- If the termination of the thread is not important to the state of the program then **detaching** a thread will cause the thread to continue executing until the OS destroys it when the program quits.

Thread Communication

Thread Communication

Suppose we have an application that records the number of clients serviced. What problem may arise from the code?

```
void handle_request(int & requests_served) {
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  /* Do stuff */
  requests_served++;
}

int main(int argc, char * argv[]) {
  int requests_served = 0;
  std::vector< std::thread > threads;
  for(int i = 0; i < 1000; i++) {
    threads.push_back(std::thread(handle_request,
      std::ref(requests_served)));
  }
  for(std::vector< std::thread >::iterator t = threads.begin();
        t != threads.end(); ++t)
  {
    t->join();
  }
  std::cout << "Handled " << requests_served << " requests." << std::endl;
  return 0;
}
```


Race Condition

- When `requests_served++` is executed a race condition may occur.
- `requests_served++` is not an atomic operation. Expanding it to machine code would result in:

$$register_1 = requests_served \quad (1)$$
$$register_1 = register_1 + 1 \quad (2)$$
$$requests_served = register_1 \quad (3)$$

- If a context change were to arise between lines 1 and 2 or 1 and 3. Then there is the possibility that *requests_served* will have changed due to another thread. Which would make $register_1$ inconsistant with *requests_served*.

Solution?

- Ensure the threads don't overstep other threads with atomic operations.
- atomic operations guarantee consistency across threads when a variable is modified.

Atomic Operations

- Atomic operations used to only use features provided by the operating system to guarantee process syncronization.
- Multicore processors now provide special instructions to aid the operating system.
- An atomic operation allows the modification of a single variable.
- C++11 stdlib provides atomic types.
- Rundown of the different atomic operations.

Atomic Operations

Code updated to make use of atomics.

Atomic Operation Example

```
void handle_request(std::atomic<int> & a_requests_served) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    /* Do stuff */
    a_requests_served++;
}

int main(int argc, char * argv[]) {
  std::atomic<int> a_requests_served(0);
  std::vector< std::thread > threads;
  for(int i = 0; i < 1000; i++) {
    threads.push_back(std::thread(handle_request,
      std::ref(requests_served)));
  }
  for(std::vector< std::thread >::iterator t = threads.begin();
      t != threads.end(); ++t)
  {
    t->join();
  }
  std::cout << "Handled " << a_requests_served << " requests." << std::endl;
  return 0;
}
```


Critical Section

- Any section of code that reads or writes to data that is shared amongst threads.
- Must satisfy three requirements ensure consistency.
  1. **Mutual Exculsion**: If a thread is in a critical section then other threads must wait for it to exit the section.
  2. **Progress**: A thread cannot wait inside of a critical section. Waiting can cause a *deadlock*.
  3. **Bounded Waiting**: Threads shall not hoard the critical section.

Mutexes

- Atomic operations are simple, but you cannot lock multiple lines of code.
- Mutexes allow you to declare a critical section and limit access to a single thread.
- Mutexes use locks to identify a critical section of code.

Mutex Example

- The *Atomic Operation Example* has been modified to use a mutex instead.

```
std::mutex mlock;

void handle_request(int & requests_served) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    mlock.lock();
    /* Do stuff */
    requests_served++;
    mlock.unlock();
}
```


Semaphores

- A mutex is a semphore that only allows a single thread to access a critical section.
- More advanced data structure that provides mutual exclusion access to a critical section.
- Unlike a classic mutex a semaphore keeps count of the threads that want to access a resource.
- Designed to allow multiple threads access a critical section.
- Two operations used.
  - `wait(semaphore)`: Thread is blocked until another thread calls signal.
  - `signal(semaphore)`: Thread calls signal to indicate exit of critical section and allow another thread to enter.

Building a Semaphore in C++11

- Tools Needed
  - mutex: See example in previous slides.
  - condition_variable: Class that manages the execution of threads that call wait on a given lock.
  - primitive to keep track of number of threads in the critical zone.

Building a Semaphore in C++11

Semaphore C++11

```
class semaphore {
    private:
        std::mutex   _mtx;
        std::condition_variable _cv;
        int _count;

    public:
        semaphore(int count = 1): _count(count) {}
        void signal() {
            std::lock_guard<std::mutex> lck(_mtx);
            _count++;
            _cv.notify_one();
        }
        void wait() {
            std::unique_lock<std::mutex> lck(_mtx);
                        /* C++11 Anonymous (lamda) Function  */
            _cv.wait(lck, [this](){ return _count > 0; });
            _count--;
        }
};
```

Readers-Writers Problem

- The semaphore can then be used to synchronize communication between a writer thread and set of reader threads.

Reader

```
void writer(somedata & shared_data, semaphore & wrt) {
  while(1) {
    wrt.wait();
    \\\< Write to shared data.
    wrt.signal();
    total_writes++;
  }
}
```


Readers-Writers Problem

Writer

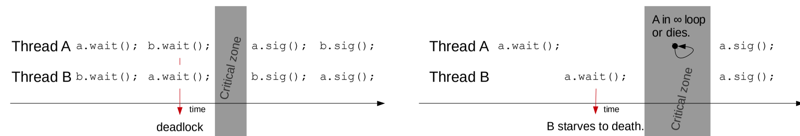
```
void reader(somedata & shared_data, semaphore & wrt,
    semaphore & mtx) {
  while(1) {
    mtx.wait();
    read_count++;
    if(read_count == 1) {
      wrt.wait();
    }
    mtx.signal();

    rd.wait();
    \\\< Read the shared data.
    rd.signal();

    mtx.wait();
    read_count--;
    if(read_count == 0) {
      wrt.signal();
    }
    mtx.signal();
  }
}
```

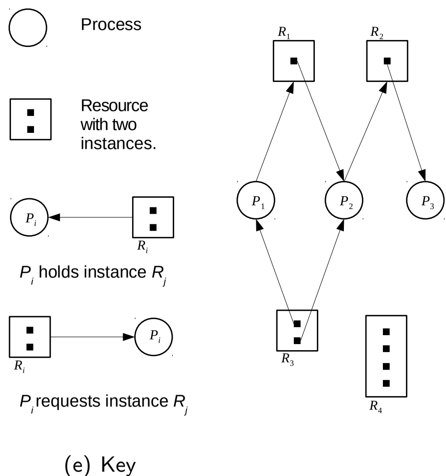

Deadlocks

- What can cause deadlocks? There exists four conditions.
  1. **Mutual exclusion**: There exists at least a single resource that can be held in a non-sharable mode.
  2. **Hold and wait**: Thread holds onto a resource and waits for another resource to be freed.
  3. **No preemption**: Threads cannot be stripped of their resources by other threads.
  4. **Circular wait**: When two or more threads are holding and waiting on shared resources.



(c) Mutual Exclusion (d) No Preemption | Hold & Wait

Resource-Allocation Graph



(e) Key

Figure 2: Adopted from *Operating System Concepts*

Lock-Free Programming

- Another method to prevent deadlocks is to use lock-free constructs.
- A lock-free structure guarantees throughput, but doesn't prevent starvation.
- Need to make use of atomic operations to construct the lock free data structure.
  - Atomic types, for example `std::atomic<T>`.
  - Atomic compare and swap (CAS). Such as `std::atomic_compare_exchange_*`

Advantages

- Guarantees no deadlocks.
- Scalable.

Disadvantages

- May be slower than lock-based structures.
- More difficult to implement.

Wait-Free Programming

- Subset of lock-free programming that ensures all threads complete their task within a finite set of steps.

Advantages

- Eliminates thread starvation.
- Scalable.

Disadvantages

- Runs slower than locked-free structures.

Inter-Process Communication

Forking

- How about process level parallelism? Use the `fork` command.
- *forking* a process will create a *child* (new) process that is an exact copy of the *parent* (calling) process except:
  - Locks are not preserved (including file locks).
  - Other threads from the parent process are not copied. Only the thread that forked the process is copied.
  - Process IDs are not preserved. The child will be assigned new IDs.
  - For more exceptions refer to the POSIX.1-2008 specifications for `fork()`.

Forking

- How many times will `printf` be called? How will we know what `printf` was called by the root process?

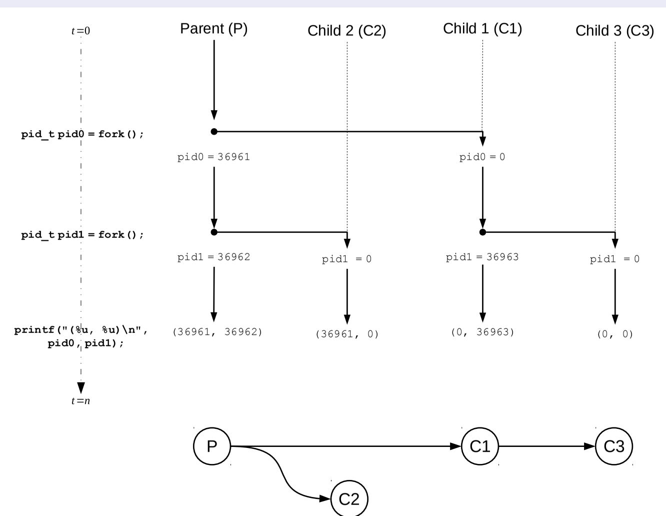
Forking Example

```
int main(int argc, char * argv[])
{
  pid_t pid0 = fork(); // fork returns 0 to the child process.
  pid_t pid1 = fork();

  printf("(%u, %u)\n", pid0, pid1); // print two unsiged numbers.
}
```


Forking

Forking Diagram



Pipes

- Shared between the parent and child process (or multiple children). Different processes cannot share pipes.
- Enables communication between the parent and child process.
- Ordinary pipes provide unidirectional communication so that one end can be written to (write-end) and the other read from (read-end).

Pipe

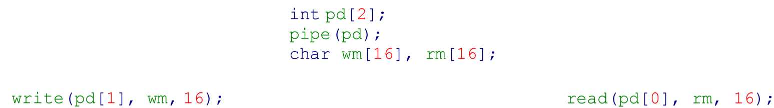


Figure 3: Visual Representation of a Pipe: Adopted from *Operating System Concepts*.

Named Pipes: FIFO

- Referred to as First-In First-Out (FIFO) on POSIX/Unix sytems.
- Enables communication between separate processes.
- Represented as a special file handle that points to a location in memory.
- Functionality similar to pipes; except bidirectional communiction is possible. Unlike a pipe, reading and writing from the same file descriptor is possible.
- More overhead.
  1. Create the FIFO file.
  2. Open the FIFO.
  3. Read/Write to the FIFO.

Sockets

- Sockets provide full-duplex communication streams.
- Remote connections across the network.
- Primary tool to setup client-server communication model.

Advantages

- Dynamic: Allows the distribution of processes across multiple machines.
- More abstracted since the machines could be running their own operating system.

Disadvantages

- More overhead to set up.
  - Create a socket.
  - Bind the socket to an address.
  - Connect to the socket.
- Slower than FIFOs and Pipes since the data passes through the network stack.

Threading Revisited

Thread Patterns Revisited

- Event driven designs.
- Thread pools
- Schedulers

Event Driven Designs

- Performance is not always a priority when using threads.
- Responsiveness may be another reason.
- Graphical User Interfaces, servers, and other producer-consumer patterns.

Thread Pools

- Solution for when system resources are limited and thrashing may happen.
- It is expensive to create threads. Therefore create a set of threads for later use.
- Pass jobs to the thread pool which then hands the jobs over to the threads.
- Thread pools are scalable, depending on the system resources the number of threads can be increased or decreased.

Schedulers

- Pools of threads do not guarantee optimal execution.
- Different threads will have varying execution times.
- Use a scheduler to ensure the desired optimal performance is achieved.
- Using a defined set of heuristics the scheduler will dequeue and run the desired threads from the pool.

Limitations

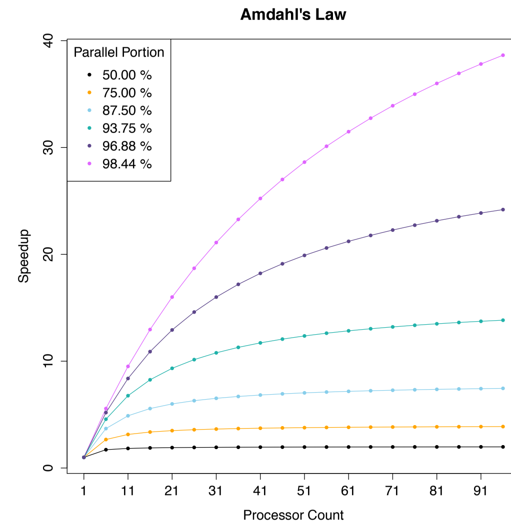
Amdahl's Law

- Determine the potential code speedup with Amdahl's Law.
  - This version assumes that a case of parallelization.

$$T(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

- P is a value between 0 and 1. P is the fraction of the program that is executed in parallel.
- As P approaches 1 then the program becomes more parallelized.
- If $P == 1$ then the program is solving an *embarrassingly parallel* problem. The speedup is linear to the number of cores.
- N is the count of processors.
- Amdahl's law assumes a fixed problem size. Which causes a diminishing returns effect as the number of cores increase.

Amdahl's Law



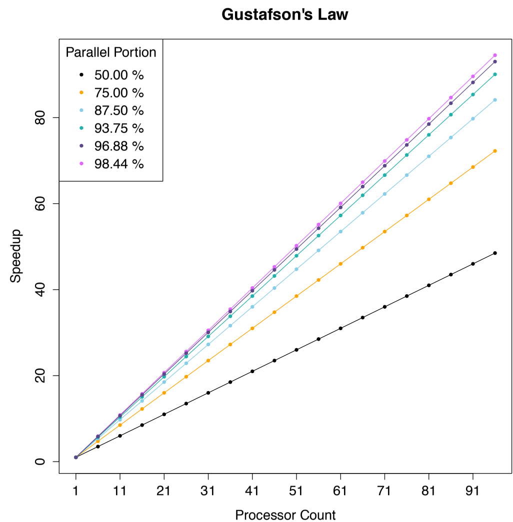
Gustafson's Law

- Unlike Amdahl's Law, Gustafson's Law assumes that the program will work on larger problems to utilize all of the processors.

$$S(N) = N - P(N-1)$$

- P is the fraction of the program that is parallel.
- N is the number of processors.

Gustafson's Law



Thrashing / IO / Starvation / Busses

- If an application spawns too many threads then the CPU will spend more time on context switches between the threads instead of executing the threads.
- Threads that perform a lot of I/O bound operations will be limited to the speed of the resources that they share.
  - Data that needs to be passed through a bus will limit thread performance.

Problem

- Why would we want to use threads?
- Lets say we have a service that performs a simple task. Take in sets of number sequences and sorts them using quicksort. $f(S) = S'$ such that $S = \{s_i | s_i = \{a_1, ... a_n\}, 1 <= i <= m; m, n \in \mathbb{N}\}$ and $S = \{S' | s_i \text{ is sorted}\}$. $f(S)$ will then call $Q(s_i)$ on each sequence m times.

- Quicksorting sets of subsequences is a good example since the subsequences can be of any size and the quicksort's performance may vary.

Introduction

Introduction to using threads and thread management techniques.

NOTE: To whom ever is reading this document. Pandoc processes divs that inherit from the 'notes' class as a note. These will not show up in your slides. To compile slides with notes run `make notes`. You will need to be running pandoc –version >= 1.12.A

- All of the examples in these lectures will make use of C++11 features. C++11 provides thread abstractions that should in theory work on any OS.

Parallel Example

- A program that utilizes threads to speed up the process would spawn multiple threads.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  std::vector< std::thread > threads;
  //Spawn thread to sort each subvector.
  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    threads.push_back(std::thread(sort, std::ref((*s_i))));
  }

  // Wait for each thread to complete its work.
  for(std::vector< std::thread >::iterator t = threads.begin();
      t != threads.end(); ++t)
  {
    (*t).join(); // or t->join();
  }
}

void sort(std::vector< int > & to_sort) { /* .... */ }
```

- Spawning 10,000 threads probably isn't a smart idea.

Serial Example

- A simple program may contain a single loop in f that calls the quicksort function at each iteration.

```
int main() {
  // A vector with 10,000 vectors of size 1,000 with unsorted integers.
  std::vector< std::vector< int > > S = { std::vector<int>(1000),
                                          /*...*/
                                          std::vector<int>(1000) };

  for(std::vector< std::vector< int > >::iterator s_i = S.begin();
      s_i != S.end(); ++s_i)
  {
    sort(*s_i);
  }
}

void sort(std::vector< int > & to_sort) { /* .... */ }
```

- Code is in `Concurrency/parallel_soring_example`.

Early Multithreading & Multitasking Systems

- Early Machines
  - Single process model.
  - Batch processing.
- Berkeley Timesharing System
  - Give processes **time-slots** of execution.
  - Memory is shared.
  - Computer remains usable for other operators.
- Unix
  - Processes now have dedicated memory.
  - Later, threading support added. Subprocesses that share memory with the processes.

- `http://www.faqs.org/faqs/os-research/part1/section-10.html` provides a nice and simple intro to the history of threading.
- Chapter 4 in *Operating System Concepts* contains additional information about threads.
- `http://en.wikipedia.org/wiki/Unix`

Use Cases

- Operating Systems: Linux, Windows, and Unix.
- Graphical interfaces use event driven multithreading to preserve responsiveness.
- Games, separation of input, physics, and rendering.
- Web server technologies such as databases, search engines, and web servers.
- HMMER
- Bioinformatics.

- `http://www.valvesoftware.com/publications/2007/GDC2007_SourceMulticore.pdf`
- Web server technologies such as Apache, Nginx, and Microsoft ISS emplay multithreading to separate requests.
- `http://en.wikipedia.org/wiki/Event-driven_programming`

Hardware

- Duplication of registers to store multiple states (Intel Hyper-threading).
  - Threads can still lock while waiting for CPU resources.
- Intel Hyper-Threading.
- Multiple cores.
- Multiple sockets.

- Intel Hyper-Threading Image adopted from `https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology?language=es`.
- Multi-core processor diagram was retrieved from `https://en.wikipedia.org/wiki/Multi-core_processor`

Threading Models

- kernel threads (most kernel threads on Linux are processes).
- user threads (threads that processes spawn).

Figure 1: Threading Models. Retrieved from *Operating System Concepts*

- Refer to Chapter 4: **Operating System Concepts 8e.** for more information on threading models.
  - Book states that Linux uses the one-to-one model.

- `#include<thread>` a C++11 wrapper for various system threads. In the case of Linux and Unix it is a PThread (POSIX) thread wrapper.

- Includes at the top of the example.
  - `stdlib.h`, `iostream`, `algorithm`, `thread`, and `utils.h`.
- `utils.h` contains `fill_with_vectors(std::vector< std::vector<int> >, int, int)`.

- What problems may arise from the code?
  - Race conditions. *Critical Section*
  - Operations that appear atomic may not be once compiled.
  - Provide Assembly example of race condition.

- The STL is not the C++ Standard Library. It just stands for Standard Library which was created by Alexander Stepanov.
  - `http://stackoverflow.com/questions/5205491/whats-this-stl-vs-c-standard-library-fight-all-about`

Semaphores

- A mutex is a semphore that only allows a single thread to access a critical section.
- More advanced data structure that provides mutual exclusion access to a critical section.
- Unlike a classic mutex a semaphore keeps count of the threads that want to access a resource.
- Designed to allow multiple threads access a critical section.
- Two operations used.
  - `wait(semaphore)`: Thread is blocked until another thread calls signal.
  - `signal(semaphore)`: Thread calls signal to indicate exit of critical section and allow another thread to enter.

- Explanation and definition of semaphores came from lecture 6 in the lecture materials from *CSE 120: Principles of Operating by Systems Alex C. Snoeren* (also included in the resources directory).
- Page 189 from the *Art of Multiprocessor Programming* by *Maurice Herlihy & Nir Shavit*.

Critical Section

- Any section of code that reads or writes to data that is shared amongst threads.
- Must satisfy three requirements ensure consistency.
  1. **Mutual Exculsion**: If a thread is in a critical section then other threads must wait for it to exit the section.
  2. **Progress**: A thread cannot wait inside of a critical section. Waiting can cause a *deadlock*.
  3. **Bounded Waiting**: Threads shall not hoard the critical section.

- Refer to page 228 chapter 6 in *Operating System Concepts* 8th Edition.

Readers-Writers Problem

- The semaphore can then be used to synchronize communication between a writer thread and set of reader threads.

Reader

```
void writer(somedata & shared_data, semaphore & wrt) {
  while(1) {
    wrt.wait();
    \\\< Write to shared data.
    wrt.signal();
    total_writes++;
  }
}
```

- Refer to page 241 from *Operating System Concepts 8 e.* or slide 10 in the lecture slides provided by Alex C. Snoeren CSE 120.
- Refer to `main.cpp` in the `semaphore_example` for a working demonstration of the readers/writers problem.

Building a Semaphore in C++11

Semaphore C++11

```
class semaphore {
    private:
        std::mutex   _mtx;
        std::condition_variable _cv;
        int _count;

    public:
        semaphore(int count = 1): _count(count) {}
        void signal() {
            std::lock_guard<std::mutex> lck(_mtx);
            _count++;
            _cv.notify_one();
        }
        void wait() {
            std::unique_lock<std::mutex> lck(_mtx);
                        /* C++11 Anonymous (lamda) Function  */
            _cv.wait(lck, [this](){ return _count > 0; });
            _count--;
        }
};
```

- It most cases it is recommended to use an existing library for semaphores.
- Such as boost or the POSIX defined `semaphore.h`
- lock_guard will lock within the scope using a mutex.
- unique_lock allows the condition variable to associate a set of threads to a common lock and defer their execution.
- condition_variable
  - `notify_one()` will wake up a sleeping thread.
  - `wait(unique_lock &, predicate)` will put a thread to sleep. `predicate` determines if a thread should go back to sleep after a spurious wakeup.
  - `predicate` is a C++11 anonymous function.

2015-01-08

Concurrent Programming
└Thread Communication
└Deadlocks

Deadlocks

- Chapter 7 from *Operating System Concepts* covers deadlocks and deadlock avoidance.
  - Section 7.2 covers the four conditions necessary for a deadlock.
  - `http://nob.cs.ucdavis.edu/classes/ecs150-1999-02/dl-cond.html`
- Figure (d) Thread A goes into the critical zone and holds a resource and waits for another resource to be freed. Thread B starves while waiting.

2015-01-08

Concurrent Programming
└Thread Communication
└Readers-Writers Problem

Readers-Writers Problem

- Refer to page 241 from *Operating System Concepts 8 e.* or slide 10 in the lecture slides provided by Alex C. Snoeren CSE 120.
- Refer to `main.cpp` in the `semaphore_example` for a working demonstration of the readers/writers problem.

2015-01-08

Concurrent Programming
└Thread Communication
└Lock-Free Programming

Lock-Free Programming

- Another method to prevent deadlocks is to use lock-free constructs.
- A lock-free structure guarantees throughput, but doesn't prevent starvation.
- Need to make use of atomic operations to construct the lock free data structure.
  - Atomic types, for example std::atomic<T>.
  - Atomic compare and swap (CAS). Such as std::atomic_compare_exchange_*

Advantages

- Guarantees no deadlocks.
- Scalable.

Disadvantages

- May be slower than lock-based structures.
- More difficult to implement.

- Page 60 from *The Art of Multiprocessor Programming* provides a nice definition of lock-free programming.
- Dr. Dobbs has a nice article about lock-free programming.
  - `http://www.drdobbs.com/lock-free-data-structures/184401865.`
- Lock-Free programming is a difficult subject.
- libcds provides a set of concurrent data structures `http://libcds.sourceforge.net/`
- Nice set of Youtube videos about lock-free programming.
  - CppCon by Herb Sutter Part 1: `https://www.youtube.com/watch?v=c1g09aB9nbs`
  - CppCon by Herb Sutter Part 2:
  -

2015-01-08

Concurrent Programming
└Thread Communication
└Resource-Allocation Graph

Resource-Allocation Graph

Figure 2: Adopted from *Operating System Concepts*

- Chapter 7.4 Deadlock Prevention from *Operating System Concepts* and the Chapter 7 deadlock slides has more information.
- Represent resource usage as a directed acyclic graph.
  - If any cycles exist then there may exist a deadlock.

2015-01-08

Concurrent Programming
└─Inter-Process Communication
└─Forking

Forking

- How many times will `printf` be called? How will we know what `printf` was called by the root process?

Forking Example

- `fork` will return the child pid to the parent and 0 to the child.
- `printf` requires `#include <stdio.h>`
- `fork` requires `#include <unistd.h>`
- Program Output:
  (36961, 36962)
  (36961, 0)
  (0, 36963)
  (0, 0)

2015-01-08

Concurrent Programming
└─Inter-Process Communication
└─Forking

Forking

- How about process level parallelism? Use the `fork` command.
- *forking* a process will create a *child* (new) process that is an exact copy of the *parent* (calling) process except:
  - Locks are not preserved (including file locks).
  - Other threads from the parent process are not copied. Only the thread that forked the process is copied.
  - Process IDs are not preserved. The child will be assigned new IDs.
  - For more exceptions refer to the POSIX.1-2008 specifications for `fork()`.

- Provide simple code example and image.
- Section 4.4.1 in *Operating System Concepts* mentions that there exists fork implementations that will duplicate all threads when `fork` is called. Most versions of `fork` will only duplicate the thread that called the function. The book doesn't provide any references so it may be safe to assume that `fork` only duplicates the calling thread.
- `man 2 fork` on OS X and Linux will provide usage details.
- The Open Group provides the specifications of `fork` on a POSIX.1-2008 compatable system. `http://pubs.opengroup.org/onlinepubs/9699919799/`

2015-01-08

Concurrent Programming
└─Inter-Process Communication
└─Pipes

Pipes

- Shared between the parent and child process (or multiple children). Different processes cannot share pipes.
- Enables communication between the parent and child process.
- Ordinary pipes provide unidirectional communication so that one end can be written to (write-end) and the other read from (read-end).

Pipe

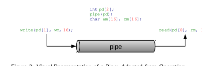


Figure 3: Visual Representation of a Pipe. Adopted from *Operating System Concepts*.

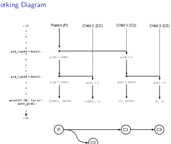
- Provide simple code example and image.
- Refer to *Operating System Concepts* section 3.6.3: Pipes.
- Refer to the man pages
  - `man 2 pipe`
  - `man 2 read`
  - `man 2 write`

2015-01-08

Concurrent Programming
└─Inter-Process Communication
└─Forking

Forking

Forking Diagram



- A Visual example of the process behind `fork()`.
- The graph below shows the inheritance order.

Sockets

- Sockets provide full-duplex communication streams.
- Remote connections across the network.
- Primary tool to setup client-server communication model.

Advantages

- Dynamic: Allows the distribution of processes across multiple machines.
- More abstracted since the machines could be running their own operating system.

Disadvantages

- More overhead to set up.
  - Create a socket.
  - Bind the socket to an address.
  - Connect to the socket.
- Slower than FIFOs and Pipes since the data passes through the network stack.

- Refer to the man pages:
  - `man 2 socket`

Named Pipes: FIFO

- Referred to as First-In First-Out (FIFO) on POSIX/Unix sytems.
- Enables communication between separate processes.
- Represented as a special file handle that points to a location in memory.
- Functionality similar to pipes; except bidirectional communiction is possible. Unlike a pipe, reading and writing from the same file descriptor is possible.
- More overhead.
  1. Create the FIFO file.
  2. Open the FIFO.
  3. Read/Write to the FIFO.

- Provide simple code example and image.
- Refer to the operating systems book for more information.
- Refer to the man pages.
  - `man 2 mkfifo`
  - `man 2 open`
  - `man 2 read`
  - `man 2 write`
  - `man 2 close`

Amdahl's Law

- Determine the potential code speedup with Amdahl's Law.
  - This version assumes that a case of parallelization.

$$T(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

- P is a value between 0 and 1. P is the fraction of the program that is executed in parallel.
- As P approaches 1 then the program becomes more parallelized.
- If $P == 1$ then the program is solving an *embarrassingly parallel* problem. The speedup is linear to the number of cores.
- N is the count of processors.
- Amdahl's law assumes a fixed problem size. Which causes a diminishing returns effect as the number of cores increase.

- `http://en.wikipedia.org/wiki/Amdahl%27s_law`
- `http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980`
- Amdahl's Law assumes the dataset that the program is working on is static in size.
- For example a program that only parallelizes a fixed sized problem would follow this rule. Let's say that a program will always multiply two 1000x1000 matrices. Then the speedup of that program will follow Amdahl's law as the number of processors increase.

Schedulers

- Pools of threads do not guarantee optimal execution.
- Different threads will have varying execution times.
- Use a scheduler to ensure the desired optimal performance is achieved.
- Using a defined set of heuristics the scheduler will dequeue and run the desired threads from the pool.

- There isn't any defined thread pool class in C++11.

Gustafson's Law

- Unlike Amdahl's Law, Gustafson's Law assumes that the program will work on larger problems to utilize all of the processors.

$$S(N) = N - P(N - 1)$$

- P is the fraction of the program that is parallel.
- N is the number of processors.

- http://en.wikipedia.org/wiki/Gustafson%27s_law
- http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980

9.2 Projects

9.2.1 Matrix Multiplication

kernels.cu

```
#include <sstream> // stringstream
#include <stdexcept> // invalid_argument
#include <stdio.h> // printf
#include <cmath> // ceil

#include "kernels.h"
#include "Matrix.hpp"

namespace project
{

Matrix Matrix::cuda_multiply_by(Matrix& B, float &cuda_ms)
{
    if(this->c != B.r)
    {
        std::stringstream msg;
        msg << "Cannot multiply a " << this->r << "x" << this->c
            << " matrix and " << B.r << "x" << B.c << " matrix together";
        throw std::invalid_argument( msg.str() );
    }

    // Step one allocate the resources onto the GPU.
    double *AD; //allocate space on the device for the data in Matrix A.
    int sizeA = this->c * this->r * sizeof(double);
    cudaMalloc((void**)&AD, sizeA);
    cudaMemcpy(AD, &this->D.front(), sizeA, cudaMemcpyHostToDevice);

    double *BD; // allocate for Matrix B
    int sizeB = B.c * B.r * sizeof(double);
    cudaMalloc((void**)&BD, sizeB);
    cudaMemcpy(BD, &B.D.front(), sizeB, cudaMemcpyHostToDevice);

    double * CD;
    double * CD_host = new double[this->r * B.c];
    Matrix C(this->r, B.c, 0.0);
    int sizeC = C.r * C.c * sizeof(double);
```

```
cudaMalloc((void**)&CD, sizeC);

// Step two setup the grids of thread blocks.
//const dim3 blockSize(blockSide, blockSide, 1);
unsigned int blk_w = 32, blk_h = 16;
const dim3 blockSize(blk_w, blk_h, 1);

// Dynamically determine the grid size
// I could use the standard casting notation but this is easier to read.
unsigned int grd_w = std::ceil(double(C.c) / double(blk_w));
unsigned int grd_h = std::ceil(double(C.r) / double(blk_h));
const dim3 gridSize(grd_w, grd_h, 1);

// Step three perform the computations.
// Create the cuda events for time monitoring.
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Run the kernel
cudaEventRecord(start);
KernelMatrixMultiply<<<gridSize,blockSize>>>(AD, BD, CD, this->c,
        this->c, this->r, B.c);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
cudaEventElapsedTime(&cuda_ms, start, stop);

//Step 4: Pull the data from the GPU.
cudaMemcpy(CD_host, CD, sizeC, cudaMemcpyDeviceToHost);
for(unsigned int i = 0; i < this->r * B.c; i++)
{
    C.D[i] = CD_host[i];
}

// Free memory allocated on teh GPU.
cudaFree(AD);
cudaFree(BD);
cudaFree(CD);

return C;
```

```
}

__global__ void KernelMatrixMultiply(const double * AD, const double * BD,
                                     double * CD, const unsigned int m,
                                     const unsigned int colsA,
                                     const unsigned int rowsA,
                                     const unsigned int colsB)
{
    const int row = blockIdx.y * blockDim.y + threadIdx.y;
    const int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row < rowsA && col < colsB) // stay within the matrix bounds.
    {
        double dot = 0.0;
        // Performing math using double makes the GPU operate slower.
        for(unsigned int j = 0; j < m; j++)
        {
            dot += AD[row * colsA + j] * BD[colsB * j + col];
        }
        CD[colsB * row + col] = dot;
    }
}

}
```

kernels.h

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

#ifndef KERNELS_H
#define KERNELS_H

namespace project
{
    //!
    //! GPU kernel function that performs matrix multiplication.
    //!
    __global__ void KernelMatrixMultiply(const double * AD, const double * BD,
                                         double * CD, const unsigned int m,
                                         const unsigned int colsA,
```

```
                                        const unsigned int rowsA,
                                        const unsigned int colsB);
}



#endif
```

main.cpp

```
#include <iostream> // cout, endl
#include <sstream> // stringstream
#include <stdexcept> // invalid_argument
#include <algorithm> // copy
#include <utility> // swap
#include <ctime> // clock
#include <chrono> // time conversion to milliseconds.
#include <string> // string
#include <array> //array
#include <cassert> // assert

#include <sysexits.h> // Program termination codes for linux! Woot woot! :)
#include <getopt.h> // for parsing command line arguments.

#include "Matrix.hpp"

//--- Function Prototypes ---//
void stdin_to_two_matrices(project::IMatrix & A, project::IMatrix & B, char delimiter);
float optarg_to_float(const char * optarg);
void dimstring4dims(char * dimensions, unsigned int &r, unsigned int &c);
void print_help();
int run_tests();

int main(int argc, char * argv[])
{
    int help_flag = 0;
    int gpu_flag  = 0;

    const struct option longopts[]
    {
        // name,              has arg,                flag,         val
```

```
    {"help",             no_argument,          &help_flag,     1},
    {"A-dimensions",     required_argument,    NULL,          'a'},
    {"B-dimensions",     required_argument,    NULL,          'b'},
    {"delimiter",        required_argument,    NULL,          'd'},
    {"gpu",              no_argument,          &gpu_flag,     'g'},
    {NULL,               0,                    NULL,           0 }
};

// you need this for the short options.
const char * short_opts = "hga:b:d:";

char delimiter = ' ';
unsigned int Ar = 2, Ac = 2, Br = 2, Bc = 2;

int longopts_index = 0;
int iarg = 0;
while((iarg = getopt_long(argc, argv, short_opts, longopts,
                &longopts_index)) != -1)
{
    switch(iarg)
    {
        case 'h':
            /// print out help information
            print_help();
            return EX_OK;
            break;
        case 'a':
            /// set the dimensions of matrix A
            dimstring4dims(optarg, Ar, Ac);
            break;
        case 'b':
            /// Set the dimensions of B
            dimstring4dims(optarg, Br, Bc);
            break;
        case 'g':
            /// Set GPU flag
            gpu_flag = true;
           break;
        case 'd':
            /// Set delimiter
            delimiter = optarg[0];
```

```
                break;
            default:
                std::cout << "IARG: " << std::to_string(iarg) << std::endl;
                print_help();
                return EX_IOERR;
        }
    }

    project::Matrix A(Ar, Ac, 0.0);
    project::Matrix B(Br, Bc, 0.0);
    stdin_to_two_matrices(A, B, delimiter);

    if(gpu_flag) {
        float cuda_ms = 0.0;
        project::Matrix C = A.cuda_multiply_by(B, cuda_ms);
        std::cout << project::IMatrix::dump_to_string(C, delimiter) << std::endl;
    } else {
        project::Matrix C = A.multiply_by(B);
        std::cout << project::IMatrix::dump_to_string(C, delimiter) << std::endl;
    }

    return EX_OK;
}

///
/// Takes in two initialized matrices and populates with values from stdin
///
/// \param[in, out] An initialized IMatrix object.
/// \param[in, out] An initialized IMatrix object.
///
void stdin_to_two_matrices(project::IMatrix & A, project::IMatrix & B, char delimiter) {
    unsigned int i = 0;

    for(std::string line; std::getline(std::cin, line);) {
        std::stringstream ss(line);
        std::string number;
        while(std::getline(ss, number, delimiter)) {
            if(number.size() >= 1) {
                if(i < (A.get_row_count() * A.get_col_count()) ) {
                    A.set_value_at(i, std::stof(number));
                } else {
```

```
                int bi = i - A.get_row_count() * A.get_col_count();
                B.set_value_at(bi, std::stof(number));
            }
            i++;
        }
    }
  }
}

///
/// Takes in an optarg (char pointer) and converts it into a
/// float.
///
/// \param[in] optarg The optarg char * from getopt.h
/// \returns a float representation of the optarg value.
float optarg_to_float(const char * optarg)
{
    std::string *tmp = new std::string(optarg);
    int flt = std::stof(*tmp);
    delete tmp;
    return flt;
}

///
/// Takes in the dimensions that the user entered as text and extracts the rows
/// and columns.
///
/// \param[in] the dimensions the user entered.
/// \param[in, out] a pointer to the row placeholder.
/// \param[in, out] a pointer to the column placeholder.
void dimstring4dims(char * dimensions, unsigned int &r, unsigned int &c)
{
    std::string dims = dimensions;



    unsigned int i = 0;
    for(std::string::iterator it = dims.begin(); it != dims.end(); ++it)
    {
        if(*it == '=') {
            std::cerr << "Get rid of the '=' character in your dimensions parameter: `"
```

```
                    << dimensions << "`" << std::endl;
            std::exit(EX_IOERR);
        }
        if(*it == 'x' || *it == 'X')
        {
            break;
        }
        i++;
    }

    r = std::stoi(dims.substr(0,i).c_str());
    c = std::stoi(dims.substr(i + 1, dims.length() - i).c_str());
}

void print_help()
{

    std::cout << "\nSYNOPSIS\n";
    std::cout << "\t./matrixMultiply --gpu --A-dimensions RxC --delimiter ' '"
              << "--B-dimensions RxC \n";
    std::cout << "\t./matrixMultiply -g -a RxC -b RxC -d ' ' \n";

    std::cout << "Description\n";
    std::cout << "\t-h, --help\n\t\tPrint usage information.\n\n";
    std::cout << "\t-a, --A-dimensions=RxC\n"
              << "\t\tExpects a string of the format RxC where R and C are "
              << "two integers\n"
              << "\t\trepresenting the dimensions of matrix A\n\n";
    std::cout << "\t-b, --B-dimensions=RxC\n"
              << "\t\tExpects a string of the format RxC where R and C are "
              << "two integers\n"
              << "\t\trepresenting the dimensions of matrix B\n\n";
    std::cout << "\t-g, --gpu\n"
              << "\t\tFlag that enables matrix multiplication with the GPU\n";

    std::cout << "EXAMPLE\n";
    std::cout << "\techo \"12.0 16.0 9.0 2.0 4.0 5.0 0.0 9.0 8.0 4.0 0.0 1.0 2.0 3.0 5.0 9.0\" |
 ./$(BIN) -a 2x4 -b 4x2 -d ' ' -g --\n\n";
}
```

Matrix.cpp

```
#include <sstream> // stringstream
#include <stdexcept> // invalid_argument
#include <algorithm> // copy
#include <utility> // swap

#include "Matrix.hpp"

namespace project
{

Matrix::Matrix(unsigned int rows, unsigned int cols, double init_val)
    : IMatrix(rows, cols, init_val)
{
    unsigned int len = rows * cols;
    this->D = std::vector<double>(len, init_val);
    this->c = cols;
    this->r = rows;
}

unsigned int Matrix::get_row_count()
{
    return this->r;
}



unsigned int Matrix::get_col_count()
{
    return this->c;
}

void Matrix::set_value_at(unsigned int index, double value)
{
    this->D[index] = value;
}

void Matrix::set_value_at(unsigned int row, unsigned int col, double value)
{
    this->D[row * col + col] = value;
}
```

```
double Matrix::get_value_at(unsigned int row, unsigned int col)
{
    return this->D[row * col + col];
}



Matrix Matrix::multiply_by(Matrix& B)
{
    if(this->c != B.r)
    {
        std::stringstream msg;
        msg << "Cannot multiply a " << this->r << "x" << this->c
            << " matrix and " << B.r << "x" << B.c << " matrix together";
        throw std::invalid_argument( msg.str() );
    }

    // let m represent the matching col and row in A and B.
    unsigned int m = this->c;

    // New matrix
    Matrix C(this->r, B.c, 0.0);
    unsigned int i = 0;
    unsigned int len = C.c * C.r;
    unsigned int row = 0, col = 0;
    // Recall that the multiplication of matricies involves calculated the dot
    // product of the repsective row and column for a cell.
    // TODO: Perform loop unrolling to take advantage of SSE and CPU based
    // parallization.
    do
    {
        row = i / C.c;
        col = i % C.c;

        double dot = 0.0;
        for(unsigned j = 0; j < m; j++)
        {
            dot += this->D[row * this->c + j] * B.D[B.c * j + col];
        }
        C.D[i] = dot;
    }
    while( ++i < len );
```

```
    return C;
}

Matrix::~Matrix() // Destructor
{
//    delete[] this->D;
}

} // END project NAMESPACE
```

Matrix.hpp

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>

#ifndef MATRIX_HPP
#define MATRIX_HPP

namespace project
{
    class Matrix;

    //!
    //! \brief Interface that students will need to implement.
    //!
    //! Allows testing of their code for their convenience.
    class IMatrix
    {
        public:
            IMatrix(unsigned int rows, unsigned int cols, double init_val) { };

            virtual ~IMatrix() { };

            //!
            //! Dump the contents of a matrix to the screen.
            //!
            static std::string dump_to_string(IMatrix & A, char delimiter) {
                std::ostringstream ostr;
```

```
    for(unsigned int r = 0; r < A.get_row_count(); r++) {
        for(unsigned int c = 0; c < A.get_col_count(); c++) {
            ostr << A.get_value_at(r, c) << delimiter;
        }
    }

    return ostr.str();
}

//!
//! Get the row count of the matrix.
//!
//! \return rows
virtual unsigned int get_row_count() = 0;

//!
//! Get the col count of the matrix.
//!
//! \return cols
virtual unsigned int get_col_count() = 0;

//!
//! Set the value at an index. You need to implement this method if
//! you want the test script to be able to initialize your matrix.
//!
virtual void set_value_at(unsigned int index, double value) = 0;

//!
//! Set the value at a specific row and column in the matrix.
//!
virtual void set_value_at(unsigned int row, unsigned int col, double value) = 0;

//!
//! Get a the value at row a and column b in the matrix.
//!
//! \return the value at the index.
virtual double get_value_at(unsigned int row, unsigned int col) = 0;

//!
//! Multiplies the matrix by another matrix using a CUDA enabled
//! GPU.
```

```
        //!
        //! \return the result as a new matrix.
        virtual Matrix multiply_by(Matrix& B) = 0;

        //!
        //! Multiplies the matrix by another matrix.
        //! NOTE: It is best to implement this method in the kernel.cu file
        //! so that nvcc compilation can be seperated from gcc compilation.
        //!
        //! \return the result as a new Matrix.
        virtual Matrix cuda_multiply_by(Matrix& B, float &cuda_ms) = 0;
};

//! \brief class to represent a matrix.
//!
//! The matrix class implements the IMatrix interface. Allows students to
//! get familiar with C++ inheritance and how to emulate an interface via
//! abstract classes.
//!
//! NOTE: You must implement this class in Matrix.cpp
//!
//! C++ structs default to public for all internal properties and methods.
class Matrix : public IMatrix
{
    private:
        std::vector<double> D; ///< Use a vector to avoid memory management.
        unsigned int r; ///< The number of rows in the matrix.
        unsigned int c; ///< The number of cols in the matrix.

    public:
        //!
        //! Initializes a new matrix object.
        //!
        //! \param the number of rows.
        //! \param the number of columns.
        //! \param the initialization data.
        //!
        Matrix(unsigned int rows, unsigned int cols, double init_val);
        unsigned int get_row_count();
        unsigned int get_col_count();
        void set_value_at(unsigned int index, double value);
```

```
            void set_value_at(unsigned int row, unsigned int col, double value);
            double get_value_at(unsigned int row, unsigned int col);
            Matrix multiply_by(Matrix& B);
            Matrix cuda_multiply_by(Matrix& B, float &cuda_ms); ///< Located in kernel.cu
            ~Matrix(); ///> Destructor
    };
}



#endif
```

README.md

```
# CUDA Matrix Multiplication


## Description
This assignment will require that you implement a classic matrix multiplication
function. Afterwards you will be required to implement the same function using
the CUDA API and run it on the GPU.


## Directions
  0. Documentation
    1. Crate a blank document that contains your full name and the assignment
    name (CUDA Matrix Multiplication)
  1. Getting Started
    1. First run the command `make`. That will compile the program (there
    should be no errors). Take a screenshot of the results.
    2. Next run the executable `./matrixMultiply -h` that was created. It will print out the usage
 information.
    3. Next run the command `make testgpu` to test the binary using `tests/run_test.pl`. While the
 test is running it will output results back to the screen.
    4. Take a screenshot of the previous two steps and past it into your document.
  2. There are three files you will need to modify in order to complete the
  project.
    1. Write the implementation details for your Matrix class in `Matrix.cpp`.
    2. Write the CUDA kernel implementation in `kernels.cu`. If you decide to
change what arguments the kernel function takes make sure to update
`kernels.h` to reflect your changes.
    3. Unlike the other Matrix class methods,
`Matrix cuda_multiply_by(Matrix& B, float &cuda_ms);` needs to be located
```

by the kernels.cu file. This is due to the fact that calling a kernel
function requires using a special syntax that only the ‘nvcc‘ compiler
understands.

3. Once you have implemented the necessary code and all of the tests pass. Run ‘make clean && make‘ and then run ‘make testgpu‘. Take a screenshot of the output and paste it into your document.
4. You will also need to make sure that the CPU matrix multiplication algorithm works. Update the code and run ‘make testcpu‘ until there are no more errors.
5. If your are experiencing issues with the program and would like to debug it with ‘cuda-gdb‘ then run ‘make clean && make debug‘.

Extra Notes
C++ Allows class methods to be defined in separate files. For example, in
this project the ‘Matrix‘ class method ‘cuda_multiply_by‘ is in ‘kernels.cu‘
instead of ‘Matrix.cpp‘. The reason is that it makes use of the CUDA
extensions which requires that it gets compiled by ‘nvcc‘. Even though
‘nvcc‘ can compile all of the source files in this directory. It is a lot
more clean to separate kernel functions into their own file. This then allows
you to use other compilers for the rest of your code and then link the
objects together without any ill effects.

Learning Goals
After completing this project will have learned how to properly perform matrix
multiplication on the CPU and on the GPU. In conjunction, you will have learned
how to allocate and deallocate data onto the GPU and implement a kernel
function that takes advantage of the GPU and the allocated data.

Grading
The project is worth a maximum of 100 points.

- Submit the document that contains the screenshots. +20
- The Matrix constructor has been properly implemented. +10
- ‘get_rows‘ has been properly implemented. +5
- ‘get_cols‘ has been properly implemented. +5
- ‘get_value_at‘ has been properly implemented. +5
- ‘multiply_by‘ has been properly implemented. +20
- ‘cuda_multiply_by‘ has been properly implemented. +17
- ‘KernelMatrixMultiply‘ has been properly implemented. +17
- _Extra Credit_ Use the CUDA API to time how long your kernel function
takes. Log this to a file. (‘run_tests.pl‘ will get confused and report errors). +5

run_tests.pl

```
#!/usr/bin/env perl

use strict;
use warnings;
use Getopt::Long;
use Data::Dumper;
use Pod::Usage;
use Cwd;

my $numArgs = @ARGV;
my $help = 0;
my $bin_path = undef;
my $matrice_path = undef;
my $golden_path = undef;
my $delimiter = undef;
my $cpu = 0;

GetOptions(
    'help|?|h' => \$help,
    'b|bin-path=s' => \$bin_path,
    'm|matrices-path=s' => \$matrice_path,
    'g|golden-path=s' => \$golden_path,
    'd|delimiter=s' => \$delimiter,
    'c|cpu' => \$cpu
    ) or die "Error parsing arguments\n";

if($help || $numArgs == 0) {
    pod2usage(1);
} else {
    if($delimiter) {
        $delimiter = substr $delimiter, 0, 1;
    } else {
        $delimiter = ' ';
    }
    $/ = $delimiter;

    opendir my $matrice_dir, $matrice_path or die "Error: $!\n";
    while(my $file = readdir($matrice_dir)) {
```

```
        if(-f "$matrice_path/$file" && ($file =~ /([0-9]+)x([0-9]+)-([0-9]+)x([0-9]+).mat/))
        {
            print "------------------[ $file ]----------------------\n";
            my $golden_file = "$1x$2-$3x$4-$1x$4.mat";
            my $result = "";
            if($cpu) {
                print "\tExec: cat $matrice_path/$file | $bin_path -a $1x$2 -b $3x$4 -d
\"$delimiter\" --\n";

                $result = `cat $matrice_path/$file | $bin_path -a $1x$2 -b $3x$4 -d "$delimiter"
--`;
            } else {
                print "\tExec: cat $matrice_path/$file | $bin_path -a $1x$2 -b $3x$4 -d
\"$delimiter\" -g --\n";

                $result = `cat $matrice_path/$file | $bin_path -a $1x$2 -b $3x$4 -d "$delimiter" -g
--`;
            }

            print "\tGolden File: $golden_path/$golden_file\n";
            my $c_actual = string_for_vector($result);
            my $golden_result = `cat $golden_path/$golden_file`;
            my $c_golden = string_for_vector($golden_result);

            if(@$c_actual ne @$c_golden) {
                print "\tThe program's output size differed from the expected size in
$golden_path/$golden_file\n";
                print "\t   Actual: ". scalar @$c_actual ."\n";
                print "\t   Golden: ". scalar @$c_golden ."\n";
            } else {
                my $miss_dims = compare_matrices($c_actual, $c_golden, $1, $4);
                if($miss_dims) {
                    print "\tThere was a missmatch between the golden (expected) matrix and actual
matrix at index:\n";
                    print "\t   row:    $miss_dims->[0]\n";
                    print "\t   column: $miss_dims->[1]\n";
                    print "\tThe error occurred when using $matrice_path/$file as the input matrices
and $golden_path/$golden_file as the expected matrix\n";
                }
            }
        }
```

```
    }
}

#
# Compare two matrices
#
# \param actual matrix
# \param expected (golden) matrix
# \param number of rows.
# \param number of colums.
#
# \return miss index (x, y)
#
sub compare_matrices {
    my $A = shift;
    my $B = shift;
    my @dims = @_;

    for(my $r = 0; $r < $dims[0]; $r++) {
        for(my $c = 0; $c < $dims[1]; $c++) {
            if($A->[$r * $c + $c] ne $A->[$r * $c + $c]) {
                return [$r, $c];
            }
        }
    }

    return undef;
}



#
# Takes a string and parses it into an array of numeric values.
# assumes that $/ is the delimiter.
#
# \param string to parse.
#
# \return the array of values.
sub string_for_vector {
    my $string = shift;

    my $values = [map {$_} split $/, trim($string)];
```

```
    return $values;
}

# Removes blank spaces from edges of a string.
sub trim {
    my $totrim = shift;

    $totrim =~ s/^\s+//g;
    $totrim =~ s/\s+$//g;

    return $totrim;
}

#
# Read a file as a vector of values. Assumes that $/ is the delimiter.
#
# \param path to the file
#
# \return a list or undef.
sub file_for_vector {
    my $filepath = shift;

    open( my $file, '<', $filepath ) or warn "Failed to open $filepath\n";

    my @vector;
    while( my $num = <$file> ) {
        chomp $num;
        if($num =~ /[0-9]+[.]?[0-9]*/) {
            push @vector, $num * 1.0;
        }
    }

    close $file;

    return \@vector;
}

__END__

=head1 NAME
```

```
    Runs a series of tests on the specified matrix muliplication bin.

=head1 SYNOPSIS

    ./run_tests.pl

EXAMPLE:

    ./run_tests.pl -b ../matrixMultiply -m ./matrices -g ./golden -d \

=head1 OPTIONS

=item -h --help

    Print usage information to the screen.

=item -b --bin-path

    Path to the binary to execute.

=item -m --matrices-path

    Path to the directory that contains the matrices to multiply.

=item -g --golden-path

    Path to the directory that contains the expected results.

=item -d --delimiter

    The delimiter used to seperate values.


=item -c --cpu

    Test using the CPU.

=back

=cut
```

9.2.2 Radix Sort

cuda_radix.cu

```
#include <vector>
#include<cuda.h>
#include<cuda_runtime.h>

#include"utils.h"
#include"cuda_radix.h"
#include"cuda_scan.h"
#include"RadixSort.hpp"

namespace project {

//! Remember the prototype to this function is in RadixSort.hpp
//! NOTE: For simplicity make sure that to_sort.size() is always
//!     a multiple of BLOCK_SIZE in cuda_radix.h.
std::vector<int> & RadixSort::parallel_radix_sort() {
    int gDmX = (int) this->to_sort.size() / BLOCK_SIZE;
    int bDmX = BLOCK_SIZE;

    assert(this->to_sort.size() == this->sorted.size());
    assert((int)this->to_sort.size() >= BLOCK_SIZE);
    assert((int)this->to_sort.size() % BLOCK_SIZE == 0);

    // For timing
    cudaEvent_t start, stop;
    float cuda_elapsed_time_ms = 0.0;

    // Vectors for storing the totals and scanned totals.
    std::vector<int> block_totals(gDmX * 2, 0);
    std::vector<int> block_totals_scanned(gDmX * 2, 0);

    int * d_to_sort = 0, * d_sorted = 0,
        * d_block_totals = 0, * d_block_totals_scanned = 0;

    unsigned int d_to_sort_mem_size = sizeof(int) * this->to_sort.size(),
                 d_sorted_mem_size  = d_to_sort_mem_size,
                 d_block_totals_mem_size = sizeof(int) * gDmX * 2,
```

```
             d_block_totals_scanned_mem_size = d_block_totals_mem_size;

// Initialize and begin the timers
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));
checkCudaErrors(cudaEventRecord(start));

// Allocate the memory on the device
checkCudaErrors(cudaMalloc((void **)&d_to_sort, d_to_sort_mem_size));
checkCudaErrors(cudaMalloc((void **)&d_sorted, d_sorted_mem_size));
checkCudaErrors(cudaMalloc((void **)&d_block_totals,
            d_block_totals_mem_size));
checkCudaErrors(cudaMalloc((void **)&d_block_totals_scanned,
            d_block_totals_mem_size));

// Copy from the host to the device.
checkCudaErrors(cudaMemcpy(d_to_sort, &this->to_sort[0],
            d_to_sort_mem_size, cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_sorted, &this->to_sort[0],
            d_sorted_mem_size,  cudaMemcpyHostToDevice));

// Calculate the block and grid dimensions.
dim3 gridDims(gDmX, 1);
dim3 blockDims(bDmX, 1);

// Sort at each bit.
for(unsigned int exponent = 0; exponent < sizeof(int) * 8; ++exponent) {

    // First perform a single bit sort.
    kernel_radix_sort_1bit<<<gridDims, blockDims>>>(exponent, d_to_sort,
                                                    d_sorted,
                                                    d_block_totals);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&block_totals[0], d_block_totals,
                               d_block_totals_mem_size,
                               cudaMemcpyDeviceToHost));

    // Now perform a scan on the totals.
```

```
        cuda_parallel_scan(BLOCK_SIZE, block_totals, block_totals_scanned);

        // Copy from the host to the device.
        checkCudaErrors(cudaMemcpy(d_block_totals_scanned,
                                   &block_totals_scanned[0],
                                   d_block_totals_scanned_mem_size,
                                   cudaMemcpyHostToDevice));

        // Now we can map the values.
        int * d_remapped = d_to_sort; //for readability and clarity.
        kernel_remap<<<gridDims, blockDims>>>(d_sorted, d_block_totals,
                                              d_block_totals_scanned,
                                              d_remapped);
        checkCudaErrors(cudaGetLastError());
        checkCudaErrors(cudaThreadSynchronize());
    }

    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&this->sorted[0], d_sorted, d_sorted_mem_size,
                               cudaMemcpyDeviceToHost));

    // Free up the device.
    checkCudaErrors(cudaFree(d_to_sort));
    checkCudaErrors(cudaFree(d_sorted));
    checkCudaErrors(cudaFree(d_block_totals));
    checkCudaErrors(cudaFree(d_block_totals_scanned));

    // Stop the timers
    checkCudaErrors(cudaEventRecord(stop));
    checkCudaErrors(cudaEventSynchronize(stop));
    checkCudaErrors(cudaEventElapsedTime(&cuda_elapsed_time_ms, start, stop));

    return this->sorted;
}

//---------- Kernel Implementations ------------//
//! \brief Radix Sort Kernel
__global__ void kernel_radix_sort_1bit( const int exponent,
                                        const int * d_to_sort,
```

```
                                        int * d_sorted, int * d_block_totals) {
    __shared__ int s_to_sort[BLOCK_SIZE];
    __shared__ int s_predicate[BLOCK_SIZE];
    __shared__ int s_scan[BLOCK_SIZE];
    __shared__ int s_total_falses;

    // Copy the data to L2 cache.
    s_to_sort[threadIdx.x] = d_to_sort[threadIdx.x + blockIdx.x * blockDim.x];
    s_predicate[threadIdx.x] = 0;
    s_scan[threadIdx.x] = 0;

    if(threadIdx.x == 0) {
        s_total_falses = 0;
    }

    s_predicate[threadIdx.x] = !(s_to_sort[threadIdx.x] & (1 << exponent));
    s_scan[threadIdx.x] = s_predicate[threadIdx.x];
    __syncthreads();

    bl_sweep_up(s_scan, BLOCK_SIZE);
    __syncthreads();
    bl_sweep_down(s_scan, BLOCK_SIZE);
    __syncthreads();

    if(threadIdx.x == 0) {
        s_total_falses = s_scan[BLOCK_SIZE - 1] + s_predicate[BLOCK_SIZE - 1];
    }
    __syncthreads();

    // Build the scatter indexes.
    s_scan[threadIdx.x] = s_predicate[threadIdx.x]
        ? s_scan[threadIdx.x]
        : threadIdx.x - s_scan[threadIdx.x] + s_total_falses;
    d_sorted[s_scan[threadIdx.x] + blockIdx.x * blockDim.x] =
        s_to_sort[threadIdx.x];

    // Store the total falses and trues into the d_block_totals array
    // for further processing.
    d_block_totals[blockIdx.x] = s_total_falses;
    d_block_totals[gridDim.x + blockIdx.x] = blockDim.x - s_total_falses;
}
```

```
//! \brief Remap Kernel
__global__ void kernel_remap( const int * d_to_remap,
                              const int * d_block_totals,
                              const int * d_offsets, int * d_remapped) {

    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;
    int tIdx = threadIdx.x;

    if(tIdx < d_block_totals[blockIdx.x]) {
        int mapped_index = d_offsets[blockIdx.x] + tIdx;
        d_remapped[mapped_index] = d_to_remap[gIdx];
    } else {
        int mapped_index = d_offsets[gridDim.x + blockIdx.x] + tIdx
            - d_block_totals[blockIdx.x];
        d_remapped[mapped_index] = d_to_remap[gIdx];
    }
}

}
```

cuda_radix.h

```
#ifndef CUDA_RADIX_H
#define CUDA_RADIX_H

#define BLOCK_SIZE  (1 << 7) //Change the default block size.

namespace project
{
    /*! \brief Radix Sort Kernel Prototype
     *
     * Students will need to implement this function in cuda_radix.cu. Takes
     * in a pointer to the global
     * unsorted array of integers sorts them at the desired exponent and then
     * stores the results into the
     * global array d_sorted.
     *
     * @param[in] the bit position to sort at.
     * @param[in] the vector to sort.
     * @param[in, out] the sorted results.
     * @param[in, out] store the count of bits that contained 0. This will be
```

```
     *                 needed inside of the bitremap function.
     *
     !*/
    __global__ void kernel_radix_sort_1bit( const int exponent,
                                            const int * d_to_sort,
                                            int * d_sorted,
                                            int * d_block_totals);


    /*! \brief Remaps integers in a array.
     *
     * Remaps the contents of an array to a new array in global memory. The
     * mappings are dependent on the block totals and the offsets (which is
     * calculated using the scan operation).
     *
     * @param[in] the vector to remap.
     * @param[in] the block totals. Refer to 'kernel_radix_sort' for a brief
     *            description of the block totals.
     * @param[in] the offsets.
     * @param[in, out] the vector to store the mapped values. Determined by
     *                 using the block totals and offset
     *                 together.
     *
     !*/
    __global__ void kernel_remap( const int * d_to_remap,
                                  const int * d_block_totals,
                                  const int * d_offsets, int * d_global_store);
}

#endif
```

cuda_scan.cu

```
#include"cuda_scan.h"
#include"utils.h"

// find the log_2 of a value
inline __device__ int log2(int i) {
    int p = 0;
    while(i >>= 1) p++;
    return p;
}
```

```
/*! \brief Blelloch scan sweep-up operation
 *
 *  Performs the sweep-up substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_up(int * to_sweep, int size) {
    //1: for d = 0 to log2 n  1 do
    //2:      for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //3:           x[t] = x[t] + x[t - t / 2]
    int t = threadIdx.x;
    for( int d =  0; d < log2(size); ++d) {
        __syncthreads(); // wrapping the condition with synthreads prevents a
                         // rare race condition.
        if( t < size && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
            to_sweep[t] = to_sweep[t] + to_sweep[tp];
        }
    }
}

/*! \brief Blelloch scan sweep-down operation
 *
 *  Performs the sweep-down substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_down(int * to_sweep, int size) {
    //1: x[n  1] <- 0
    //2: for d = log2 n  1 down to 0 do
    //3:       for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //4:            carry = x[t]
    //5:            x[t] += x[t - t / 2]
    //6:            x[t - t / 2] = carry
    to_sweep[size - 1] = 0;
    int t = threadIdx.x;
    for( int d = log2(size) - 1; d >= 0; --d) {
        __syncthreads(); // wrapping the condition with synthreads
```

```
                    // prevents a rare race condition.
        if( (t < size) && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
            int tmp = to_sweep[t];
            to_sweep[t] += to_sweep[tp];
            to_sweep[tp] = tmp;
        }
    }
}

/*! \brief Blelloch Scan Kernel
 *
 *  The blelloch scan algorithm kernel. Implementation based on Udacity's and
 *  NVidia's explanations. This will only do a block level scan.
 *
 *  Udacity: https://www.youtube.com/watch?v=_5sM-4ODXaA
 *  Nvidia: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
 *
 *  @param[in] the sequence to scan. Assumes that the sequence is no larger
 *             than the block size.
 *  @param[in, out] the sequence to store the scanned results.
 */
__global__ void kernel_blelloch_scan(const int d_to_scan_size,
                                     const int * d_to_scan, int * d_scanned) {
    int tIdx = threadIdx.x;

    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = 0;

    if(tIdx < d_to_scan_size) {
        //copy d_to_scanto s_to_scan memory to speed up the performance.
        s_to_scan[tIdx] = d_to_scan[tIdx];
    }
    __syncthreads();

    // perform the sweep up phase of the algorithm
    bl_sweep_up(s_to_scan, blockDim.x);
    // perform the sweep down phase of the algorithm
    bl_sweep_down(s_to_scan, blockDim.x);
```

```
    if(tIdx < d_to_scan_size) {
        //copy back to global memory.
        d_scanned[tIdx] = s_to_scan[tIdx];
    }
}

/*! \brief Block scan kernel function.
 *
 *  The kernel_block_scan kernel takes in a sequence of elemetns and performs a
 *  block level scans accross the sequence. The block dimensions need to be of
 *  size 2^x <= 1024 (or cuda specific maximum block size).
 *
 *  @param[in] the sequence of integers to scan.
 *  @param[in, out] the sequence to store the block level scans.
 *  @param[in, out] the sequence to store the reductions of each block.
 *                  AKA it's length is the length of the grid.
 */
__global__ void kernel_block_scan(const int d_to_scan_size,
                                  const int * d_to_scan, int * d_scanned,
                                  int * d_block_reductions) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    // Init the dynamic shared memory.
    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = d_to_scan[d_to_scan_size - 1];

    // Copy from global to local.
    if(gIdx < d_to_scan_size) {
        s_to_scan[tIdx] = d_to_scan[gIdx];
    }
    __syncthreads();

    // Scan the shared block of data.
    bl_sweep_up(s_to_scan, blockDim.x);
    bl_sweep_down(s_to_scan, blockDim.x);

    // Now store the block reduction (sum of all elets)
    if(tIdx == (blockDim.x - 1)) {
        d_block_reductions[blockIdx.x] = s_to_scan[tIdx] + d_to_scan[gIdx];
    }
```

```
    __syncthreads();


    //Push the results back out to global memory.
    if(gIdx < d_to_scan_size) {
        d_scanned[gIdx] = s_to_scan[tIdx];
    }
}

/*! \brief Offset blocks
 *
 *  The kernel_block_offset takes in a sequence of values and performs an the
 *  corresponding block offset from the offset sequence.
 *
 *  @param[in, out] the sequence to offset.
 *  @param[in] the corresponding block offset values.
 */
__global__ void kernel_block_offset(int d_to_offset_size, int * d_to_offset,
                                    int * d_offsets) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    extern __shared__ int s_to_offset[];
    s_to_offset[tIdx] = 0;

    // Copy from global to shared.
    if(gIdx < d_to_offset_size) {
        s_to_offset[tIdx] = d_to_offset[gIdx];
    }
    __shared__ int offset;
    offset = d_offsets[blockIdx.x];
    __syncthreads();

    s_to_offset[tIdx] += offset;
    __syncthreads();

    // Copy back to global memory.
    if(gIdx < d_to_offset_size) {
        d_to_offset[gIdx] = s_to_offset[tIdx];
    }
}
```

```
/*! \brief CUDA Parallel Scan
 *
 * See cuda_parallel_scan(int, std::vector<int>, std::vector<int>,
 *                        const bool debug)
 *
 */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                         std::vector<int> & scanned) {
    return cuda_parallel_scan(block_size, to_scan, scanned, false);
}

/*! \brief CUDA Parallel Scan
 *
 *  The cuda_parallel_scan is a recursive function that takes in a large set of
 *  values and applies the scan operator across the entire set.
 *
 *  @param[in] The size of the block to use.
 *  @param[in] the vector of elements to scan.
 *             The length must **always** be a multiple of block_size and
 *             to_scan.size() / block_size must never surpass 2147483648
 *             on CUDA 3.0 devices.
 *  @param[in, out] a vector to store the scan results.
 *
 *  @returns the runtime of all the function calls involving the device.
 */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                         std::vector<int> & scanned, const bool debug) {
    // For timing
    cudaEvent_t start, stop;
    float cuda_elapsed_time_ms = 0.0;

    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_scan.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_parallel_scan\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX,
               to_scan.size(), block_size);
```

```
}

// Vector initializations
std::vector<int> block_reductions(gDmX, 0);
std::vector<int> scanned_block_reductions(block_reductions.size(), 0);

// create our pointers to the device memory (d prefix)
int * d_to_scan = 0,
    * d_scanned = 0,
    * d_block_reductions = 0;

// memory allocation sizes
unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
             scanned_mem_size = sizeof(int) * scanned.size(),
             block_reductions_mem_size = sizeof(int)
                 * block_reductions.size();

// Initialize and begin the timers
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));
checkCudaErrors(cudaEventRecord(start));

// Allocate the memory on the device
checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));
checkCudaErrors(cudaMalloc((void **)&d_block_reductions,
                           block_reductions_mem_size));

// Copy from the host to the device.
checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size,
                           cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_block_reductions, &block_reductions[0],
                           block_reductions_mem_size,
                           cudaMemcpyHostToDevice));

// Calculate the shared memory size.
unsigned int shared_mem_size = sizeof(int) * blockDims.x;

// Run a block level scan.
kernel_block_scan<<<gridDims, blockDims, shared_mem_size>>>(to_scan.size(),
        d_to_scan, d_scanned, d_block_reductions);
```

```
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size,
                               cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(&block_reductions[0], d_block_reductions,
                               block_reductions_mem_size,
                               cudaMemcpyDeviceToHost));

    // Free GPU resources before recursing.
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
    checkCudaErrors(cudaFree(d_block_reductions));

    // If the size of block_reductions is larger than a single block then
    //  recurse and call cuda_parallel_scan on the block_reductions vector.
    // Else, pass the block_reductions vector to kernel_blelloch_scan.
    if(block_reductions.size() > (unsigned int) block_size) {
        cuda_parallel_scan(block_size, block_reductions,
                scanned_block_reductions, debug);
    } else {
        cuda_blelloch_scan(block_size, block_reductions,
                scanned_block_reductions, debug);
    }

    // Now perform the offsets to get a global scan.
    cuda_block_offset(block_size, scanned, scanned_block_reductions, debug);

    // Stop the timers
    checkCudaErrors(cudaEventRecord(stop));
    checkCudaErrors(cudaEventSynchronize(stop));
    checkCudaErrors(cudaEventElapsedTime(&cuda_elapsed_time_ms, start, stop));

    return cuda_elapsed_time_ms;
}

/*! \brief CUDA Blelloch scan.
 *
 * The cuda-blelloch-scan method will only perfrom a single block scan of a
 * sequence using cuda. For devices with a compute capability of 3.0 then the
```

```
 * maximum block size is 1024.
 *
 *  @param[in] the vector of elements to scan. The size of the sequence must
 *             not surpass the maximum supported block size on the device.
 *  @param[in, out] a vector to store the scan results.
 */
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned, const bool debug) {
    int * d_to_scan = 0,
        * d_scanned = 0;

    unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
                 scanned_mem_size = to_scan_mem_size;

    if(debug) {
        printf("---> Entering: cuda_blelloch_scan\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", 1,
                to_scan.size(), block_size);
    }

    // Allocate the memory on the devicee
    checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));

    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size,
                cudaMemcpyHostToDevice));

    // Calculate the shared memory size
    unsigned int shared_mem_size = sizeof(int) * block_size;

    // Calculate the block and grid dimensions.
    dim3 gridDims(1, 1);
    dim3 blockDims(block_size, 1);

    // Run the blelloch scan on the block_reductions array.
    kernel_blelloch_scan<<<gridDims, blockDims, shared_mem_size>>>(
            to_scan.size(), d_to_scan, d_scanned);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());
```

```
    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size,
                cudaMemcpyDeviceToHost));

    // Clean up
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
}

/*! \brief CUDA Parallel Offset
 *
 * Takes in a sequence and offsets the values per block. The offsets sequence
 * contains the per block offsets.
 *
 *  @param[in] the size of the block.
 *  @param[in, out] the sequence of elements to perform block level offsets.
 *  @param[in] the offset values.
 */
void cuda_block_offset(int block_size, std::vector<int> & to_offset,
        std::vector<int> offsets, const bool debug) {
    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_offset.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_block_offset\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX,
               to_offset.size(), blockDims.x);
    }

    int * d_to_offset = 0,
        * d_offsets = 0;

    unsigned int to_offset_mem_size = sizeof(int) * to_offset.size(),
                 offsets_mem_size = sizeof(int) * offsets.size();

    // Allocate the memory
    checkCudaErrors(cudaMalloc((void **)&d_to_offset, to_offset_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_offsets, offsets_mem_size));
```

```
    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_offset, &to_offset[0], to_offset_mem_size,
                               cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_offsets, &offsets[0], offsets_mem_size,
                               cudaMemcpyHostToDevice));

    // Calculate the shared memory size.
    unsigned int shared_mem_size = sizeof(int) * blockDims.x;

    // Execute the kernel
    kernel_block_offset<<<gridDims, blockDims, shared_mem_size>>>(
            to_offset.size(), d_to_offset, d_offsets);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&to_offset[0], d_to_offset, to_offset_mem_size,
                               cudaMemcpyDeviceToHost));

    // Free unused memory
    checkCudaErrors(cudaFree(d_to_offset));
    checkCudaErrors(cudaFree(d_offsets));
}

/*! \brief Sequential Sum Scan
 *
 *  @param[in] the vector to scan.
 *  @param[in, out] the vector to store the scanned results.
 *  @param[in] perform an inclusive or exclusive scan.
 */
void sequential_sum_scan(std::vector<int> & in, std::vector<int> & out,
        bool inclusive) {
    assert(in.size() == out.size());

    int sum = 0;
    for( std::vector<int>::iterator in_it = in.begin(), out_it = out.begin();
            (in_it != in.end()) && (out_it != in.end());
            ++in_it, ++out_it) {

        if( inclusive ) {
            sum += *in_it; //include the first element of in.
```

```
            *out_it = sum;
        } else {
            *out_it = sum; //do not include the first element of in.
            sum += *in_it;
        }
    }
}
```

cuda_scan.h

```
#ifndef SCAN_H
#define SCAN_H

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<cuda_runtime.h>
#include<assert.h>
#include<vector>

/* Kernel Function Prototypes */
__device__ int log2(int i);
__device__ void bl_sweep_up(int * to_sweep, int size);
__device__ void bl_sweep_down(int * to_sweep, int size);
__global__ void kernel_blelloch_scan(int * d_to_scan, int * d_scanned);
__global__ void kernel_block_scan(int * d_to_scan, int * d_scanned, int * d_block_reductions);
__global__ void kernel_block_offset(int * d_to_offset, int * d_offsets);

/* CUDA Function Prototypes */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned);
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                         std::vector<int> & scanned, const bool debug);
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned, const bool debug);
void cuda_block_offset(int block_size, std::vector<int> & to_offset,
                       std::vector<int> offsets, const bool debug);

/* Serial Functions */
void sequential_sum_scan(std::vector<int> & in, std::vector<int> & out, bool inclusive);
```

```
#endif
```

main.cpp

```
#include<iostream>
#include<sstream>
#include<string>
#include<vector>

#include <sysexits.h> // Program termination codes for linux!

#include"utils.h"
#include"RadixSort.hpp"

/// Function Prototypes
void cin2vectors(std::vector< int > & v0, std::vector< int > & v1,
                 const char del, const int vector_size);

/*! \brief Program entry point
 *
 * Main entry point to the program. To run the application first compile
 * and then run the following command:
 *      cat filewithseq.txt | ./radix_sort -
 *
 * This passes in a sequence from a file to ./radix_sort and the em-dash at the
 * end tells the program to accept from standard input.
 *
 * \param[in] argc the number of arguments
 * \param[in] argv the arguments.
 *
 */
int main(int argc, char *argv[]) {
    std::vector< int > to_sort; // the vector to sort.
    std::vector< int > golden;  // The expected results.

    char del = '\n';
    int vector_size = -1;

    if(argc < 4) {
        std::cout << "Not enough arguments." << std::endl;
        std::cout << "Key: [optional] <required>" << std::endl;
```

```
        std::cout << "usage: \n\tcat shuffled.vec | "
                  << "./radix_sort <delimiter> <vector size> - " << std::endl;
        std::cout << "usage: \n\tcat shuffled.vec [golden.vec] | "
                  << "./radix_sort <delimiter> <vector size> - " << std::endl;

        return EX_IOERR;
    } else {

        if(*argv[3] == '-') {
            del = *argv[1];
            vector_size = std::stoi(std::string(argv[2]));
            cin2vectors(to_sort, golden, del, vector_size);

            project::RadixSort sorter(to_sort);
            sorter.parallel_radix_sort();
            sorter.sequential_radix_sort();

            if(golden.size() == to_sort.size())
            {
                // For personal use when building the program.
                // Just make sure not to output anything else
                // except the sorted vector when submitting the
                // assignment.
            }

            print_vector(sorter.get_sorted(), del);
        } else {
            return EX_IOERR;
        }
    }
}

/*! \brief cin to an int vector
 *
 * Reads from cin and constructs two vectors given the specified
 * length.
 *
 * \param[in, out] The vector to fill.
*/
void cin2vectors(std::vector< int > & v0, std::vector< int > & v1,
                 const char del, const int vector_size) {
```

```
    int i = 0;
    for(std::string line; std::getline(std::cin, line);) {
        std::stringstream ss(line);
        std::string number;
        while(std::getline(ss, number, del)) {
            if( i < vector_size)
            {
                v0.push_back(std::stoi(number));
            } else if(i < vector_size * 2) {
                v1.push_back(std::stoi(number));
            }
            i++;
        }
    }
}
```

RadixSort.cpp

```
#include<vector>

#include"RadixSort.hpp"

namespace project
{

/*!
 * Implemented such that it is easy to compare the differences between the
 * CUDA radix sort and serial radix sort.
 !*/
std::vector<int> & RadixSort::sequential_radix_sort() {
    this->sorted = this->to_sort;

    std::vector<int> tmp(this->to_sort.size(), 0);
    for(unsigned int exponent = 0; exponent < sizeof(int) * 8; ++exponent) {
        int i_n = 0;
        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(!(this->sorted[i] & (1 << exponent))) {
```

```
                tmp[i_n] = this->sorted[i];
                ++i_n;
            }
        }

        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(this->sorted[i] & (1 << exponent)) {
                tmp[i_n] = this->sorted[i];
                ++i_n;
            }
        }

        this->sorted = tmp;
    }

    return this->sorted;
} // END sequential_radix_sort



///< RadixSort::parallel_radix_sort needs to be implemented in cuda_radix.cu

}
```

RadixSort.hpp

```
#ifndef RADIXSORT_H
#define RADIXSORT_H

#include<vector>

namespace project
{
    /*! \brief Radix Sort Interface / Abstract Class
     *
     * Allows the testing of their code for their
     * convenience.
     !*/
    class IRadixSort
    {
```

```
public:
    /*! \brief Default constructor.
     *
     * Copies the contents of the to_sort vector into
     * the class's internal vector.
     *
     * @param[in] the vector to sort. Copied into the class.
     !*/
    IRadixSort(const std::vector<int> &to_sort) {
        this->to_sort = to_sort;
        this->sorted = std::vector<int>(to_sort.size(), 0);
    };
    virtual ~IRadixSort() { }; ///< Default Destructor

    /*! \brief Sequential Radix Sort
     *
     * Perform the classic sequential radix sort on the vector.
     *
     * \returns the sorted vector reference.
     !*/
    virtual std::vector<int> & sequential_radix_sort() = 0;

    /*! \brief Parallel Radix Sort
     *
     * Performs paralle radix sort on the vector. Calls all
     * of the necessary CUDA memory operations and kernels
     * that will sort the data.
     *
     * \returns the sorted vector reference.
     !*/
    virtual std::vector<int> & parallel_radix_sort() = 0;

    /*! \brief Get the sorted vector.
     *
     * \returns the sorted vector.
     !*/
    virtual std::vector<int> & get_sorted()
    {
        return this->sorted;
    };
```

```
        protected:
            std::vector<int> to_sort; ///< This doesn't change
            std::vector<int> sorted;  ///< Storage container for the sorted contents.
    };

    /*! \brief Radix Sorting Class
     *
     * Class that contains all of the logic to sort a sequence using
     * the classic sequential radix sort and a modern parallel radix
     * sorting algorithm.
     *
     * The functions for this class need to be implemented in RadixSort.cpp
     * unless another location is specified.
     !*/
    class RadixSort: public IRadixSort
    {
        public:
            RadixSort(const std::vector<int> &to_sort) : IRadixSort(to_sort) { }
            ~RadixSort() { };

            std::vector<int> & sequential_radix_sort();
            std::vector<int> & parallel_radix_sort(); ///< Implement this function inside of
 'cuda_radix.cu'
    };
}

#endif
```

utils.h

```
/// The original source for this file is available from Udacity's github
/// repository for their "Intro to Parallel Computing" course.
/// url:
 https://raw.githubusercontent.com/udacity/cs344/master/Problem%20Sets/Problem%20Set%201/utils.h
/// Use this when programming!

#ifndef UTILS_H
#define UTILS_H

#include<iostream>
#include<iomanip>
```

```
#include<cstdio>
#include<cuda.h>
#include<cuda_runtime.h>
#include<cuda_runtime_api.h>
#include<cassert>
#include<cmath>

#define checkCudaErrors(val) check( (val), #val, __FILE__, __LINE__)

// Pass all cuda api calls to this fuction. That will help you find mistakes
// quicker.
template<typename T>
void check(T err, const char* const func, const char* const file,
           const int line) {
  if (err != cudaSuccess) {
    std::cerr << "CUDA error at: " << file << ":" << line << std::endl;
    std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
    exit(1);
  }
}

inline void swap_pointers(int ** p0, int ** p1) {
    int * temp = *p0;
    *p0 = *p1;
    *p1 = temp;
}

// Fills a vector with random numbers.
inline void random_fill(std::vector<int> & to_fill) {
    for( std::vector<int>::iterator it = to_fill.begin();
            it != to_fill.end(); ++it) {
        *it = (int)((float)rand() / RAND_MAX);
    }
}

// Fills a vector of size t from 0 ... t - 1.
inline void incremental_fill(std::vector<int> & to_fill) {
    int i = 0;
    for( std::vector<int>::iterator it = to_fill.begin();
            it != to_fill.end(); ++it) {
        *it = i;
```

```
        i++;
    }
}

// Dump the contents of a vector.
inline void print_vector(const std::vector<int> & to_print, const char del) {
    for(std::vector<int>::const_iterator it = to_print.begin();
            it != to_print.end(); ++it) {
        printf("%i%c", *it, del);
    }
    printf("\n");
}

// Performs a fisher yates shuffle on the array.
inline void shuffle(std::vector<int> & to_shuffle) {
    for( unsigned int i = 0; i < to_shuffle.size(); ++i) {
        unsigned int j = (unsigned int)((float) rand() / RAND_MAX * (i + 1));

        int tmp = to_shuffle[j];
        to_shuffle[j] = to_shuffle[i];
        to_shuffle[i] = tmp;
    }
}

// Prints the partitioning of a vector if it were to be passed to a CUDA kernel.
inline void print_vector_with_dims(std::vector<int> & to_print, int blockDimX,
                                   int gridDimX) {
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::iterator it = to_print.begin();
            it != to_print.end(); ++it) {
        if(bDx_counter && !(bDx_counter % blockDimX)) {
            printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                printf("****");
                gDx_counter++;
            }
            printf("\n");
        }
        printf("%i ", *it);
```

```
        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Print two vectors side by side.
inline void print_vector_comparison(std::vector<int> & v1,
                                    std::vector<int> & v2,
                                    int start_index, int stop_index,
                                    int point_at, int blockDimX, int gridDimX) {
    int index = 0;
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end()) && (index < stop_index);
         ++it1, ++it2, ++index) {

        if(bDx_counter && !(bDx_counter % blockDimX)) {
            if(index >= start_index) printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                if(index >= start_index) printf("****");
                gDx_counter++;
            }
            if(index >= start_index) printf("\n");
        }

        if(index == point_at) {
            if(index >= start_index) printf("\n|->(%i, %i)<-|\n", *it1, *it2);
        } else {
            if(index >= start_index) printf("(%i, %i) ", *it1, *it2);
        }

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Compare to vectors. If equal return -1 else return index.
// TODO: Make the parameters const
```

```
inline int equal(std::vector<int> & v1, std::vector<int> & v2) {
    int index = 0;
    for( std::vector<int>::iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end());
         ++it1, ++it2, ++index) {

        if(*it1 != *it2) {
            return index;
        }
    }

    return -1;
}



inline void reset_cuda_devs() {
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    while( dev_count --> 0 )
    {
        printf("Resetting device %i", dev_count);
        cudaSetDevice(dev_count);
        cudaDeviceReset();
    }
    printf("\n");
}

#endif
```

README.md

```
# CUDA Radix Sort Project


## Description
This project requires that you implement the radix sort algorithm. The first
implementation will focus on the sequential algorithm while the second will
focus on the parallel algorithm.


## Directions
  0. Documentation
    1. Crate a blank document that contains your full name and the assignment
```

```
    name (CUDA Radix Sort)
  1. Getting Started
    1. First run the command `make debug`. That will compile the program
    (there should be no errors). Take a screenshot of the results.
    2. Next run the executable `./radix_sort` that was created. It should
    display some usage information.
    4. Take a screenshot of the previous steps and past it into your document.
  2. There are three files you will need to modify in order to complete the
  project.
    1. Write the implementation details for your RadixSort class in
    `RadixSort.cpp`.
    2. Write the CUDA kernel implementations in `cuda_radix.cu`. If you decide
    to change what arguments the kernel function takes make sure to update
    `cuda_radix.h` to reflect your changes.
    3. Unlike the other RadixSort class methods,
    `std::vector<int> & parallel_radix_sort()` needs to be located inside the
    `cuda_radix.cu` file. This is due to the fact that calling a kernel
    function requires using a special syntax that only the `nvcc` compiler
    understands.
  3. Once you have implemented the necessary code and all of the tests pass.
  Take a screenshot of the output and paste it into your document.

## Compilation Directions
  1. To build a debugable version of the program run `make clean && make debug`
  2. To build a release version of the program run `make clean && make`

## Running the Program
  1. To run the program pass in a newline delimited vector from a file.
  For example if the vector file is called `256.vec` then run
  `cat 256.vec | ./radix_sort -`.
  2. To test the program run `tests/run_tests.sh`.

## Extra Notes
  C++ Allows class methods to be defined in separate files. For example,
  in this project the `RadixSort` class method `parallel_radix_sort` is in
  `cuda_radix.cu` instead of `RadixSort.cpp`. The reason is that it makes use
  of the CUDA extensions which requires that it gets compiled by `nvcc`. Even
  though `nvcc` can compile all of the source files in this directory. It is a
  lot more clean to separate kernel functions into their own file. This then
  allows you to use other compilers for the rest of your code and then link the
  objects together without any ill effects.
```

```
## Learning Goals
After completing this project. You will have learned how to sort a large
dataset on the GPU and how to deal with the difficulties of writing an
algorithm that is not limited by the block size.

## Grading
The project is worth a maximum of 100 points.

  - Submit the document that contains the screenshots. +20
  - ‘sequential_radix_sort‘ has been properly implemented. +5
  - ‘parallel_radix_sort‘ has been properly implemented. +30
  - ‘kernel_radix_sort_1bit‘ has been properly implemented. +30
  - ‘kernel_remap‘ has been properly implemented. +15
  - _Extra Credit_  Modify your scan function from the homeworks so that it
  can be used in this project. +10
```

run_tests.pl

```perl
#!/usr/bin/env perl
use strict;
use warnings;
use Data::Dumper;
use Cwd;

my $bin_path = shift; #cwd();
my $shuffled_path = shift or die usage();
my $sorted_path = shift or die usage();
my $golden_path = shift or die usage();
$/ = shift or die usage();

mkdir $sorted_path;

opendir(my $shuffled_dir, $shuffled_path) or die "Unable to open $shuffled_path. Did you generate
 any test vectors?\n If not use the vecGen.py python script.\n";

# run radix_sort on the shuffled vectors and compare its output to the expected vectors.
my $error = 0;
while(my $file = readdir($shuffled_dir)) {
```

```
    if(-f "$shuffled_path/$file" && ($file =~ /(\d+)[.]vec$/ )) {
        `cat "$shuffled_path/$file" | $bin_path $/ $1 - > $sorted_path/$1.vec`;
        my $sorted = file4vector("$sorted_path/$1.vec");
        my $golden = file4vector("$golden_path/$1.vec");
        my $miss_index = compare_vectors($sorted, $golden);
        if(!defined($miss_index)) {
            print "sorted/$1.vec & golden/$1.vec are not the same length. Did you complete the
 program or did you pass in the incorrect delimeter?\n";
        } elsif( $miss_index < 0 ) {
            print "sorted/$1.vec & golden/$1.vec match up.\n";
        }
        else {
            $error = 1;
            print "sorted/$1.vec & golden/$1.vec do not match at index $miss_index.\n";
        }
    }
}

exit $error;


# \brief read a file to a vector.
#
# \param path to the file
#
# \return a list or undef.
sub file4vector {
    my $filepath = shift;

    open( my $file, '<', $filepath ) or warn "Failed to open $filepath\n";

    my @vector;
    while( my $num = <$file> ) {
        chomp $num;
        if($num =~ /[0-9]+/) {
            push @vector, $num;
        }
    }

    close $file;
```

```
    return \@vector;
}

# \brief Compares two vectors.
#
# \param vector ref 1
# \param vector ref 2
#
# \param the index at which the first mismatch occurs. Otherwise return nothing.
sub compare_vectors {
    my $refvec1 = shift;
    my $refvec2 = shift;

    if (@$refvec1 != @$refvec2) {
        warn "The vectors are not the same length\n";
        return undef;
    }

    for(my $i = 0; $i < @$refvec1; $i++) {
        if($refvec1->[$i] != $refvec2->[$i]) {
            return $i;
        }
    }

    return -1;
}

# \brief Return program usage.
sub usage {
    return "Usage: ./run_tests <radix sort bin path> <shuffled dir path> <sorted dir path> <golden
 dir path> <delimiter>\n";
}
```

9.2.3 Parallel Logger

client.cpp

```
#include<iostream> ///< Needed for: cout, endl;
#include<chrono>   ///< Needed for: milliseconds
#include<thread>   ///< Needed for: sleep_for
```

```
#include<cstring>  ///< Needed for: strerror, errno

#include<sysexits.h>
#include<signal.h>

#include"server.hpp"
#include"FIFO.hpp"
#include"Connection.hpp"
#include"Log.hpp"

using namespace project;

namespace project {
    //! \class Client
    //! \brief Class for bookkeeping.
    //!
    //!  Wrapper class so that there is only a single global object that will
    //!  get cleaned up when ctr-c (SIGINT) is pressed.
    //!
    class Client {
        Log * logger_ = NULL;
        public:
            //! \brief default destructor.
            ~Client() {
                if(logger_ != NULL) delete logger_;
            }
            ///! \brief Run the client.
            void run() {
                pid_t pid = getpid();
                //std::cout << "Starting the client " << std::to_string(pid) << std::endl;

                logger_ = new Log();

                //int logcount = 0;
                while(1)
                {
                    std::string message = "Log message from process " +
                        std::to_string(pid) + ".";
                    logger_->write(message);
                    std::chrono::milliseconds ms(125);
                    std::this_thread::sleep_for(ms);
```

```
            }
        };
    };
}

//! \brief catch POSIX system signals.
void catch_function(int signo) {
    //std::cout << "Terminating the client." << std::endl;
    std::exit(EX_OK);
}

project::Client client;
int main(int argc, char * argv[])
{
    signal(SIGINT, catch_function); // catch ctr-c

    client.run();

    return EX_OK;
}
```

Connection.cpp

```
#include<thread>
#include<iostream>

#include "Connection.hpp"
#include "FIFO.hpp"
#include"server.hpp"

namespace project {

////////////////////////////// Connection Methods //////////////////////////////

Connection::Connection()
{
    // FIFO for requests.
    commPathnameFIFO_ = new FIFO<CommunicationFIFOPathnameMessage>(
        connectionInfo_.get_communication_fifo, BOTH_MODE);

    // FIFO for file names.
    requestFIFO_ = new FIFO<RequestFIFOMessage>(
```

```
        connectionInfo_.request_connection_fifo, BOTH_MODE);
}

//! Free up memory.
Connection::~Connection()
{
    //std::cout << "Shutting down the connections" << std::endl;

    delete commPathnameFIFO_;
    delete requestFIFO_;
}

//! Send a connection request to the server.
ssize_t Connection::request_connection() {
    return requestFIFO_->append(&requestComm_);
}

//! Returns a FIFO to represent the connection.
FIFO<Message2Log> * Connection::get_connection()
{
    // Request for a FIFO communication channel.
    request_connection();

    // Wait for the server to send back the name of the communication FIFO.
    commPathnameFIFO_->dequeue(&commPathname_);

    // Initialize a new FIFO object with the returned path. return it.
    FIFO<Message2Log> * message_fifo =
        new FIFO<Message2Log>(commPathname_.pathname, BOTH_MODE);
    return message_fifo;
}

////////////////////////////// ConnectionServer Methods ///////////////////////

//! Blocking operation that waits for a connection.
size_t ConnectionServer::wait_for_connection_request() {
    return requestFIFO_->dequeue(&requestComm_);
}

//! Returns a FIFO to represent the connection.
FIFO<Message2Log> * ConnectionServer::get_connection()
```

```
{
    // Construct the name.
    std::string fifo_name = comm_fifo_prefix_ +
        std::to_string(comm_index_counter_++) + comm_fifo_suffix_;
    strcpy(commPathname_.pathname, fifo_name.c_str());

    // Create the communication fifo
    FIFO<Message2Log> * message_fifo =
        new FIFO<Message2Log>(fifo_name.c_str(), BOTH_MODE);

    // Notify the client of the new communication FIFO.
    commPathnameFIFO_->append(&commPathname_);

    return message_fifo;
}

}
```

Connection.hpp

```
#pragma once
/*!
 * @file Connection.hpp
 * @date 11 Nov 2014
 * @brief Header file of all connection related classes.
 *
 * Connection.hpp contains all of the definitions for
 * the classes required to initiate a connection, communicate through a
 * connection and terminate a connection.
 *
 */
#include<cstring>

//#ifndef CONNECTION_HPP
//#define CONNECTION_HPP

#include"server.hpp"
#include"FIFO.hpp"

namespace project {
    //! \class Connection
```

```
    //! \brief Connection with the server.
    //!
    //! Setup a connection with the server.
    class Connection {
        protected:
            struct ServerConnectionInfo connectionInfo_;
            struct CommunicationFIFOPathnameMessage commPathname_;
            struct RequestFIFOMessage requestComm_;

            FIFO<CommunicationFIFOPathnameMessage> * commPathnameFIFO_;
            FIFO<RequestFIFOMessage> * requestFIFO_;

            ssize_t request_connection();
            void close_connection();
        public:
            Connection();
            virtual ~Connection();
            virtual FIFO<Message2Log> * get_connection();
    };

    //! \class ConnectionServer
    //! \brief setup connections with clients.
    //!
    //! Delegated with the task of setting up connections
    //! with the clients.
    class ConnectionServer : public Connection {
        private:
            int wait_factor_ = 0;
            int comm_index_counter_ = 0;
            std::string comm_fifo_prefix_ = "com";
            std::string comm_fifo_suffix_ = "fifo";

        public:
            ConnectionServer() : Connection() { };
            ~ConnectionServer() { };
            size_t wait_for_connection_request();
            FIFO<Message2Log> * get_connection();
    };
}

//#endif
```

FIFO.hpp

```
#pragma once
/*!
 * @file FIFO.hpp
 * @date 11 Nov 2012
 * @brief Header file that contains the FIFO class.
 *
 * Managing POSIX FIFOs requires keeping track of a lot of components.
 * to solve this issue the FIFO template serves as an abstraction layer over
 * connecting/creating to a fifo and then provides utilities to pass simple
 * data structures over the fifo.
 *
 */
//CPP includes
//Templated classes must contain logic that is contained within the header file.
#include<string>   ///< Needed for: string
#include<cstring>  ///< Needed for: strerror, errno
#include<thread>   ///< Needed for: thread
#include<cstdio>   ///< Needed for: remove
#include<fcntl.h>  ///< needed for: open
#include<unistd.h> ///< Needed for: close
#include<iostream> ///< Needed for: cout, endl;

//linux system includes
#include<sys/stat.h>

//#ifndef FIFO_HPP
//#define FIFO_HPP

#define APPEND_MODE (O_APPEND | O_WRONLY)
#define DEQUEUE_MODE O_RDONLY
#define BOTH_MODE O_RDWR

namespace project
{
    //! \class FIFO
    //! \brief Abstraction layer over linux fifo utilities.
    //!
    //! Used to hide the complexities of setting up a fifo.
    template <class T>
    class FIFO {
```

```
private:
    std::string fifo_pathname_;   ///< The path to the FIFO file.

    std::string mkfifo_errno_;    ///< mkfifo errno store.
    std::string open_fifo_errno_; ///< open fifo errno store.
    std::string append_errno_;    ///< last errno from append.
    std::string dequeue_errno_;    ///< last errno from append.

    std::thread open_thread_;     ///< thread for blocking open operation.

    int fifo_file_descriptor_;    ///< FIFO file descriptor.
    int result_mkfifo_;           ///< Result when calling mkfifo()
    bool opened_ = false;         ///< Was the file opened?
    mode_t mode_;                 ///< The FIFO mode.

    /*! \brief Opens the fifo file.
     *
     *  Since opening a FIFO is a blocking operation then make the
     *  function a class method so that it can be passed off to a
     *  thread.
     *
     *  \param[in, out] a FIFO object to open.
     */
    static void open_fifo(FIFO<T> * fifo, mode_t mode)
    {
        fifo->fifo_file_descriptor_ =
            open(fifo->fifo_pathname_.c_str(), mode);
        if(fifo->fifo_file_descriptor_ >= 0)
        {
            fifo->opened_ = true;
        } else {
            fifo->open_fifo_errno_ = strerror(errno);
        }
    };

public:
    /*! \brief Default constructor
     *
     *  Initializes a FIFO using a pathname.
     *
     *  \param[in] the pathname for the fifo to create and open.
```

```
 */
FIFO<T>(const char * fifo_pathname, mode_t mode) {
    fifo_pathname_ = std::string(fifo_pathname);
    result_mkfifo_ = mkfifo(fifo_pathname, 0777);
    mkfifo_errno_ = strerror(errno);

    open_thread_ = std::thread(open_fifo, this, mode);

    if(result_mkfifo_ < 0) {
        mkfifo_errno_ = strerror(errno);
    }
};

/*! \brief Default destructor.
 *
 *  Delete any used resources.
 *
 */
~FIFO<T>() {
    //std::cout << "Shutting down the FIFO." << std::endl;
    close(fifo_file_descriptor_);

    // Why do you have to check for joinablity when
    // join was already called in ‘append‘ or ‘dequeue‘?
    /* This error will result if there is no attempt to join
     * __again__.

           terminate called without an active exception
           Aborted
    */
    if(open_thread_.joinable()) open_thread_.join();
}

/*! \brief Delete the represented file.
 *
 *  Delete the file that FIFO represents.
 */
int delete_file() {
    return std::remove(fifo_pathname_.c_str());
};
```

```
        /*! \brief Append to the FIFO.
         *
         *  Appends a simple type to the FIFO. Only components within
         *  the type will be appended to the FIFO. For example if T is
         *  a struct then if it contains pointers to the heap then only
         *  the pointers will be appended.
         *
         *  \param[in] the type to append to the fifo.
         */
        ssize_t append(const T * to_append)
        {
            // block until the file is opened.
            if(!opened_) open_thread_.join();
            ssize_t result =
                write(fifo_file_descriptor_, to_append, sizeof(T));
            fsync(fifo_file_descriptor_);
            append_errno_ = strerror(errno);
            return result;
        }

        /*! \brief Dequeue from the FIFO.
         *
         *  Dequeues a simple type from the FIFO. Suffers the same
         *  limitations as
         *
         *  \sa append
         *
         *  \param[in] the type to append to the fifo.
         */
        size_t dequeue(T * container)
        {
            // block until the file is opened.
            if(!opened_) open_thread_.join();
            ssize_t result =
                read(fifo_file_descriptor_, container, sizeof(T));
            fsync(fifo_file_descriptor_);
            dequeue_errno_ = strerror(errno);
            return result;
        }
    };
}
```

```
//#endif
```

Log.cpp

```
#include<cstring>

#include"Log.hpp"

namespace project {

Log::Log() {
    log_fifo_ = connection_.get_connection();
}

Log::~Log() {
    delete log_fifo_;
}

void Log::write(const std::string & message) {
    strcpy(message_container_.message, message.c_str());
    log_fifo_->append(&message_container_);
}

}
```

Log.hpp

```
#pragma once
/*!
 * @file Log.hpp
 * @date 11 Nov 2012
 * @brief Header file that contains the Log class.
 *
 */
#include<string>

#include"Connection.hpp"
#include"FIFO.hpp"
#include"server.hpp"

//#ifndef LOG_HPP
//#define LOG_HPP
```

```
namespace project {
    //! \class Log
    //! \brief Client side logging utility.
    //!
    //! The Log class hides the complexities of setting up a connection and
    //! allows the client to log messages directly.
    class Log {
        private:
            Connection connection_;
            FIFO<Message2Log> * log_fifo_;
            Message2Log message_container_;
        public:
            Log();
            ~Log();
            void write(const std::string & message);
    };
}

//#endif
```

Makefile.inc

```
UNAME_S := $(shell uname -s)

CXX = g++
CXXFLAGS += -pthread -std=c++11
ifeq ($(UNAME_S),Linux)
    CXX = g++-4.8

    # Set the libraries.
    GCC4_8_2_LIBS = -L/opt/gcc/4.8.2/lib64
    GCC4_8_2_INC = -I/opt/gcc/4.8.2/include
    CXXFLAGS += $(GCC4_8_2_LIBS) $(GCC4_8_2_INC)
    CXXFLAGS += -lnsl -Wl,--no-as-needed
endif
ifeq ($(UNAME_S),Darwin)
    CXX = g++-4.9
endif
```

```
CXXFLAGS += -Wall -Wpedantic

#.SILENT:
```

ParallelLogger.cpp

```
#include<string>
#include<fstream>
#include<thread>
#include<iostream>
#include<vector>

#include"server.hpp"
#include"ParallelLogger.hpp"

namespace project {

//! Constructor.
ParallelLogger::ParallelLogger(std::ostream * log_ofstream) {
    out_ = log_ofstream;
}

//! Shutdown components.
ParallelLogger::~ParallelLogger() {
    shutdown_ = true;

    // detach the threads.
    for(std::vector<std::thread>::iterator thrd = log_threads_.begin();
            thrd != log_threads_.end(); thrd++)
    {
        thrd->detach();
    }

    out_->flush();
    delete out_;
}

//! Initialize a thread keep track of the thread and fifo.
void ParallelLogger::spawn_logger_thread(FIFO<Message2Log> * message_fifo) {
    log_threads_.push_back(std::thread(logger_thread, this, message_fifo));
    fifos_.push_back(message_fifo);
}
```

```
//! All logging happens here.
void ParallelLogger::logger_thread(ParallelLogger * logger,
        FIFO<Message2Log> * message_fifo)
{
    struct Message2Log message("");
    while(!logger->shutdown_) {
        message_fifo->dequeue(&message);
        logger->write_log_mutex_.lock();
        *(logger->out_) << logger->log_count_
                        << ": " << std::string(message.message) << "\n";
        ++logger->log_count_;
        logger->write_log_mutex_.unlock();
    }
}

}
```

ParallelLogger.hpp

```
#pragma once
/*!
 * @file ParallelLogger.hpp
 * @date 11 Nov 2012
 * @brief Header file that contains the Log class. Students will be required to
 *  fix this class and ensure there are no raceconditions.
 */

#include<string>
#include<fstream>
#include<thread>
#include<vector>

//#ifndef PARALLELLOGGER_HPP
//#define PARALLELLOGGER_HPP
#include<mutex>
#include<atomic>

#include"FIFO.hpp"
#include"server.hpp"
```

```
namespace project {
    //! \class ParallelLogger
    //! \brief Server side logging utility.
    //!
    //!  The ParallelLogger class takes messages from clients
    //!  and logs their output to a specified file. Client requests
    //!  are handled in parallel.
    class ParallelLogger {
        private:
            std::vector<FIFO<Message2Log>*> fifos_; ///< FIFOs in use.
            std::ostream * out_; ///< The output file stream to log to.
            std::vector<std::thread> log_threads_; ///< logger threads that are running.

            std::mutex write_log_mutex_;

            unsigned int log_count_ = 1; ///< Total log count.

            bool shutdown_ = false; ///< Received the shutdown signal.

            //! \brief Log a message.
            //!
            //! \param[in] the message to log.
            void log(const std::string & message);

            //! \brief Definition of Logger thread.
            //!
            //! Implementation of a logger thread instance that handles
            //! that client requests and logs them to a file.
            //!
            //! \param[in] A ParallelLogger object.
            static void logger_thread(ParallelLogger * logger,
                    FIFO<Message2Log> * message_fifo);

        public:
            //! \brief Default constructor.
            //!
            //! Creates a ParallelLogger that logs to the specified file.
            //! If the file exists then append to it. Otherwise create the
            //! file and append to it.
            //!
```

```
            //! \param[in] an output stream to the target to log to.
            ParallelLogger(std::ostream * log_ofstream);

            //! \brief Default destructor.
            ~ParallelLogger();

            //! \brief Spawn logging thread.
            void spawn_logger_thread(FIFO<Message2Log> * message_fifo);

    };
}

//#endif
```

server.cpp

```
#include<vector>
#include<iostream>
#include<thread>
#include<cstring>  ///< Needed for: strerror, errno
#include<cstdlib>

#include<sysexits.h>
#include<signal.h>

#include"server.hpp"
#include"Connection.hpp"
#include"ParallelLogger.hpp"

#define SLEEP_TIME_MS 250

namespace project {
    //! \class Server
    //! \brief Class for bookkeeping.
    //!
    //!  Wrapper class so that there is only a single global object that will
    //!  get cleaned up when ctr-c (SIGINT) is pressed.
    //!
    class Server {
        private:
            ConnectionServer * connection_server_;
            ParallelLogger * parallel_logger_;
```

```
        public:
            //! \brief Default constructor.
            //!
            //! \param[in] a stream pointing to the desired output location.
            Server(std::ofstream * out_log) {
                connection_server_ = new ConnectionServer();
                parallel_logger_ = new ParallelLogger(out_log);
            };
            //! \brief Default destructor.
            ~Server() {
                delete connection_server_;
                delete parallel_logger_;
            }
            //! \brief run the server.
            void run() {
                while(1) {
                    connection_server_->wait_for_connection_request();
                    FIFO<Message2Log> * message_fifo =
                        connection_server_->get_connection();
                    parallel_logger_->spawn_logger_thread(message_fifo);
                }
            }
    };
}

project::Server * server = NULL; ///< Server object. Pointer prevents blocking.

//! \brief catch POSIX system signals.
void catch_function(int signo) {
    //std::cout << "Terminating the server." << std::endl;

    if(server != NULL) delete server;

    std::exit(EX_OK);
}

int main(int argc, char * argv[])
{
    signal(SIGINT, catch_function); // catch ctr-c
```

```
    std::string log_filepath = "default_log.log";
    if(argc > 1) {
        log_filepath = argv[1];
    }
    server = new project::Server(new std::ofstream(log_filepath.c_str()));
    server->run();

    return EX_OK;
}
```

server.hpp

```
#pragma once

#include<cstring> ///< Needed for: strcpy

//#ifndef SERVER_HPP
//#define SERVER_HPP

//#include"FIFO.hpp"
//#include"Connection.hpp"
//#include"ParallelLogger.hpp"

#define CHAR_LENGTH 256
#define MESSAGE_LENGTH 2048

namespace project {
    //! \brief Contains connection information.
    struct ServerConnectionInfo {
        char request_connection_fifo[CHAR_LENGTH];
        char get_communication_fifo[CHAR_LENGTH];

        ServerConnectionInfo() {
            strcpy(request_connection_fifo, "RequestConnection");
            strcpy( get_communication_fifo, "CommNameSender");
        };
    };

    //! \brief Store a FIFO path name.
    struct CommunicationFIFOPathnameMessage {
```

```
        char pathname[CHAR_LENGTH];
    };

    //! \brief Request a fifo from the server.
    struct RequestFIFOMessage {
        bool request = true;
    };

    //! \brief Send a message to the server.
    struct Message2Log {
        char message[MESSAGE_LENGTH];
        //int date; //UNIX Timestamp.

        Message2Log() : Message2Log("") { };

        Message2Log(const char * a_message) {
            strcpy(message, a_message);
        };
    };
}

//#endif
```

README.md

```
# Parallel Logging Project

## Description
This project is designed to introduce students to the threading utilities provided by C++11 and the
 interprocess communication tools provided by Linux (they follow the POSIX standard).

## Directions
  0. Documentation
    1. Crate a blank document that contains your full name and the assignment
    name (Parallel Logging Project)
  1. Getting Started
    1. First run the command `make clean && make debug`. That will compile the program
    (there should be no errors). Take a screenshot of the results.
    2. Next open two new terminal windows. Both will go into an infinite loop.
      a. Run the command `./server testing.log` in the first window.
```

b. Run the command './client' in the second window.

4. Take a screenshot of the previous steps and past it into your document.

2. There are two files you will need to modify in order to complete the program.

1. 'FIFO.hpp' needs to be modified such that the client and server can setup a named pipe and begin communicating.

2. 'ParallelLogger.cpp' needs to be modified such that it is able to habnd

3. Once you have implemented the necessary code and all of the tests pass.

Take a screenshot of the output and paste it into your document.

Compilation Directions

1. To build a debuggable version of the program run 'make clean && make debug'.

2. To build the program run 'make clean && make'.

Running & Testing the Program

To test the program run 'make test'. Two files will exist in the project directory after running the test. The first is 'testing.log' which contains all of the logs from the program. The second is 'test_results.txt' which tells you where there are inconsistencies in 'testing.log'.

Extra Notes

This project will require joining some threads and detaching other threads for optimal performance. Make sure to run 'make clean' before 'make build' in order for changes in 'FIFO.hpp' to be detected.

Learning Goals

After completing this project you will have learned how to:

1. Create named FIFOs for inter-process communication.

2. Create threads to hide latencies and service multiple client requests.

3. Prevent race conditions.

4. Properly join or detach threads.

Grading

The project is worth a maximum of 100 points.

- Submit the document that contains the screenshots. +10
- Server is able to log the requests of multiple clients (even if there are errors). +10
- Completion of all of the components in FIFO.hpp. +25
- Completion of all of the components in ParallelLogger.cpp. +25
- There are no errors when 'make test' is ran. +10

run_tests.pl

9.3 Homework

9.3.1 Intro to C++11 Threads

Job.cpp

```
#include "Job.h"


void Job::doJob() {
    for(int i = 0; i < 10000; i++) {
        //doing absolutely nothing!
    }
    jobData *= 2;
}
```

Job.h

```
#ifndef _JOB
#define _JOB
class Job {
    public:
        void doJob();
        int jobID = 0; //Best to initialize to 0 (some compilers will not)
        int jobData = 0;
    private:
};
#endif
```

main.cpp

```
// program takes in three arguments.
// <max_threads> <max_jobs> <verbose>



//C++ Libraries
#include <iostream>
#include <thread>
```

```
#include <mutex>
#include <vector>
#include <random>
#include <ctime>
#include <cstdlib> //for atoi

// My Libraries
#include "Job.h"

using namespace std;

// Global Variable Hell
vector<Job> jobJar;
vector<Job> completedJobs;
vector<thread> threads;

mutex pop_mutex;
mutex push_mutex;

void jobDoer() {
    //do some stuff
    while(jobJar.size() > 0) {
        Job myJob;
        bool doJob = false;
        //using this method is not exception safe. Use std::lock_guard
        pop_mutex.lock();
            if(jobJar.size() > 0) {
                myJob = jobJar.back();
                jobJar.pop_back();
                doJob = true;
            }
        pop_mutex.unlock();

        if(doJob) { myJob.doJob(); }

        //using this method is not exception safe. Use std::lock_guard
        push_mutex.lock();
            completedJobs.push_back(myJob);
        push_mutex.unlock();
    }
}
```

```
int main(int argc, char *argv[]) {
    if(argc < 4) {
        cout << "Not enough arguments." << endl;
        cout << "usage: <MAX_THREADS> <MAX_JOBS> <VERBOSE>" << endl;
        cout << "MAX_THREADS and MAX_JOBS are both integers. "
             << "VERBOSE is either 0|1" << endl;
        return 1;
    } else if(argc > 4) {
        cout << "Too many arguments." << endl;
        cout << "usage: <MAX_THREADS> <MAX_JOBS>" << endl;
        cout << "MAX_THREADS and MAX_JOBS are both integers. "
             << "VERBOSE is either 0|1" << endl;
        return 1;
    } else { } // Continue the program

    int max_threads = atoi(argv[1]);
    int max_jobs = atoi(argv[2]);
    int verbose = atoi(argv[3]);

    //Generate a set of jobs for the threads.
    Job aJob;
    srand(time(NULL));
    for(int i = 0; i < max_jobs; i++) {
        aJob.jobID = i;
        aJob.jobData = (rand() % 100) + 1;
        jobJar.push_back(aJob);
    }

    //Generate a set of threads to pull from the jobJar.
    for(int i = 0; i < max_threads; i++) {
        threads.push_back(thread(jobDoer));
    }

    // Use an iterator to make sure that all of the thread complete their
    // execution.
    for(std::vector<thread>::iterator t = threads.begin();
            t != threads.end(); ++t) {
        t->join();
    }
```

```
    // Iterate and print out all of the completed jobs.
    if(verbose) {
        for(std::vector<Job>::iterator jb = completedJobs.begin();
                jb != completedJobs.end(); ++jb) {
           cout << "Job " << jb->jobID << " has a value of "
                << jb->jobData << "." << endl;
        }
    }

    return(0);
}
```

main.cpp.old

```
// Takes in two arguments. The maximum number of threads and the maximum number of jobs.
// ./syncron <MAX_THREADS> <MAX_JOBS> <VERBOSE>

//C++ Libraries
#include <iostream>
#include <vector>
#include <cstdlib> //for atoi

//C Libraries (notice the .h extension)
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

// My Libraries
#include "Job.h"

using namespace std;

// Global Variable Hell
vector<Job> jobJar;
vector<Job> completedJobs;
vector<pthread_t> tids;

pthread_mutex_t pop_mutex;
pthread_mutexattr_t pop_mutex_attr;
```

```
pthread_mutex_t push_mutex;
pthread_mutexattr_t push_mutex_attr;

void *jobDoer(void *param) {
    //do some stuff

    while(jobJar.size() > 0) {
        bool skipiter = false;
        Job myJob;
        pthread_mutex_lock(&pop_mutex);
            if(jobJar.size() > 0) {
                myJob = jobJar.back();
                jobJar.pop_back();
            } else {
                skipiter = true;
            }
        pthread_mutex_unlock(&pop_mutex);

        if(skipiter) {
            skipiter = false;
            continue;
            }

        myJob.doJob();

        pthread_mutex_lock(&push_mutex);
            completedJobs.push_back(myJob);
        pthread_mutex_unlock(&push_mutex);
    }

    //Using this prevents a compiler warning.
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if(argc < 4) {
        cout << "Not enough arguments." << endl;
        cout << "usage: <MAX_THREADS> <MAX_JOBS> <VERBOSE>" << endl;
        cout << "MAX_THREADS and MAX_JOBS are both integers. VERBOSE is either 0|1" << endl;
        return 1;
    } else if(argc > 4) {
```

```
    cout << "Too many arguments." << endl;
    cout << "usage: <MAX_THREADS> <MAX_JOBS>" << endl;
    cout << "MAX_THREADS and MAX_JOBS are both integers. VERBOSE is either 0|1" << endl;
    return 1;
} else { } // Continue the program

int max_threads = atoi(argv[1]);
int max_jobs = atoi(argv[2]);
int verbose = atoi(argv[3]);

//Generate a set of jobs for the threads.
Job aJob;
srand(time(NULL));
for(int i = 0; i < max_jobs; i++) {
    aJob.jobID = i;
    aJob.jobData = (rand() % 100) + 1;
    jobJar.push_back(aJob);
}

// Initialize the Mutexes
int popm_istat = pthread_mutex_init(&pop_mutex, &pop_mutex_attr);
if(popm_istat) {
    cout << "ERROR: Failed to initialize the pop mutex with error " << popm_istat << endl;
    return(1);
}
int pushm_istat = pthread_mutex_init(&push_mutex, &push_mutex_attr);
if(pushm_istat) {
    cout << "ERROR: Failed to initialize the push mutex with error " << popm_istat << endl;
    return(1);
}

//Generate a set of threads to pull from the jobJar.
for(int i = 0; i < max_threads; i++) {
    pthread_t tid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    int t_istat = pthread_create(&tid, &attr, jobDoer, NULL);
    if(t_istat) {
        cout << "ERROR: Failed to initilize thread " << i
            << " with error code " << t_istat << endl;
    }
```

```
        tids.push_back(tid);
    }

    // Use an iterator to make sure that all of the thread complete their execution.
    for(std::vector<pthread_t>::iterator tid = tids.begin(); tid != tids.end(); ++tid) {
        pthread_join(*tid, NULL);
    }

    // Iterate and print out all of the completed jobs.
    if(verbose == 1) {
        for(std::vector<Job>::iterator jb = completedJobs.begin(); jb != completedJobs.end(); ++jb)
 {
           cout << "Job " << jb->jobID << " has a value of " << jb->jobData << "." << endl;
        }
    }

    return(0);
}
```

speedtest.sh

```
#!/bin/bash

for max_threads in {1..20}
do
    time ./syncron $max_threads 100000 0
done
```

README.md

```
# Homework 1 Solution
## Introduction to C++11 and its Concurrency Features

## Description
This assignment will require that you update ‘‘main.cpp‘‘ such that it uses the
Standard Library threading components talked about in class.
If your system has been properly setup (or you are using a lab computer). This
assignment should take no more than two hours to complete.

## Directions
```

```
  0. Documentation
    1. Create a blank document that contains your full name and the assignment
    name (Homework 1).
  1. Coding Section
    1. First take a look at ``main.cpp.old``. This file contains the original
    code which makes use of the posix thread library.
    2. Take a look at ``main.cpp``. This will be the file that you will update
    such that it uses the C++11 Standard Library thread components.
    3. Update ``main.cpp`` such that all comments that contain _TODO_
    statements are replaced with the proper code.
    4. Build the code by running ``make all`` in the terminal. Simply running
    ``make`` will print what commands will be executed by the Makefile.
    5. You have the option to either run ``speedtest.sh`` or run the ``syncron``
    executable.
    6. Take a screenshot of the results and paste them into the document.
  2. Question Section
    1. In ``main.cpp`` locate the three questions in the comments.
    2. In your document write down your answer for each question.
    Make sure that I know what question each answer belongs to.

## Learning Goals
After completing this assignment you will have a basic understanding on how to
import the thread library and make use of threads and mutex locks to
access a job pool.

## Grading
The homework is worth a maximum of 100 points.

  - Answer the three questions located in the comments. +30
  - Replace all the _TODO_ components with the proper code. +50
  - Application compiles and runs without any issues. +20
  - Submit the assignment. +0
```

9.3.2 Inter-Process Communication

main.cpp

```
#include <unistd.h>
#include <sys/types.h>
#include <sysexits.h>
```

```
#include <string>
#include <cstring>
#include <iostream>
#include <vector>

// Makes life easier when writing and reading from the pipe.
struct pipe_message {
    char message[1024];
};

/// Function Prototypes.
int optarg_to_int(const char * optarg);

int main(int argc, char * argv[])
{
    if(argc <= 1)
    {
        std::cout << "USAGE: ./ipc <child spawn count>" << std::endl;
        return EX_USAGE;
    }
    int child_count = optarg_to_int(argv[1]);
    int actual_child_count = 0;

    // Initialize the message print request pipe.
    int req_pipe_desc[2];
    if(pipe(req_pipe_desc))
    {
        std::cout << "Failed to initialize req_pipe_desc terminating program"
            << std::endl;
        std::cout << strerror(errno) << std::endl;
        return EX_IOERR;
    }
    // Initialize the pipes.
    int * pipe_desc = new int[child_count * 2];
    for(int i = 0; i < child_count * 2; i += 2)
    {
        if(pipe(&pipe_desc[i]))
        {
            std::cout << strerror(errno) << std::endl;
        }
        else
```

```
    {
        actual_child_count++;
    }
}

// Spawn the required number of child processes that will
int pd_idx = 0;
int pid = 0;
int * pids = new int[child_count];
for(int i = 0; i < child_count; i++)
{
    pd_idx = i * 2;
    pids[i] = fork();
    pid = pids[i];
    if(pid > 0)
    {
        break;
    }
    else if( pid < 0 )
    {
        std::exit(EX_OSERR);
    }
}

// Send the hello world messages.
if(pid) {
    std::string message = "Hello from child: " + std::to_string(pd_idx / 2);
    struct pipe_message to_send;
    strcpy(to_send.message, message.c_str());

    if(write(req_pipe_desc[1], &pd_idx, sizeof(int)) >= 0)
    {
        if (write(pipe_desc[pd_idx + 1],
                    &to_send, sizeof(pipe_message)) < 0)
        {
            std::cout << strerror(errno) << std::endl;
        }
        // Close the child write-end.
        if(close(pipe_desc[pd_idx + 1] < 0)) {
            std::cout << strerror(errno) << std::endl;
            exit(EX_OSFILE);
```

```
        }
    }
    else
    {
        std::cout << strerror(errno) << std::endl;
    }

    exit(EX_OK);
}

// Read the messages
if(!pid) {
    int children_served = 0;
    while(children_served < actual_child_count)
    {
        int child_idx = 0;
        if(read(req_pipe_desc[0], &child_idx, sizeof(int)) == sizeof(int))
        {
            struct pipe_message to_receive;
            if(read(pipe_desc[child_idx],
                        &to_receive, sizeof(pipe_message)) < 0)
            {
                std::cout << strerror(errno) << std::endl;
            }
            std::cout << to_receive.message << std::endl;
            children_served++;

            // Close the read-end.
            if(close(pipe_desc[child_idx]) < 0) {
                std::cout << strerror(errno) << std::endl;
                exit(EX_OSFILE);
            }
        } else {
            std::cout << "Failed to read request pipe." << std::endl;
            std::cout << strerror(errno) << std::endl;
        }
    }
}

// Clean up a bit.
if(!pid)
```

```
    {
        for(int i = 0; i < child_count; i++)
        {
            int stat_loc = -1;
            waitpid(pids[i], &stat_loc, WNOHANG);
        }
    }

    delete pids;
    delete pipe_desc;

    close(req_pipe_desc[0]);
    close(req_pipe_desc[1]);

    return EX_OK;
}


/// Takes in an optarg (char pointer) and converts it into a
/// float.
///
/// \param[in] optarg The optarg char * from getopt.h
/// \returns a float representation of the optarg value.
int optarg_to_int(const char * optarg)
{
    std::string tmp(optarg);
    int flt = std::stoi(tmp);
    return flt;
}
```

Makefile.inc

```
UNAME_S := $(shell uname -s)

CXX = g++
CXXFLAGS += -pthread -std=c++11
ifeq ($(UNAME_S),Linux)
    CXX = g++-4.8

    # Set the libraries.
```

```
    GCC4_8_2_LIBS = -L/opt/gcc/4.8.2/lib64
    GCC4_8_2_INC = -I/opt/gcc/4.8.2/include
    CXXFLAGS += $(GCC4_8_2_LIBS) $(GCC4_8_2_INC)
    CXXFLAGS += -lnsl -Wl,--no-as-needed
endif
ifeq ($(UNAME_S),Darwin)
    CXX = g++-4.9
endif

CXXFLAGS += -Wall -Wpedantic

#.SILENT:
```

README.md

```
# Radix Sort Homework

## Description
Using the fork command and pipes setup a program that spawns a set of child processes that
 communicate back to the parent process using pipes.

# Directions
Complete 'main.cpp' such that:

  0. The program sets up an array of pipe descriptors. If there are $x$ child processes then $x$
 pipe descriptor *pairs* will be needed. Each pair consists of a write and read end respectively.
 For more information run 'man 2 pipe'.
  1. After initializing the pipes, start spawning the child processes.
  2. Have each child process send a message through the its pipe back to the parent. (Tip: Use a
 struct to store the message)
  3. Have the parent remain active until it has printed all of the messages from the child
 processes. You may need an additional pipe for storing a flag when there is a message that needs to
 be printed.

# Learning Goals

After completing this homework you will understand how to fork a process and use pipes to facilitate
 communication between the parent and child process.

# Grading

This homework is worth 100 points.
```

0. Implement an array of pipe descriptors. +10
1. Write code that spawns the child processes. +25
2. Each child should send a message through the pipe back to the parent. +25
3. The parent process will print the child message to the screen. +25
4. The program properly terminates. +15

9.3.3 CUDA Radix Sort

radix.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<cuda_runtime.h>
#include<assert.h>
#include<vector>
#include<algorithm>

#include"utils.h"
#include"scan.h"

void binary_radix_sort(std::vector<int> & in, std::vector<int> & out, unsigned int passes);
void seq_bit_remap(const std::vector<int> &to_remap, std::vector<int> &remapped,
                   const std::vector<int> &block_totals, const std::vector<int> &block_offsets,
                   const dim3 gridDims, const dim3 blockDims);

#define SEQ_SIZE    (1 << 27) //(1 << 10) //((1 << 27) + ( 1 << 26) + ( 1 << 25) + ( 1 << 24)) //(1
 << 26)
#define BLOCK_SIZE  (1 << 7)

/// Parallel version of the radix sort kernel.
/// This modified version of the parallel sort algorithm will only perform a single pass based on
 the
/// exponent passed in.
///
/// \param[in] exponent: The location in the bits to sort on.
/// \param[in, out] d_in: The unsorted set of elements.
/// \param[in, out] d_totals: Store the total counts of true and false for each block.
__global__ void kernel_radix_sort_1bit(int exponent, int * d_to_sort, int * d_sorted, int *
 d_block_totals) {
```

```
    __shared__ int s_to_sort[BLOCK_SIZE];
    __shared__ int s_predicate[BLOCK_SIZE];
    __shared__ int s_scan[BLOCK_SIZE];
    __shared__ int s_total_falses;

    // Copy the data to L2 cache.
    s_to_sort[threadIdx.x] = d_to_sort[threadIdx.x + blockIdx.x * blockDim.x];
    s_predicate[threadIdx.x] = 0;
    s_scan[threadIdx.x] = 0;

    if(threadIdx.x == 0) {
        s_total_falses = 0;
    }

    s_predicate[threadIdx.x] = !(s_to_sort[threadIdx.x] & (1 << exponent));
    s_scan[threadIdx.x] = s_predicate[threadIdx.x];
    __syncthreads();

    bl_sweep_up(s_scan, BLOCK_SIZE);
    __syncthreads();
    bl_sweep_down(s_scan, BLOCK_SIZE);
    __syncthreads();

    if(threadIdx.x == 0) {
        s_total_falses = s_scan[BLOCK_SIZE - 1] + s_predicate[BLOCK_SIZE - 1];
    }
    __syncthreads();

    // Build the scatter indexes.
    s_scan[threadIdx.x] = s_predicate[threadIdx.x] ? s_scan[threadIdx.x] : threadIdx.x -
 s_scan[threadIdx.x] + s_total_falses;
    d_sorted[s_scan[threadIdx.x] + blockIdx.x * blockDim.x] = s_to_sort[threadIdx.x];

    // Store the total falses and trues into the d_block_totals array for further processing.
    d_block_totals[blockIdx.x] = s_total_falses;
    d_block_totals[gridDim.x + blockIdx.x] = blockDim.x - s_total_falses;
}

/*! \brief Remaps the elements of a sequence
 *
 *  Remaps the elements of a sequence based on the offsets calculated from the earlier radix 1bit
```

```
 kernel
 *  call.
 *
 *  @param[in] The values to remap.
 *  @param[in, out] A temporaray staging ground.
 *  @param[in] The bit totals per block.
 *  @param[in] The offsets to map to.
 */
__global__ void bit_remap_kernel(int to_remap_size, int * d_to_remap, int * d_global_store,
                                 int * d_block_totals, int * d_offsets) {

    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;
    int tIdx = threadIdx.x;

    if(tIdx < d_block_totals[blockIdx.x]) {
        int mapped_index = d_offsets[blockIdx.x] + tIdx;
        d_global_store[mapped_index] = d_to_remap[gIdx];
    } else {
        int mapped_index = d_offsets[gridDim.x + blockIdx.x] + tIdx - d_block_totals[blockIdx.x];
        d_global_store[mapped_index] = d_to_remap[gIdx];
    }
}

/*! \brief Parallel Radix Sort using CUDA.
 *
 *  Calls the necessary functions to perform a GPGPU based radix sort using the CUDA API.
 *  >>> Requires the definition of BLOCK_SIZE in the source.
 *
 *  @param[in, out] The sequence to sort.
 *  @param[in] The sorted sequence.
 *
 *  @returns The execution time in milliseconds.
 */
float cuda_parallel_radix_sort(std::vector<int> & to_sort, std::vector<int> & sorted) {
    int gDmX = (int) to_sort.size() / BLOCK_SIZE;
    int bDmX = BLOCK_SIZE;

    assert(to_sort.size() == sorted.size());
    assert((int)to_sort.size() >= bDmX);
    assert((int)to_sort.size() % bDmX == 0);
```

```
    // For timing
    cudaEvent_t start, stop;
    float cuda_elapsed_time_ms = 0.0;

    // Vectors for storing the totals and scanned totals.
    std::vector<int> block_totals(gDmX * 2, 0);
    std::vector<int> block_totals_scanned(gDmX * 2, 0);

    int * d_to_sort = 0, * d_sorted = 0,
        * d_block_totals = 0, * d_block_totals_scanned = 0;

    unsigned int d_to_sort_mem_size = sizeof(int) * to_sort.size(),
                 d_sorted_mem_size  = d_to_sort_mem_size,
                 d_block_totals_mem_size = sizeof(int) * gDmX * 2,
                 d_block_totals_scanned_mem_size = d_block_totals_mem_size;

    // Initialize and begin the timers
    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&stop));
    checkCudaErrors(cudaEventRecord(start));

    // Allocate the memory on the device
    checkCudaErrors(cudaMalloc((void **)&d_to_sort, d_to_sort_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_sorted, d_sorted_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_block_totals, d_block_totals_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_block_totals_scanned, d_block_totals_mem_size));

    // Copy from the host to the device.
    checkCudaErrors(cudaMemcpy(d_to_sort, &to_sort[0],  d_to_sort_mem_size,
cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_sorted, &to_sort[0], d_sorted_mem_size,  cudaMemcpyHostToDevice));

    // Calculate the block and grid dimensions.
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(bDmX, 1);

    // Sort at each bit.
    for(unsigned int exponent = 0; exponent < sizeof(int) * 8; ++exponent) {
        // First perform a single bit sort.
        kernel_radix_sort_1bit<<<gridDims, blockDims>>>(exponent, d_to_sort, d_sorted,
d_block_totals);
```

```
        checkCudaErrors(cudaGetLastError());
        checkCudaErrors(cudaThreadSynchronize());

        // Copy from the device to the host.
        checkCudaErrors(cudaMemcpy(&block_totals[0], d_block_totals, d_block_totals_mem_size,
                                   cudaMemcpyDeviceToHost));

        // Now perform a scan on the totals.
        cuda_parallel_scan(BLOCK_SIZE, block_totals, block_totals_scanned);

        // Copy from the host to the device.
        checkCudaErrors(cudaMemcpy(d_block_totals_scanned, &block_totals_scanned[0],
                                   d_block_totals_scanned_mem_size, cudaMemcpyHostToDevice));

        // Now we can map the values.
        int * d_remapped = d_to_sort; //for readability and clarity.
        bit_remap_kernel<<<gridDims, blockDims>>>(sorted.size(), d_sorted, d_remapped,
 d_block_totals,
                                                 d_block_totals_scanned);
        checkCudaErrors(cudaGetLastError());
        checkCudaErrors(cudaThreadSynchronize());
    }

    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&sorted[0], d_sorted, d_sorted_mem_size, cudaMemcpyDeviceToHost));

    // Free up the device.
    checkCudaErrors(cudaFree(d_to_sort));
    checkCudaErrors(cudaFree(d_sorted));
    checkCudaErrors(cudaFree(d_block_totals));
    checkCudaErrors(cudaFree(d_block_totals_scanned));

    // Stop the timers
    checkCudaErrors(cudaEventRecord(stop));
    checkCudaErrors(cudaEventSynchronize(stop));
    checkCudaErrors(cudaEventElapsedTime(&cuda_elapsed_time_ms, start, stop));

    return cuda_elapsed_time_ms;
}
```

```
/*! \brief Sequential Bit Remap
 *
 *  Serial implementation of the bit remap kernel.
 *
 *  @param[in] to remap.
 *  @param[in, out] remapped.
 *  @param[in] block totals.
 *  @param[in] block offsets.
 */
void seq_bit_remap(const std::vector<int> &to_remap, std::vector<int> &remapped,
                   const std::vector<int> &block_totals, const std::vector<int> &block_offsets,
                   const dim3 gridDims, const dim3 blockDims) {

    for(unsigned int bIdX = 0; bIdX < (unsigned int) gridDims.x; ++bIdX) {
        for(unsigned int tIdX = 0; tIdX < (unsigned int) blockDims.x; ++tIdX) {
            unsigned int gIdX = tIdX + bIdX * (unsigned int) blockDims.x;

            if(tIdX < (unsigned int) block_totals[bIdX]) {
                unsigned int mapping = tIdX + (unsigned int) block_offsets[bIdX];
                remapped[mapping] = to_remap[gIdX];
            } else {
                unsigned int mapping = (tIdX - block_totals[bIdX]) + (unsigned int)
 block_offsets[gridDims.x + bIdX];
                remapped[mapping] = to_remap[gIdX];
            }
        }
    }
}

/*! \brief Wikipedia Radix Sort Implementation
 *
 *  The C radix sort implementation from wikipia. Modified to to handle vectors.
 *
 *  @param[in, out] The sequence to sort.
 *  @param[in] The sorted sequence.
 *  @param[in] The base to sort on.
 */
void radix_sort(std::vector<int> & in, std::vector<int> & out, int base) {
    out = in; // copy in to out.
```

```
    std::vector<int> partial_out(out.size());
    int exp = 1;

    // find the maximum value in the vector.
    int max = out[0];
    for(std::vector<int>::iterator it = out.begin(); it != out.end(); ++it) {
        if( max < *it ) {
            max = *it;
        }
    }

    // Start sorting.
    while(max / exp > 0) {
        std::vector<int> bucket(base, 0);

        // bin up the values.
        for(std::vector<int>::iterator it = out.begin(); it != out.end(); ++it) {
            bucket[(*it / exp) % base]++; // why do you not want to use a bucket on the GPU?? Then
 you have to lock the cell to prevent race conditions. That is not good!
        }

        // perform a SCAN to aquire the indexes.
        for(unsigned int i = 1; i < bucket.size(); ++i) {
            bucket[i] += bucket[i - 1];
        }

        // Now use the calculated indexes to copy out to partial_out.
        for(int i = out.size() - 1; i >= 0; --i) {
            partial_out[--bucket[(out[i] / exp) % base]] = out[i];
        }

        // copy back to out.
        out = partial_out;

        // look at the next sig digit.
        exp *= base;
    }
}

/*! \brief Binary version of radix sort.
 *
```

```
 *  Binary implemenation of radix sort. Function setup such that it is
 *  easier to compare to the CUDA implementation.
 *
 *  @param[in] the sequence to sort.
 *  @param[out] the sorted sequence.
 */
void binary_radix_sort(std::vector<int> & in, std::vector<int> & out) {
    out = in;

    std::vector<int> tmp(in.size(), 0);
    for(unsigned int exponent = 0; exponent < sizeof(int) * 8; ++exponent) {
        int i_n = 0;
        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(!(out[i] & (1 << exponent))) {

                tmp[i_n] = out[i];
                ++i_n;
            }
        }

        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(out[i] & (1 << exponent)) {
                tmp[i_n] = out[i];
                ++i_n;
            }
        }

        out = tmp;
    }
}

/*! \brief Binary version of radix sort.
 *
 *  Binary implemenation of radix sort. Function setup such that it is
 *  easier to compare to the CUDA implementation.
 *
 *  @param[in] the sequence to sort.
 *  @param[out] the sorted sequence.
 *  @param[in] the number of passes.
 */
void binary_radix_sort(std::vector<int> & in, std::vector<int> & out, unsigned int passes) {
```

```
    out = in;

    std::vector<int> tmp(in.size(), 0);
    for(unsigned int exponent = 0; exponent < passes; ++exponent) {
        //printf("brs: %i\n", exponent);
        int i_n = 0;
        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(!(out[i] & (1 << exponent))) {

                tmp[i_n] = out[i];
                ++i_n;
            }
        }

        for(unsigned int i = 0; i < tmp.size(); ++i) {
            if(out[i] & (1 << exponent)) {
                tmp[i_n] = out[i];
                ++i_n;
            }
        }

        out = tmp;
    }
}

int main(void) {
    reset_cuda_devs();
    srand( 0 /*time(NULL)*/);

    //printf("Maximum Vector Size %u\n", seq_to_sort.max_size());
    printf("Vector Size %i\n", SEQ_SIZE);
    printf("Estimated Memory Usage is %f MB\n", (float) (SEQ_SIZE * sizeof(int)) / 1e6 * 4.0);
    printf("Allocating four vectors\n");
    std::vector<int> seq_to_sort(SEQ_SIZE, 0);
    std::vector<int> sorted(SEQ_SIZE, 0);
    std::vector<int> gpu_sorted(SEQ_SIZE, 0);
    incremental_fill(seq_to_sort);
    //std::reverse(seq_to_sort.begin(), seq_to_sort.end());
    shuffle(seq_to_sort);

    // Implement classic radix sort algorithm.
```

```
    printf("Performing Sequential Sort\n");
    clock_t t = clock();
    binary_radix_sort(seq_to_sort, sorted);
    t = clock();

    float t_sec =  (float)t / (float) CLOCKS_PER_SEC;
    if(t_sec < 1) {
        printf("Done. Took %0.2f ms\n", t_sec * 1e3);
    } else {
        printf("Done. Took %0.2f s\n", t_sec );
    }



    // Implement gpu radix sort algorithm.
    printf("Performing Parallel Sort\n");
    printf("To analyze the performance run 'nvprof ./radix_sort'. \n");
    for(int i = 0; i < 100; i++) {
        printf("--------------------------[ %i ]-----------------------------------\n", i);
        float cuda_runtime_ms = cuda_parallel_radix_sort(seq_to_sort, gpu_sorted);
        if(cuda_runtime_ms < 1000.0) {
            printf("Done. Took %0.2f ms.\n", cuda_runtime_ms);
        } else {
            printf("Done. Took %0.2f s.\n", cuda_runtime_ms / 1000.0);
        }

        int miss_index = equal(sorted, gpu_sorted);
        if( miss_index != -1 ) {
            printf("Expected %i got %i at index %i\n", sorted[miss_index], gpu_sorted[miss_index],
 miss_index);
        } else {
            printf("Success!\n");
        }
    }

    return 0;
}
```

scan.cu

```
#include"scan.h"
```

```
#include"utils.h"

// find the log_2 of a value
inline __device__ int log2(int i) {
    int p = 0;
    while(i >>= 1) p++;
    return p;
}

/*! \brief Blelloch scan sweep-up operation
 *
 *  Performs the sweep-up substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_up(int * to_sweep, int size) {
    //1: for d = 0 to log2 n  1 do
    //2:      for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //3:           x[t] = x[t] + x[t - t / 2]
    int t = threadIdx.x;
    for( int d =  0; d < log2(size); ++d) {
        __syncthreads(); // wrapping the condition with synthreads prevents a rare race condition.
        if( t < size && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
            to_sweep[t] = to_sweep[t] + to_sweep[tp];
        }
        __syncthreads();
    }
}

/*! \brief Blelloch scan sweep-down operation
 *
 *  Performs the sweep-down substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_down(int * to_sweep, int size) {
    //1: x[n  1] <- 0
    //2: for d = log2 n  1 down to 0 do
```

```
    //3:       for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //4:            carry = x[t]
    //5:            x[t] += x[t - t / 2]
    //6:            x[t - t / 2] = carry
    to_sweep[size - 1] = 0;
    int t = threadIdx.x;
    for( int d = log2(size) - 1; d >= 0; --d) {
        __syncthreads(); // wrapping the condition with synthreads prevents a rare race condition.
        if( (t < size) && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
            int tmp = to_sweep[t];
            to_sweep[t] += to_sweep[tp];
            to_sweep[tp] = tmp;
        }
        __syncthreads();
    }
}

/*! \brief Blelloch Scan Kernel
 *
 *  The blelloch scan algorithm kernel. Implementation based on Udacity's and
 *  NVidia's explanations. This will only do a block level scan.
 *
 *  Udacity: https://www.youtube.com/watch?v=_5sM-4ODXaA
 *  Nvidia: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
 *
 *  @param[in] the sequence to scan. Assumes that the sequence is no larger than the block size.
 *  @param[in, out] the sequence to store the scanned results.
 */
__global__ void kernel_blelloch_scan(const int d_to_scan_size, const int * d_to_scan, int *
 d_scanned) {
    int tIdx = threadIdx.x;

    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = 0;

    if(tIdx < d_to_scan_size) {
        //copy d_to_scanto s_to_scan memory to speed up the performance.
        s_to_scan[tIdx] = d_to_scan[tIdx];
    }
    __syncthreads();
```

```
    // perform the sweep up phase of the algorithm
    bl_sweep_up(s_to_scan, blockDim.x);
    // perform the sweep down phase of the algorithm
    bl_sweep_down(s_to_scan, blockDim.x);


    if(tIdx < d_to_scan_size) {
        //copy back to global memory.
        d_scanned[tIdx] = s_to_scan[tIdx];
    }
}

/*! \brief Block scan kernel function.
 *
 *  The kernel_block_scan kernel takes in a sequence of elemetns and performs a block level scans
 *  accross the sequence. The block dimensions need to be of size 2^x <= 1024 (or cuda specific
 maximum block size).
 *
 *  @param[in] the sequence of integers to scan.
 *  @param[in, out] the sequence to store the block level scans.
 *  @param[in, out] the sequence to store the reductions of each block. AKA it's length is the
 *      length of the grid.
 */
__global__ void kernel_block_scan(const int d_to_scan_size, const int * d_to_scan, int * d_scanned,
 int * d_block_reductions) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    // Init the dynamic shared memory.
    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = d_to_scan[d_to_scan_size - 1];

    // Copy from global to local.
    if(gIdx < d_to_scan_size) {
        s_to_scan[tIdx] = d_to_scan[gIdx];
    }
    __syncthreads();

    // Scan the shared block of data.
    bl_sweep_up(s_to_scan, blockDim.x);
```

```
    bl_sweep_down(s_to_scan, blockDim.x);

    // Now store the block reduction (sum of all elets)
    if(tIdx == (blockDim.x - 1)) {
        d_block_reductions[blockIdx.x] = s_to_scan[tIdx] + d_to_scan[gIdx];
    }
    __syncthreads();


    //Push the results back out to global memory.
    if(gIdx < d_to_scan_size) {
        d_scanned[gIdx] = s_to_scan[tIdx];
    }
}

/*! \brief Offset blocks
 *
 *  The kernel_block_offset takes in a sequence of values and performs an the corresponding block
 *  offset from the offset sequence.
 *
 *  @param[in, out] the sequence to offset.
 *  @param[in] the corresponding block offset values.
 */
__global__ void kernel_block_offset(int d_to_offset_size, int * d_to_offset, int * d_offsets) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    extern __shared__ int s_to_offset[];
    s_to_offset[tIdx] = 0;

    // Copy from global to shared.
    if(gIdx < d_to_offset_size) {
        s_to_offset[tIdx] = d_to_offset[gIdx];
    }
    __shared__ int offset;
    offset = d_offsets[blockIdx.x];
    __syncthreads();

    s_to_offset[tIdx] += offset;
    __syncthreads();
```

```
    // Copy back to global memory.
    if(gIdx < d_to_offset_size) {
        d_to_offset[gIdx] = s_to_offset[tIdx];
    }
}

/*! \brief CUDA Parallel Scan
 *
 * See cuda_parallel_scan(int, std::vector<int>, std::vector<int>, const bool debug)
 *
 */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned) {
    return cuda_parallel_scan(block_size, to_scan, scanned, false);
}

/*! \brief CUDA Parallel Scan
 *
 *  The cuda_parallel_scan is a recursive function that takes in a large set of values and applies
 *  the scan operator across the entire set.
 *
 *  @param[in] The size of the block to use.
 *  @param[in] the vector of elements to scan. The length must ALWAYS be a multiple of block_size
 and
 *     to_scan.size() / block_size must never surpass 2147483648 on CUDA 3.0 devices.
 *  @param[in, out] a vector to store the scan results.
 *
 *  @returns the runtime of all the function calls involving the device.
 */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned,
 const bool debug) {
    // For timing
    cudaEvent_t start, stop;
    float cuda_elapsed_time_ms = 0.0;

    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_scan.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_parallel_scan\n");
```

```
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX, to_scan.size(),
block_size);
    }

    // Vector initializations
    std::vector<int> block_reductions(gDmX, 0);
    std::vector<int> scanned_block_reductions(block_reductions.size(), 0);

    // create our pointers to the device memory (d prefix)
    int * d_to_scan = 0,
        * d_scanned = 0,
        * d_block_reductions = 0;

    // memory allocation sizes
    unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
                 scanned_mem_size = sizeof(int) * scanned.size(),
                 block_reductions_mem_size = sizeof(int) * block_reductions.size();

    // Initialize and begin the timers
    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&stop));
    checkCudaErrors(cudaEventRecord(start));

    // Allocate the memory on the device
    checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_block_reductions, block_reductions_mem_size));

    // Copy from the host to the device.
    checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_block_reductions, &block_reductions[0], block_reductions_mem_size,
                               cudaMemcpyHostToDevice));

    // Calculate the shared memory size.
    unsigned int shared_mem_size = sizeof(int) * blockDims.x;

    // Run a block level scan.
    kernel_block_scan<<<gridDims, blockDims, shared_mem_size>>>(to_scan.size(), d_to_scan,
d_scanned, d_block_reductions);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());
```

```
    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size, cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(&block_reductions[0], d_block_reductions, block_reductions_mem_size,
                               cudaMemcpyDeviceToHost));

    // Free GPU resources before recursing.
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
    checkCudaErrors(cudaFree(d_block_reductions));



    // If the size of block_reductions is larger than a single block then recurse and call
    //  cuda_parallel_scan on the block_reductions vector.
    // Else, pass the block_reductions vector to kernel_blelloch_scan.
    if(block_reductions.size() > (unsigned int) block_size) {
        cuda_parallel_scan(block_size, block_reductions, scanned_block_reductions, debug);
    } else {
        cuda_blelloch_scan(block_size, block_reductions, scanned_block_reductions, debug);
    }

    // Now perform the offsets to get a global scan.
    cuda_block_offset(block_size, scanned, scanned_block_reductions, debug);

    // Stop the timers
    checkCudaErrors(cudaEventRecord(stop));
    checkCudaErrors(cudaEventSynchronize(stop));
    checkCudaErrors(cudaEventElapsedTime(&cuda_elapsed_time_ms, start, stop));

    return cuda_elapsed_time_ms;
}

/*! \brief CUDA Blelloch scan.
 *
 * The cuda-blelloch-scan method will only perfrom a single block scan of a sequence using cuda.
 * For devices with a compute capability of 3.0 then the maximum block size is 1024.
 *
 *  @param[in] the vector of elements to scan. The size of the sequence must not surpass the
 *      maximum supported block size on the device.
 *  @param[in, out] a vector to store the scan results.
 */
```

```
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned,
 const bool debug) {
    int * d_to_scan = 0,
        * d_scanned = 0;

    unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
                 scanned_mem_size = to_scan_mem_size;

    if(debug) {
        printf("---> Entering: cuda_blelloch_scan\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", 1, to_scan.size(),
 block_size);
    }

    // Allocate the memory on the devicee
    checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));

    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size, cudaMemcpyHostToDevice));

    // Calculate the shared memory size
    unsigned int shared_mem_size = sizeof(int) * block_size;

    // Calculate the block and grid dimensions.
    dim3 gridDims(1, 1);
    dim3 blockDims(block_size, 1);

    // Run the blelloch scan on the block_reductions array.
    kernel_blelloch_scan<<<gridDims, blockDims, shared_mem_size>>>(to_scan.size(), d_to_scan,
 d_scanned);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size, cudaMemcpyDeviceToHost));

    // Clean up
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
}
```

```
/*! \brief CUDA Parallel Offset
 *
 * Takes in a sequence and offsets the values per block. The offsets sequence contains the
 * per block offsets.
 *
 *  @param[in] the size of the block.
 *  @param[in, out] the sequence of elements to perform block level offsets.
 *  @param[in] the offset values.
 */
void cuda_block_offset(int block_size, std::vector<int> & to_offset, std::vector<int> offsets, const
 bool debug) {
    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_offset.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_block_offset\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX, to_offset.size(),
 blockDims.x);
    }

    int * d_to_offset = 0,
        * d_offsets = 0;

    unsigned int to_offset_mem_size = sizeof(int) * to_offset.size(),
                 offsets_mem_size = sizeof(int) * offsets.size();

    // Allocate the memory
    checkCudaErrors(cudaMalloc((void **)&d_to_offset, to_offset_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_offsets, offsets_mem_size));

    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_offset, &to_offset[0], to_offset_mem_size,
 cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_offsets, &offsets[0], offsets_mem_size, cudaMemcpyHostToDevice));

    // Calculate the shared memory size.
    unsigned int shared_mem_size = sizeof(int) * blockDims.x;
```

```
    // Execute the kernel
    kernel_block_offset<<<gridDims, blockDims, shared_mem_size>>>(to_offset.size(), d_to_offset,
 d_offsets);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&to_offset[0], d_to_offset, to_offset_mem_size,
 cudaMemcpyDeviceToHost));

    // Free unused memory
    checkCudaErrors(cudaFree(d_to_offset));
    checkCudaErrors(cudaFree(d_offsets));
}

/*! \brief Sequential Sum Scan
 *
 *  @param[in] the vector to scan.
 *  @param[in, out] the vector to store the scanned results.
 *  @param[in] perform an inclusive or exclusive scan.
 */
void sequential_sum_scan(std::vector<int> & in, std::vector<int> & out, bool inclusive) {
    assert(in.size() == out.size());

    int sum = 0;
    for( std::vector<int>::iterator in_it = in.begin(), out_it = out.begin();
            (in_it != in.end()) && (out_it != in.end());
            ++in_it, ++out_it) {

        if( inclusive ) {
            sum += *in_it; //include the first element of in.
            *out_it = sum;
        } else {
            *out_it = sum; //do not include the first element of in.
            sum += *in_it;
        }
    }
}
```

scan.h

```
#ifndef SCAN_H
#define SCAN_H

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<cuda_runtime.h>
#include<assert.h>
#include<vector>

/* Kernel Function Prototypes */
__device__ int log2(int i);
__device__ void bl_sweep_up(int * to_sweep, int size);
__device__ void bl_sweep_down(int * to_sweep, int size);
__global__ void kernel_blelloch_scan(int * d_to_scan, int * d_scanned);
__global__ void kernel_block_scan(int * d_to_scan, int * d_scanned, int * d_block_reductions);
__global__ void kernel_block_offset(int * d_to_offset, int * d_offsets);

/* CUDA Function Prototypes */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned);
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                         std::vector<int> & scanned, const bool debug);
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned, const bool debug);
void cuda_block_offset(int block_size, std::vector<int> & to_offset,
                       std::vector<int> offsets, const bool debug);

/* Serial Functions */
void sequential_sum_scan(std::vector<int> & in, std::vector<int> & out, bool inclusive);

#endif
```

utils.h

```
/// The original source for this file is available from Udacity's github repository for
/// their "Intro to Parallel Computing" course.
/// url:
 https://raw.githubusercontent.com/udacity/cs344/master/Problem%20Sets/Problem%20Set%201/utils.h
/// Use this when programming!
```

```
#ifndef UTILS_H
#define UTILS_H

#include <iostream>
#include <iomanip>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <cassert>
#include <cmath>

#define checkCudaErrors(val) check( (val), #val, __FILE__, __LINE__)

// Pass all cuda api calls to this fuction. That will help you find mistakes quicker.
template<typename T>
void check(T err, const char* const func, const char* const file, const int line) {
  if (err != cudaSuccess) {
    std::cerr << "CUDA error at: " << file << ":" << line << std::endl;
    std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
    exit(1);
  }
}

// Swap two pointers.
inline void swap_pointers(int ** p0, int ** p1) {
    int * temp = *p0;
    *p0 = *p1;
    *p1 = temp;
}

// Fills a vector with random numbers.
inline void random_fill(std::vector<int> & to_fill) {
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
        *it = (int)((float)rand() / RAND_MAX);
    }
}

// Fills a vector of size t from 0 ... t - 1.
inline void incremental_fill(std::vector<int> & to_fill) {
    int i = 0;
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
```

```
        *it = i;
        i++;
    }
}

// Dump the contents of a vector.
inline void print_vector(const std::vector<int> & to_print) {
    for(std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        printf("%i ", *it);
    }
    printf("\n");
}

// Performs a fisher yates shuffle on the array.
inline void shuffle(std::vector<int> & to_shuffle) {
    for( unsigned int i = 0; i < to_shuffle.size(); ++i) {
        unsigned int j = (unsigned int)((float) rand() / RAND_MAX * (i + 1));

        int tmp = to_shuffle[j];
        to_shuffle[j] = to_shuffle[i];
        to_shuffle[i] = tmp;
    }
}

// Prints the partitioning of a vector if it were to be passed to a CUDA kernel.
inline void print_vector_with_dims(const std::vector<int> & to_print,
                                   const int blockDimX, const int gridDimX) {
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        if(bDx_counter && !(bDx_counter % blockDimX)) {
            printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                printf("****");
                gDx_counter++;
            }
            printf("\n");
        }
        printf("%i ", *it);
```

```
        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Print two vectors side by side.
inline void print_vector_comparison(const std::vector<int> & v1,
                                    const std::vector<int> & v2,
                                    const int start_index, const int stop_index,
                                    const int point_at, const int blockDimX,
                                    const int gridDimX) {
    int index = 0;
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end()) && (index < stop_index);
         ++it1, ++it2, ++index) {

        if(bDx_counter && !(bDx_counter % blockDimX)) {
            if(index >= start_index) printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                if(index >= start_index) printf("****");
                gDx_counter++;
            }
            if(index >= start_index) printf("\n");
        }

        if(index == point_at) {
            if(index >= start_index) printf("\n|->(%i, %i)<-|\n", *it1, *it2);
        } else {
            if(index >= start_index) printf("(%i, %i) ", *it1, *it2);
        }

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Compare to vectors. If equal return -1 else return index.
inline int equal(const std::vector<int> & v1, const std::vector<int> & v2) {
```

```
    int index = 0;
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end());
         ++it1, ++it2, ++index) {

        if(*it1 != *it2) {
            return index;
        }
    }

    return -1;
}



inline void reset_cuda_devs() {
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    while( dev_count --> 0 )
    {
        printf("Resetting device %i", dev_count);
        cudaSetDevice(dev_count);
        cudaDeviceReset();
    }
    printf("\n");
}

#endif
```

README.md

```
# Radix Sort Homework

## Description
Implement a block-level radix sort on a set of integers.

# Directions
  - Complete `radix.cu` such that the following functions have been implemented.
    1. `cuda_parallel_radix_sort`
    2. `kernel_radix_sort`
  - `cuda_parallel_radix_sort` will serve as the staging grounds to copy data from and to the GPU.
  - `kernel_radix_sort` will contain the parallel sorting instructions.
```

```
  - To build the program for debugging run ‘make clean && make debug‘.
  - To build a release then run ‘make clean && make debug‘.

# Learning Goals

After completing this assignment you will have a basic understanding how to sort sequences in
 parallel on the GPU. In conjunction to sorting, you will have additional exposure to initializing
 variables in static memory.

# Grading

This homework is worth 100 points.

  - Implement all of the necessary code in ‘cuda_parallel_radix_sort‘. +10
  - Implement ‘kernel_radix_sort‘.
    0. Allocate the shared memory in the kernel. +15
    1. Calculate the predicate given the target bits in the sequence of numbers. +15
    2. Using the either the supplied parallel scanning function or you own scan the predicate. +15
    3. Calculate the scatter indexes. +15
    4. Remap the sequence of numbers to their new locations. +15
    5. Repeat steps 1-5 for all of the bit locations. +15
  - EXTRA CREDIT: If possible use the scan function you created in the previous homework. +5
```

9.3.4 CUDA Scanning

scan.cu

```
/*!
 * @file main.cu
 * @date 14 Dec 2014
 * @brief Main entry point for the scanning algorithm solutions.
 *
 * Unlike the Unsovled portion of this homework. The solution is able run the scan
 * algorithm on sequences that are larger than the maximum supported block size.
 * Given the difficulty of performing a multiblock scan the portion of the homework
 * should only require students to scan a sequence that is no larger than a single block.
 *
 */

#include<stdio.h>
```

```
#include<stdlib.h>
#include<time.h>
#include<cuda_runtime.h>
#include<assert.h>
#include<vector>

#include"utils.h"
#include"scan_utils.h"

#define SEQ_SIZE    134217757 //4194319  //1083109 //(1 << 20) //8192 //((1 << 27) + ( 1 << 26) + (
 1 << 25) + ( 1 << 24))
#define BLOCK_SIZE  256     //1024

// Function Declarations
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned);
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan,
                         std::vector<int> & scanned, const bool debug);
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan,
                        std::vector<int> & scanned, const bool debug);
void cuda_block_offset(int block_size, std::vector<int> & to_offset,
                       std::vector<int> offsets, const bool debug);

/*!
 *  \brief Program Entry Point
 *
 * First run and benchmark the serial scan. Then run and benchmark
 * the parallel scan using different block sizes.
 *
 */
int main(void) {
    reset_cuda_devs();
    srand(0);

    std::vector<int> seq_to_scan(SEQ_SIZE, 0);
    std::vector<int> scanned(SEQ_SIZE, 0);
    std::vector<int> gpu_scanned(SEQ_SIZE, 0);
    random_fill(seq_to_scan, 100);

    clock_t t = clock();
    sequential_sum_scan(seq_to_scan, scanned, false);
```

```
    t = clock();
    float t_sec =  (float)t / (float) CLOCKS_PER_SEC;
    if(t_sec < 1) {
        printf("Done. Took %0.2f ms\n", t_sec * 1e3);
    } else {
        printf("Done. Took %0.2f s\n", t_sec );
    }

    float cuda_runtime_ms = 0.0;
    for(int i = 4; (1 << i) <= 1024; i++) {
        for(int j = 0; j < 100; j++) {
            printf("---------------------------[ %i, %i ]------------------------------------\n", (1
 << i), j);
            cuda_runtime_ms = cuda_parallel_scan((1 << i), seq_to_scan, gpu_scanned);


            if(cuda_runtime_ms < 1000.0) {
                printf("Done. Took %0.2f ms.\n", cuda_runtime_ms);
            } else {
                printf("Done. Took %0.2f s.\n", cuda_runtime_ms / 1000.0);
            }

            int miss_index = equal(scanned, gpu_scanned);
            if( miss_index != -1 ) {
                printf("Missed at index %i\n", miss_index);
                printf("Expected %i got %i\n", scanned[miss_index], gpu_scanned[miss_index]);
            } else {
                printf("Scans match.\n");
            }
        }
    }
    return 0;
}

/*!
 *  \brief Calculate the base 2 log on the GPU.
 */
inline __device__ int log2(int i) {
    int p = 0;
    while(i >>= 1) p++;
    return p;
```

```
}

/*! \brief Hillis and Steel sum scan algorithm
 *
 * Work inefficient requires O(n log(n)) operations compared to n.
 * Udacity: https://www.youtube.com/watch?v=_5sM-4ODXaA
 * Nvidia: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
 * NOTE: Slightly modified so that the code makes more sense.
 *
 * This kernel will only work on a single block.
 *
 * @param[in] the sequence of elements to scan.
 * @param[in, out] results stored in the out sequence
 * @param[in] inclusive or exclusive scan.
 */
__global__ void hs_scan_kernel(int * d_to_scan, int * d_scanned, bool inclusive) {
    //if(threadIdx.x == 0 && blockIdx.x == 0) printf("Commencing Hillis-Steele scan\n");

    int tIdx = threadIdx.x;

    __shared__ int s_to_scan[BLOCK_SIZE];

    //copy in to s_to_scan memory to speed up the performance.
    if( !inclusive && (tIdx < BLOCK_SIZE - 1)) {
        s_to_scan[tIdx + 1] = d_to_scan[tIdx];
    } else {
        s_to_scan[tIdx] = d_to_scan[tIdx];
    }
    __syncthreads();

    for( int d = 1; d <= log2(SEQ_SIZE); ++d ) {
        if( tIdx >= (1 << (d - 1))) {
            s_to_scan[tIdx] += s_to_scan[tIdx - (1 << (d - 1))];
        }
        __syncthreads();
    }

    d_scanned[tIdx] = s_to_scan[tIdx];
    __syncthreads();
}
```

```
/*! \brief Blelloch scan sweep-up operation
 *
 *  Performs the sweep-up substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_up(int * to_sweep, int size) {
    //1: for d = 0 to log2 n  1 do
    //2:      for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //3:           x[t] = x[t] + x[t - t / 2]
    int t = threadIdx.x;
    for( int d =  0; d < log2(size); ++d) {
        __syncthreads(); // wrapping the condition with synthreads prevents a rare race condition.
        if( t < size && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
            to_sweep[t] = to_sweep[t] + to_sweep[tp];
        }
    }
}

/*! \brief Blelloch scan sweep-down operation
 *
 *  Performs the sweep-down substep in the blelloch scan.
 *
 *  @param[in, out] the sequence to sweep.
 *  @param[in] the size of the sequence to sweep.
 */
__device__ void bl_sweep_down(int * to_sweep, int size) {
    //1: x[n  1] <- 0
    //2: for d = log2 n  1 down to 0 do
    //3:       for all t = 0 to n  1 by 2^{d + 1} in parallel do
    //4:            carry = x[t]
    //5:            x[t] += x[t - t / 2]
    //6:            x[t - t / 2] = carry
    to_sweep[size - 1] = 0;
    int t = threadIdx.x;
    for( int d = log2(size) - 1; d >= 0; --d) {
        __syncthreads(); // wrapping the condition with synthreads prevents a rare race condition.
        if( (t < size) && !((t + 1) % (1 << (d + 1)))) {
            int tp = t - (1 << d);
```

```
            int tmp = to_sweep[t];
            to_sweep[t] += to_sweep[tp];
            to_sweep[tp] = tmp;
        }
    }
}

/*! \brief Blelloch Scan Kernel
 *
 *  The blelloch scan algorithm kernel. Implementation based on Udacity's and
 *  NVidia's explanations. This will only do a block level scan.
 *
 *  Udacity: https://www.youtube.com/watch?v=_5sM-4ODXaA
 *  Nvidia: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
 *
 *  @param[in] the sequence to scan. Assumes that the sequence is no larger than the block size.
 *  @param[in, out] the sequence to store the scanned results.
 */
__global__ void kernel_blelloch_scan(const int d_to_scan_size, const int * d_to_scan, int *
 d_scanned) {
    int tIdx = threadIdx.x;

    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = 0;

    if(tIdx < d_to_scan_size) {
        //copy d_to_scanto s_to_scan memory to speed up the performance.
        s_to_scan[tIdx] = d_to_scan[tIdx];
    }
    __syncthreads();

    // perform the sweep up phase of the algorithm
    bl_sweep_up(s_to_scan, blockDim.x);
    // perform the sweep down phase of the algorithm
    bl_sweep_down(s_to_scan, blockDim.x);


    if(tIdx < d_to_scan_size) {
        //copy back to global memory.
        d_scanned[tIdx] = s_to_scan[tIdx];
    }
```

```
}

/*! \brief Block scan kernel function.
 *
 *  The kernel_block_scan kernel takes in a sequence of elemetns and performs a block level scans
 *  accross the sequence. The block dimensions need to be of size 2^x <= 1024 (or cuda specific
 maximum block size).
 *
 *  @param[in] the sequence of integers to scan.
 *  @param[in, out] the sequence to store the block level scans.
 *  @param[in, out] the sequence to store the reductions of each block. AKA it's length is the
 *      length of the grid.
 */
__global__ void kernel_block_scan(const int d_to_scan_size, const int * d_to_scan, int * d_scanned,
 int * d_block_reductions) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    // Init the dynamic shared memory.
    extern __shared__ int s_to_scan[];
    s_to_scan[tIdx] = d_to_scan[d_to_scan_size - 1];

    // Copy from global to local.
    if(gIdx < d_to_scan_size) {
        s_to_scan[tIdx] = d_to_scan[gIdx];
    }
    __syncthreads();

    // Scan the shared block of data.
    bl_sweep_up(s_to_scan, blockDim.x);
    bl_sweep_down(s_to_scan, blockDim.x);

    // Now store the block reduction (sum of all elets)
    if(tIdx == (blockDim.x - 1)) {
        d_block_reductions[blockIdx.x] = s_to_scan[tIdx] + d_to_scan[gIdx];
    }
    __syncthreads();


    //Push the results back out to global memory.
    if(gIdx < d_to_scan_size) {
```

```
        d_scanned[gIdx] = s_to_scan[tIdx];
    }
}

/*! \brief Offset blocks
 *
 *  The kernel_block_offset takes in a sequence of values and performs an the corresponding block
 *  offset from the offset sequence.
 *
 *  @param[in, out] the sequence to offset.
 *  @param[in] the corresponding block offset values.
 */
__global__ void kernel_block_offset(int d_to_offset_size, int * d_to_offset, int * d_offsets) {
    int tIdx = threadIdx.x;
    int gIdx = threadIdx.x + blockIdx.x * blockDim.x;

    extern __shared__ int s_to_offset[];
    s_to_offset[tIdx] = 0;

    // Copy from global to shared.
    if(gIdx < d_to_offset_size) {
        s_to_offset[tIdx] = d_to_offset[gIdx];
    }
    __shared__ int offset;
    offset = d_offsets[blockIdx.x];
    __syncthreads();

    s_to_offset[tIdx] += offset;
    __syncthreads();

    // Copy back to global memory.
    if(gIdx < d_to_offset_size) {
        d_to_offset[gIdx] = s_to_offset[tIdx];
    }
}

/*! \brief CUDA Parallel Scan
 *
 * See cuda_parallel_scan(int, std::vector<int>, std::vector<int>, const bool debug)
 *
 */
```

```
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned) {
    return cuda_parallel_scan(block_size, to_scan, scanned, false);
}

/*! \brief CUDA Parallel Scan
 *
 *  The cuda_parallel_scan is a recursive function that takes in a large set of values and applies
 *  the scan operator across the entire set.
 *
 *  @param[in] The size of the block to use.
 *  @param[in] the vector of elements to scan. The length must ALWAYS be a multiple of block_size
 and
 *     to_scan.size() / block_size must never surpass 2147483648 on CUDA 3.0 devices.
 *  @param[in, out] a vector to store the scan results.
 *
 *  @returns the runtime of all the function calls involving the device.
 */
float cuda_parallel_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned,
 const bool debug) {
    // For timing
    cudaEvent_t start, stop;
    float cuda_elapsed_time_ms = 0.0;

    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_scan.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_parallel_scan\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX, to_scan.size(),
 blockDims.x);
    }

    // Vector initializations
    std::vector<int> block_reductions(gDmX, 0);
    std::vector<int> scanned_block_reductions(block_reductions.size(), 0);

    // create our pointers to the device memory (d prefix)
    int * d_to_scan = 0,
        * d_scanned = 0,
```

```
        * d_block_reductions = 0;

    // memory allocation sizes
    unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
                 scanned_mem_size = sizeof(int) * scanned.size(),
                 block_reductions_mem_size = sizeof(int) * block_reductions.size();

    // Initialize and begin the timers
    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&stop));
    checkCudaErrors(cudaEventRecord(start));

    // Allocate the memory on the device
    checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_block_reductions, block_reductions_mem_size));

    // Copy from the host to the device.
    checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_block_reductions, &block_reductions[0], block_reductions_mem_size,
                               cudaMemcpyHostToDevice));

    // Calculate the shared memory size.
    unsigned int shared_mem_size = sizeof(int) * blockDims.x;

    // Run a block level scan.
    kernel_block_scan<<<gridDims, blockDims, shared_mem_size>>>(to_scan.size(), d_to_scan,
d_scanned, d_block_reductions);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host.
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size, cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(&block_reductions[0], d_block_reductions, block_reductions_mem_size,
                               cudaMemcpyDeviceToHost));

    // Free GPU resources before recursing.
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
    checkCudaErrors(cudaFree(d_block_reductions));
```

```
    // If the size of block_reductions is larger than a single block then recurse and call
    //  cuda_parallel_scan on the block_reductions vector.
    // Else, pass the block_reductions vector to kernel_blelloch_scan.
    if(block_reductions.size() > (unsigned int) block_size) {
        cuda_parallel_scan(block_size, block_reductions, scanned_block_reductions, debug);
    } else {
        cuda_blelloch_scan(block_size, block_reductions, scanned_block_reductions, debug);
    }

    // Now perform the offsets to get a global scan.
    cuda_block_offset(block_size, scanned, scanned_block_reductions, debug);

    // Stop the timers
    checkCudaErrors(cudaEventRecord(stop));
    checkCudaErrors(cudaEventSynchronize(stop));
    checkCudaErrors(cudaEventElapsedTime(&cuda_elapsed_time_ms, start, stop));

    return cuda_elapsed_time_ms;
}

/*! \brief CUDA Blelloch scan.
 *
 * The cuda-blelloch-scan method will only perfrom a single block scan of a sequence using cuda.
 * For devices with a compute capability of 3.0 then the maximum block size is 1024.
 *
 *  @param[in] the vector of elements to scan. The size of the sequence must not surpass the
 *      maximum supported block size on the device.
 *  @param[in, out] a vector to store the scan results.
 */
void cuda_blelloch_scan(int block_size, std::vector<int> & to_scan, std::vector<int> & scanned,
 const bool debug) {
    int * d_to_scan = 0,
        * d_scanned = 0;

    unsigned int to_scan_mem_size = sizeof(int) * to_scan.size(),
                 scanned_mem_size = to_scan_mem_size;

    if(debug) {
        printf("---> Entering: cuda_blelloch_scan\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", 1, to_scan.size(),
 block_size);
```

```
    }

    // Allocate the memory on the devicee
    checkCudaErrors(cudaMalloc((void **)&d_to_scan, to_scan_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_scanned, scanned_mem_size));

    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_scan, &to_scan[0], to_scan_mem_size, cudaMemcpyHostToDevice));

    // Calculate the shared memory size
    unsigned int shared_mem_size = sizeof(int) * block_size;

    // Calculate the block and grid dimensions.
    dim3 gridDims(1, 1);
    dim3 blockDims(block_size, 1);

    // Run the blelloch scan on the block_reductions array.
    kernel_blelloch_scan<<<gridDims, blockDims, shared_mem_size>>>(to_scan.size(), d_to_scan,
 d_scanned);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&scanned[0], d_scanned, scanned_mem_size, cudaMemcpyDeviceToHost));

    // Clean up
    checkCudaErrors(cudaFree(d_to_scan));
    checkCudaErrors(cudaFree(d_scanned));
}

/*! \brief CUDA Parallel Offset
 *
 * Takes in a sequence and offsets the values per block. The offsets sequence contains the
 * per block offsets.
 *
 *  @param[in] the size of the block.
 *  @param[in, out] the sequence of elements to perform block level offsets.
 *  @param[in] the offset values.
 */
void cuda_block_offset(int block_size, std::vector<int> & to_offset, std::vector<int> offsets, const
 bool debug) {
```

```
    // Calculate the block and grid dimensions.
    int gDmX = (int) (to_offset.size() + block_size - 1) / block_size;
    dim3 gridDims(gDmX, 1);
    dim3 blockDims(block_size, 1);

    if(debug) {
        printf("---> Entering: cuda_block_offset\n");
        printf("\tGrid dims: %i\n\tSeq Size: %lu\n\tBlock Dims: %i\n", gDmX, to_offset.size(),
blockDims.x);
    }

    int * d_to_offset = 0,
        * d_offsets = 0;

    unsigned int to_offset_mem_size = sizeof(int) * to_offset.size(),
                 offsets_mem_size = sizeof(int) * offsets.size();

    // Allocate the memory
    checkCudaErrors(cudaMalloc((void **)&d_to_offset, to_offset_mem_size));
    checkCudaErrors(cudaMalloc((void **)&d_offsets, offsets_mem_size));

    // Copy from the host to the device
    checkCudaErrors(cudaMemcpy(d_to_offset, &to_offset[0], to_offset_mem_size,
cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_offsets, &offsets[0], offsets_mem_size, cudaMemcpyHostToDevice));

    // Calculate the shared memory size.
    unsigned int shared_mem_size = sizeof(int) * blockDims.x;

    // Execute the kernel
    kernel_block_offset<<<gridDims, blockDims, shared_mem_size>>>(to_offset.size(), d_to_offset,
d_offsets);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaThreadSynchronize());

    // Copy from the device to the host
    checkCudaErrors(cudaMemcpy(&to_offset[0], d_to_offset, to_offset_mem_size,
cudaMemcpyDeviceToHost));

    // Free unused memory
    checkCudaErrors(cudaFree(d_to_offset));
```

```
    checkCudaErrors(cudaFree(d_offsets));
}
```

scan_utils.h

```
#ifndef SCAN_UTILS_H__
#define SCAN_UTILS_H__

#include<vector>
#include<stdio.h>
#include<stdlib.h>

/*! \brief Sequential Sum Scan
 *
 *  @param[in] the vector to scan.
 *  @param[in, out] the vector to store the scanned results.
 *  @param[in] perform an inclusive or exclusive scan.
 */
inline void sequential_sum_scan(std::vector<int> & in, std::vector<int> & out, bool inclusive) {
    assert(in.size() == out.size());

    int sum = 0;
    for( std::vector<int>::iterator in_it = in.begin(), out_it = out.begin();
            (in_it != in.end()) && (out_it != in.end());
            ++in_it, ++out_it) {

        if( inclusive ) {
            sum += *in_it; //include the first element of in.
            *out_it = sum;
        } else {
            *out_it = sum; //do not include the first element of in.
            sum += *in_it;
        }
    }
}

/*! \brief Sequential Block Offset
 *
 *  Used for testing the block offset kernel.
 *
```

```
 *  \param[in] the block size to emulate.
 *  \param[in] the vector to offset.
 *  \param[in, out] the offsetted vector.
 *  \param[in] the offset to use per block.
 */
inline void sequential_block_offset(int block_size, std::vector<int> & to_offset, std::vector<int> &
 offsetted, std::vector<int> & offsets) {
    for(unsigned int i = 0; i < to_offset.size(); ++i) {
        int bIdx = i / block_size;

        offsetted[i] = to_offset[i] + offsets[bIdx];
    }
}

/*! \brief Sequential Block Scan
 *
 * Used for testing the block_scan_kernel.
 *
 * \param[in] The values to scan.
 * \param[in, out] The scanned values.
 * \param[in, out] The reductions of each block.
 *
 *__global__ void kernel_block_scan(int to_scan_size, int * d_to_scan, int * d_scanned, int *
 d_block_reductions);
 */
inline void sequential_block_scan(int block_size, std::vector<int> & to_scan, std::vector<int> &
 scanned, std::vector<int> & block_reductions) {
    int block_sum = 0;
    int bIdX = 0;

    scanned[0] = block_sum;
    block_sum += to_scan[0];

    for(unsigned int i = 1; i < to_scan.size(); ++i) {
        scanned[i] = block_sum;
        block_sum += to_scan[i];

        if(!((i + 1) % block_size)) {
            block_reductions[bIdX] = block_sum;
            block_sum = 0;
```

```
            bIdX++;
        }
    }
}

#endif
```

utils.h

```
/// The original source for this file is available from Udacity's github repository for
/// their "Intro to Parallel Computing" course.
/// url:
 https://raw.githubusercontent.com/udacity/cs344/master/Problem%20Sets/Problem%20Set%201/utils.h
/// Use this when programming!

#ifndef UTILS_H
#define UTILS_H

#include <iostream>
#include <iomanip>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <cassert>
#include <cmath>

#define checkCudaErrors(val) check( (val), #val, __FILE__, __LINE__)

// Pass all cuda api calls to this fuction. That will help you find mistakes quicker.
template<typename T>
void check(T err, const char* const func, const char* const file, const int line) {
  if (err != cudaSuccess) {
    std::cerr << "CUDA error at: " << file << ":" << line << std::endl;
    std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
    exit(1);
  }
}

// Swap two pointers.
inline void swap_pointers(int ** p0, int ** p1) {
    int * temp = *p0;
    *p0 = *p1;
```

```
    *p1 = temp;
}

// Fills a vector with random numbers.
inline void random_fill(std::vector<int> & to_fill, int max) {
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
        *it = (int)((float)rand() / RAND_MAX * max);
    }
}

// Fills a vector of size t from 0 ... t - 1.
inline void incremental_fill(std::vector<int> & to_fill) {
    int i = 0;
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
        *it = i;
        i++;
    }
}

// Dump the contents of a vector.
inline void print_vector(const std::vector<int> & to_print) {
    for(std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        printf("%i ", *it);
    }
    printf("\n");
}

// Performs a fisher yates shuffle on the array.
inline void shuffle(std::vector<int> & to_shuffle) {
    for( unsigned int i = 0; i < to_shuffle.size(); ++i) {
        unsigned int j = (unsigned int)((float) rand() / RAND_MAX * (i + 1));

        int tmp = to_shuffle[j];
        to_shuffle[j] = to_shuffle[i];
        to_shuffle[i] = tmp;
    }
}

// Prints the partitioning of a vector if it were to be passed to a CUDA kernel.
inline void print_vector_with_dims(const std::vector<int> & to_print,
                                   const int blockDimX, const int gridDimX) {
```

```
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        if(bDx_counter && !(bDx_counter % blockDimX)) {
            printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                printf("****");
                gDx_counter++;
            }
            printf("\n");
        }
        printf("%i ", *it);

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Print two vectors side by side.
inline void print_vector_comparison(const std::vector<int> & v1,
                                    const std::vector<int> & v2,
                                    const int start_index, const int stop_index,
                                    const int point_at, const int blockDimX,
                                    const int gridDimX) {
    int index = 0;
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end()) && (index < stop_index);
         ++it1, ++it2, ++index) {

        if(bDx_counter && !(bDx_counter % blockDimX)) {
            if(index >= start_index) printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                if(index >= start_index) printf("****");
                gDx_counter++;
            }
            if(index >= start_index) printf("\n");
        }
```

```
        if(index == point_at) {
            if(index >= start_index) printf("\n|->(%i, %i)<-|\n", *it1, *it2);
        } else {
            if(index >= start_index) printf("(%i, %i) ", *it1, *it2);
        }

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Compare to vectors. If equal return -1 else return index.
inline int equal(const std::vector<int> & v1, const std::vector<int> & v2) {
    int index = 0;
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end());
         ++it1, ++it2, ++index) {

        if(*it1 != *it2) {
            return index;
        }
    }

    return -1;
}


inline void reset_cuda_devs() {
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    while( dev_count --> 0 )
    {
        printf("Resetting device %i", dev_count);
        cudaSetDevice(dev_count);
        cudaDeviceReset();
    }
    printf("\n");
}
```

```
#endif
```

README.md

```
# CUDA Scanning

## Description

This homework will give you the option to either implement the Hillis & Steele or Blelloch scan
 algorithms.

## Instructions

  - Add the necessary code to the ‘cuda_parallel_scan‘ function and call the desired scanning
 kernel.
  - Implement either the ‘kernel_blelloch_scan‘ and ‘hs_scan_kernel‘.
  - To build the program run:
    - ‘make clean && make debug‘ to build the binary with debugging symbols.
    - ‘make clean && make‘ to build the binary.

## Learning Goals

You will learn how to perform a parallel scan across a sequence of values. In conjunction you will
 be able to implement a kernel allocates dynamic shared memory, and device only functions.

## Grading

This homework is worth 100 points.

  - Implement ‘cuda_parallel_scan‘.
    - Allocate and Free memory on the device. +16
    - Copy memory to and from the device. +16
    - Implement a kernel call that specifies the amount of dynamic shared memory to allocate. +16
  - Implement either the ‘kernel_blelloch_scan‘ or ‘hs_scan_kernel‘ global  functions.
    - Allocate the dynamic shared memory. +16
    - Implement the desired scanning algorithm. +16
    - Implement the necessary device functions. +16
  - EXTRA CREDIT: Implement a multi-block scan. +25
```

9.3.5 CUDA Reducing

calloc.c

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

void random_fill(unsigned int * to_fill, unsigned int len) {
    srand(time(NULL));
    for( unsigned int i = 0; i < len; i++) {
        to_fill[i] = (unsigned int)((float)rand() / RAND_MAX * 100);
    }
}


#define SEQ_SIZE 2048
int main(void) {
    printf("Sequence Size: %i\n", SEQ_SIZE);
    unsigned int * to_reduce = 0;
    to_reduce = calloc(SEQ_SIZE, sizeof(unsigned int) * SEQ_SIZE);

    random_fill(to_reduce, SEQ_SIZE);

    for(int i = 0; i < SEQ_SIZE; i++) {
        printf("%u ", to_reduce[i]);
    }
    printf("\n");

    free((void *)to_reduce);

    return 0;
}
```

complex.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
//#include<cuda_runtime.h>

//#define SEQ_SIZE ((1 << 27) + 12)
#define SEQ_SIZE ((1 << 10) + 1023 )
#define BLOCK_SIZE 1024
```

```
__global__ void cuda_sum_reduce(unsigned int * to_reduce, unsigned int * reduced, const unsigned int
 seq_len) {

    // Add zero padding to the end of 'to_reduce'
    unsigned int t_i = threadIdx.x + blockIdx.x * blockDim.x;
    if(t_i >= seq_len) {
        printf("Adding padding at %i\n", t_i);
        to_reduce[t_i] = 0;
    }
    __syncthreads();

    if(blockIdx.x == 0 && threadIdx.x == 0) {
        printf("Commencing the reduction\n");
        printf("Block Dim: %u\n", blockDim.x);
        printf("Grid Dim: %u\n", gridDim.x);
    }

    __shared__ unsigned int partial_sum[1024];

    partial_sum[threadIdx.x] = to_reduce[blockIdx.x * blockDim.x + threadIdx.x];
    __syncthreads();

    // Now lets reduce
    unsigned int t = threadIdx.x;
    for( unsigned int stride = blockDim.x >> 1; stride > 0; stride >>= 1 ) {
        __syncthreads();
        if ( t < stride ) {
            partial_sum[t] += partial_sum[t + stride];
        }
    }

    // Send the block sum back to global memory.
    __syncthreads();
    if(threadIdx.x == 0) {
        printf("Setting index %i to %u \n", blockIdx.x, partial_sum[0]);
        reduced[blockIdx.x] = partial_sum[0];
    }
}

unsigned int cuda_sum_reduce_launcher(unsigned int * to_reduce, unsigned int seq_len, float
 &cuda_ms) {
```

```
// Determine the number of blocks to use to buffer the problem
int blocks = seq_len / BLOCK_SIZE;
int remain = seq_len - blocks * BLOCK_SIZE;
blocks = (remain > 0) ? blocks + 1 : blocks;

unsigned int * reduced = (unsigned int*)calloc(blocks, sizeof(unsigned int));

printf("%i blocks of %i ints allocated\n", blocks, BLOCK_SIZE);
printf("Unsigned int size is %i bytes\n", sizeof(unsigned int));

unsigned int * d_to_reduce = 0;
unsigned int * d_reduced = 0;
cudaMalloc((void **)&d_to_reduce, sizeof(unsigned int) * blocks * BLOCK_SIZE);
cudaMalloc((void **)&d_reduced, sizeof(unsigned int) * blocks);
cudaMemcpy(d_to_reduce, to_reduce, sizeof(unsigned int) * seq_len, cudaMemcpyHostToDevice);
cudaMemcpy(d_reduced, reduced, sizeof(unsigned int) * blocks, cudaMemcpyHostToDevice);

// Setup the block and grid dimensions.
dim3 dimBlock(BLOCK_SIZE, 1);
dim3 dimGrid(blocks, 1);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// launch a kernel with a single thread to greet from the device
cudaEventRecord(start);
cuda_sum_reduce<<<dimGrid, dimBlock>>>(d_to_reduce, d_reduced, seq_len);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
cudaEventElapsedTime(&cuda_ms, start, stop);

//Copy answer and dealloc
cudaMemcpy(to_reduce, d_to_reduce, sizeof(unsigned int) * seq_len, cudaMemcpyDeviceToHost);
cudaMemcpy(reduced, d_reduced, sizeof(unsigned int) * blocks, cudaMemcpyDeviceToHost);
cudaFree(d_to_reduce);
cudaFree(d_reduced);

free(reduced);
```

```
    return reduced[0];
}

unsigned int sequential_sum_reduce(unsigned int * to_reduce, unsigned int seq_len) {
    unsigned int reduction = 0;
    for( unsigned int i = 0; i < seq_len; ++i) {
        reduction += to_reduce[i];
    }

    return reduction;
}

void random_fill(unsigned int * to_fill, unsigned int len) {
    for( unsigned int i = 0; i < len; i++) {
        to_fill[i] = (unsigned int)((float)rand() / RAND_MAX * 100);
    }
}

void reset_cuda_devs() {
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    while( dev_count --> 0 )
    {
        printf("Resetting device %i", dev_count);
        cudaSetDevice(dev_count);
        cudaDeviceReset();
    }
    printf("\n");
}

int main(void) {
    srand(time(NULL));
    reset_cuda_devs();

    float cuda_ms = 0.0;
    printf("Sequence Size: %i\n", SEQ_SIZE);
    unsigned int * to_reduce = 0;
    to_reduce = (unsigned int *) calloc(SEQ_SIZE, sizeof(unsigned int));

    random_fill(to_reduce, SEQ_SIZE);
```

```
    printf("SEQU Reduce: %u\n", sequential_sum_reduce(to_reduce, SEQ_SIZE));
    printf("CUDA Reduce: %u\n", cuda_sum_reduce_launcher(to_reduce, SEQ_SIZE, cuda_ms));

    free(to_reduce);

    return 0;
}
```

main.cu

```
///////////////////////////////////////////////////////////////////////////////////////////////
//
// This reduction example was influenced by Programming Massively Parallel Processors page 102
// and Udacity’s Introduction to Parallel Programming at https://github.com/udacity/cs344
// refer to lecture 3 materials in the repository.
//
///////////////////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>
#include <vector>

#include "utils.h"

#define SEQ_SIZE 1024
#define BLOCK_SIZE 1024

/// Function prototypes
__global__ void cuda_sum_reduce(int * sequence_to_reduce, int * reduced);
int cuda_sum_reduce_launcher(std::vector<int> & sequence_to_reduce);
int sequential_sum_reduce(std::vector<int> & sequence_to_reduce);



/// Main
int main(void) {
    srand(0 /*time(NULL)*/); // call srand only once.
    reset_cuda_devs();
```

```
    std::vector<int> sequence_to_reduce(SEQ_SIZE, 0);
    random_fill(sequence_to_reduce, 100);

    printf("Sequential Reduce: %d\n", sequential_sum_reduce(sequence_to_reduce));
    printf("GPGPU Reduce:      %d\n", cuda_sum_reduce_launcher(sequence_to_reduce));

    return 0;
}

/// \brief Reduction kernel.
///
/// Implementation of the reduction algorithm (Figure 6.4) on page 102 in
/// Programming Massively Parallel Processors. Takes in a sequence of values
/// and reduces them using a binary operator. The operator for this function
/// is the addition operator. This function will only perform a reduction
/// across a block.
///
/// \param Sequence of values to reduce.
/// \param The calculated reduction.
__global__ void cuda_sum_reduce(int * sequence_to_reduce, int * reduced) {
    __shared__ unsigned int partial_sum[BLOCK_SIZE];

    partial_sum[threadIdx.x] = sequence_to_reduce[threadIdx.x];
    __syncthreads();

    // Now lets reduce
    // Page 102 of Programming Massively Parallel Processors
    unsigned int t = threadIdx.x;
    for( unsigned int stride = blockDim.x >> 1; stride > 0; stride >>= 1 ) {
        __syncthreads();
        if ( t < stride ) {
            partial_sum[t] += partial_sum[t + stride];
        }
    }

    // Send the block sum back to global memory.
    __syncthreads();
    if(threadIdx.x == 0) {
        * reduced = partial_sum[0];
    }
}
```

```
/// \brief Reduces Sequence using CUDA
///
/// Allocates memory, sets up the block and grid dimensions, and executes the cuda_sum_reduce
 kernel.
///
/// \param The sequence to reduce.
///
/// \param The reduction.
int cuda_sum_reduce_launcher(std::vector<int> & sequence_to_reduce) {
    // Pointers to memory locations on the device
    int * d_sequence_to_reduce = 0;
    int * d_reduced = 0;
    int reduced = 0;

    cudaMalloc((void **)&d_sequence_to_reduce, sizeof(int) * sequence_to_reduce.size());
    cudaMalloc((void **)&d_reduced, sizeof(int));

    cudaMemcpy(d_sequence_to_reduce, &sequence_to_reduce[0], sizeof(int) *
 sequence_to_reduce.size(), cudaMemcpyHostToDevice);

    // Setup the block and grid dimensions.
    dim3 dimBlock(BLOCK_SIZE, 1);
    dim3 dimGrid(1, 1);

    cuda_sum_reduce<<<dimGrid, dimBlock>>>(d_sequence_to_reduce, d_reduced);
    cudaThreadSynchronize();

    ////Copy answer and dealloc
    cudaMemcpy(&reduced, d_reduced, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_sequence_to_reduce);
    cudaFree(d_reduced);

    return reduced;
}

/// \brief Serial reduction algorithm.
int sequential_sum_reduce(std::vector<int> & sequence_to_reduce) {
    int reduction = 0;
    for( std::vector<int>::iterator it = sequence_to_reduce.begin(); it != sequence_to_reduce.end();
 ++it) {
```

```
        reduction += *it;
    }

    return reduction;
}
```

utils.h

```
/// The original source for this file is available from Udacity's github repository for
/// their "Intro to Parallel Computing" course.
/// url:
 https://raw.githubusercontent.com/udacity/cs344/master/Problem%20Sets/Problem%20Set%201/utils.h
/// Use this when programming!

#ifndef UTILS_H
#define UTILS_H

#include <iostream>
#include <iomanip>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <cassert>
#include <cmath>

#define checkCudaErrors(val) check( (val), #val, __FILE__, __LINE__)

// Pass all cuda api calls to this fuction. That will help you find mistakes quicker.
template<typename T>
void check(T err, const char* const func, const char* const file, const int line) {
  if (err != cudaSuccess) {
    std::cerr << "CUDA error at: " << file << ":" << line << std::endl;
    std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
    exit(1);
  }
}

// Swap two pointers.
inline void swap_pointers(int ** p0, int ** p1) {
    int * temp = *p0;
```

```
    *p0 = *p1;
    *p1 = temp;
}

// Fills a vector with random numbers.
inline void random_fill(std::vector<int> & to_fill, int max) {
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
        *it = (int)((float)rand() / RAND_MAX * max);
    }
}

// Fills a vector of size t from 0 ... t - 1.
inline void incremental_fill(std::vector<int> & to_fill) {
    int i = 0;
    for( std::vector<int>::iterator it = to_fill.begin(); it != to_fill.end(); ++it) {
        *it = i;
        i++;
    }
}

// Dump the contents of a vector.
inline void print_vector(const std::vector<int> & to_print) {
    for(std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        printf("%i ", *it);
    }
    printf("\n");
}

// Performs a fisher yates shuffle on the array.
inline void shuffle(std::vector<int> & to_shuffle) {
    for( unsigned int i = 0; i < to_shuffle.size(); ++i) {
        unsigned int j = (unsigned int)((float) rand() / RAND_MAX * (i + 1));

        int tmp = to_shuffle[j];
        to_shuffle[j] = to_shuffle[i];
        to_shuffle[i] = tmp;
    }
}

// Prints the partitioning of a vector if it were to be passed to a CUDA kernel.
inline void print_vector_with_dims(const std::vector<int> & to_print,
```

```
                                   const int blockDimX, const int gridDimX) {
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it = to_print.begin(); it != to_print.end(); ++it) {
        if(bDx_counter && !(bDx_counter % blockDimX)) {
            printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                printf("****");
                gDx_counter++;
            }
            printf("\n");
        }
        printf("%i ", *it);

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Print two vectors side by side.
inline void print_vector_comparison(const std::vector<int> & v1,
                                    const std::vector<int> & v2,
                                    const int start_index, const int stop_index,
                                    const int point_at, const int blockDimX,
                                    const int gridDimX) {
    int index = 0;
    int bDx_counter = 0;
    int gDx_counter = 0;
    printf("\n---------------\n");
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end()) && (index < stop_index);
         ++it1, ++it2, ++index) {

        if(bDx_counter && !(bDx_counter % blockDimX)) {
            if(index >= start_index) printf("\n");
            if(gDx_counter && !(gDx_counter % gridDimX)) {
                if(index >= start_index) printf("****");
                gDx_counter++;
            }
            if(index >= start_index) printf("\n");
```

```
        }

        if(index == point_at) {
            if(index >= start_index) printf("\n|->(%i, %i)<-|\n", *it1, *it2);
        } else {
            if(index >= start_index) printf("(%i, %i) ", *it1, *it2);
        }

        bDx_counter++;
    }
    printf("\n---------------\n");
}

// Compare to vectors. If equal return -1 else return index.
inline int equal(const std::vector<int> & v1, const std::vector<int> & v2) {
    int index = 0;
    for( std::vector<int>::const_iterator it1 = v1.begin(), it2 = v2.begin();
         (it1 != v1.end()) && (it2 != v2.end());
         ++it1, ++it2, ++index) {

        if(*it1 != *it2) {
            return index;
        }
    }

    return -1;
}


inline void reset_cuda_devs() {
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    while( dev_count --> 0 )
    {
        printf("Resetting device %i", dev_count);
        cudaSetDevice(dev_count);
        cudaDeviceReset();
    }
    printf("\n");
}
```

```
#endif
```

README.md

```
# CUDA Reducing

## Description

Implement the parallel reduction algorithm explained in page 102 from Programming Massively Parallel
 Processors.

## Directions

  - Complete the components within `main.cu`.
    - `cuda_sum_reduce_launcher` will need allocate and copy memory, setup the grid and block
 dimensions, and execute the kernel.
    - Implement  the missing components within `cuda_sum_reduce` such that it can sum the elements
 of a sequence in parallel and store the result back into global memory.
  - To compile the program run `make clean && make`

## Learning Goals

After completing this assignment you will have learned how to perform a block-level reduction on a
 sequence of
values.

## Grading

The homework is worth a maximum of 100 points.
  - `cuda_sum_reduce_launcher` needs to be able to:
    - Allocate memory on the device. +12
    - Copy data to the device. +12
    - Setup the block and grid dimensions and execute the kernel. +12
    - Copy data from the device. +12
    - Free up the memory allocated on the device. +12
  - `cudo_sum_reduce` needs to be able to:
    - Allocate shared memory. +12
    - Execute the parallel reduction loop as described on page 102 in Programming Massively Parallel
 Processors. +12
    - Store the result back into global memory. +12
  - EXTRA CREDIT: Use dynamic shared memory. +5.
```

9.4 Quizzes

9.4.1 CUDA Quiz Questions

CUDA-Quizzes.gift

```
// question: 511117  name: Abbreviations
::Abbreviations::[html]<p>Match abbreviations to their correct definition.</p>{
    =<p>GPU</p> -> Graphics Processing Unit
    =<p>CPU</p> -> Central Processing Unit
    =<p>GPGPU</p> -> General-Purpose Processing Unit
    =<p>GLSL</p> -> OpenGL Shading Language
    =<p>CUDA</p> -> Compute Unified Device Architecture
    =<p>OpenGL</p> -> Open Graphics Library
    =<p>ALU</p> -> Arithmetic Logic Unit
    =<p>SM</p> -> Streaming Multiprocessor
    =<p>OpenCL</p> -> Open Compute Language
}


// question: 511277  name: Compilation
::Compilation::[html]<p>Match the terms to the numbers in the image.</p>\n<p><img
 src\="@@PLUGINFILE@@/Compilation_Procedure.png" alt\="Compilation Procedure Image" width\="800"
 height\="449" /></p>{
    =<p>1</p> -> Seperate CUDA and C/C++.
    =<p>2</p> -> Compile CUDA Code
    =<p>3</p> -> Compile C/C++ Code
    =<p>4</p> -> Link objects together.
}


// question: 511248  name: CUDA Thread Hierarchy
::CUDA Thread Hierarchy::[html]<p>Match the term that is the direct child the item.</p>{
    =<p>Thread</p> -> Kernel
    =<p>Block</p> -> Thread
    =<p>Grid</p> -> Block
    =<p>GPU</p> -> Grid
}
```

```
// question: 511112  name: Flynn’s Taxonomy
::Flynn’s Taxonomy::[html]<p>There are four main categories to Flynn’s taxonomy. Match the
 definitions to their corresponding abbreviations. </p>{
    =<p>SISD</p> -> Single Instruction, Single Data
    =<p>SIMD</p> -> Single Instruction, Multiple Data
    =<p>MISD</p> -> Multiple Instruction, Single Data
    =<p>MIMD</p> -> Multiple Instruction, Multiple Data
}


// question: 511202  name: Flynn’s Taxonomy Real Life
::Flynn’s Taxonomy Real Life::[html]<p>Use Flynn’s architecture to categorize each system.</p>{
    =<p>Parallel processors acting on independent subsets of a dataset.</p> -> SIMD
    =<p>Single processor working sequentially on a single dataset.</p> -> SISD
    =<p>Different processes working on the same dataset.</p> -> MISD
    =<p>Multiple processes working on multiple datasets.</p> -> MIMD
}


// question: 511275  name: Function Scope
::Function Scope::[html]<p>Match the qualifiers to their associated use.</p>{
    =<p><code> __device__ </code></p> -> Function that is only callable from the device.
    =<p><code> __global__ </code></p> -> Function that executes on the device but is callable from the
 host.
    =<p><code> __host__ </code></p> -> Function that only executes on the host.
}


// question: 511188  name: Graphics Pipeline
::Graphics Pipeline::[html]<p>What is the general order of the graphics pipeline? Select the
 appropriate term for each letter in the image.</p>\n<p><img src\="@@PLUGINFILE@@/Pipeline.png"
 alt\="Basic Graphics Pipeline" width\="800" height\="387" /></p>{
    =<p>A</p> -> Vertex Stage
    =<p>B</p> -> Primative Stage
    =<p>C</p> -> Raster Stage
    =<p>D</p> -> Shader Stage
}


// question: 511276  name: Memory Management
```

```
::Memory Management::[html]<p>Match the CUDA API calls their intended role.</p>{
    =<p><code> cudaMalloc </code></p> -> Allocate memory on the device.
    =<p><code> cudaMemcpy</code></p> -> Copy data.
    =<p><code> cudaFree</code></p> -> Clean up allocated memory.
}


// question: 511208  name: Amdahls Law Calculation 1
::Amdahls Law Calculation 1::[html]<p>You have a program with a parallel fraction of 0.9 that is
 executing on a machine with 10 processors. What is the speedup?</p>{
    =<p>5.26</p>
    ~<p>1</p>
    ~<p>0.5</p>
    ~<p>10</p>
    ~<p>7.26</p>
}


// question: 511209  name: Amdahls Law Calculation 2
::Amdahls Law Calculation 2::[html]<p>You have a program with a parallel fraction of 0.9 that is
 executing on a machine with 100 processors. What is the speedup?</p>{
    =<p>9.17</p>
    ~<p>5.26</p>
    ~<p>1</p>
    ~<p>0.01</p>
    ~<p>10</p>
}


// question: 511210  name: Amdahls Law Calculation 3
::Amdahls Law Calculation 3::[html]<p>You have a program with a parallel fraction of 0.5 that is
 executing on a machine with 10 processors. What is the speedup?</p>{
    ~<p>2.00</p>
    ~<p>1.98</p>
    =<p>1.82</p>
    ~<p>1.0</p>
    ~%50%<p>0.84</p>
}


// question: 511211  name: Amdahls Law Calculation 4
```

```
::Amdahls Law Calculation 4::[html]<p>You have a program with a parallel fraction of 0.5 that is
 executing on a machine with 100 processors. What is the speedup?</p>{
    ~%50%<p>2.00</p>
    =<p>1.98</p>
    ~<p>1.02</p>
    ~<p>0.98</p>
    ~<p>0.02</p>
}


// question: 511198  name: Bit-level Parallesim
::Bit-level Parallesim::[html]<p>Select the appropriate definition of bit-level parallelism.</p>{
    =<p>Increase the word size a processor can handle to reduce the number of passes when adding two
 values.</p>
    ~<p>If a word has 32 bits then create 32 threads to handle each bit.</p>
    ~<p>If a word is several bytes in size then farm out each byte to each core.</p>
    ~<p>A mathematical operator.</p>
}


// question: 511219  name: Classic Hardware Parallelism
::Classic Hardware Parallelism::[html]<p>What are the classic implementations of hardware
 parallelism using CPUs only?</p>{
    ~<p>Multi-core CPUs</p>
    ~<p>Multi-socket motherboards</p>
    ~<p>Clustered Hardware</p>
    =<p>All of the above.</p>
}


// question: 511245  name: CUDA SM Threads Execution
::CUDA SM Threads Execution::[html]<p>An SM can processes several hundred to thousands of threads.
 In order to execute all of the threads. The SM will execute batches of threads concurrently until
 all of the threads have been processed. What is the term used by CUDA to describe the execution of
 a batch of threads?</p>{
    =<p>Warp</p>
    ~<p>Cycle</p>
    ~<p>Jump</p>
    ~<p>Orbit</p>
    ~<p>Revolution</p>
}
```

```
// question: 511258  name: CUDA Threads Properties
::CUDA Threads Properties::[html]<p>Select all that is true about threads.</p>{
    ~%25%<p>Each thread has a unique ID.</p>
    ~<p>Threads do not have IDs.</p>
    ~%25%<p>Threads belong to blocks.</p>
    ~<p>Threads do not belong to blocks.</p>
    ~<p>Threads belong to grids.</p>#<p>A thread is not a direct child of a grid.</p>
    ~%25%<p>The implementation of a thread is called a kernel.</p>
    ~%25%<p>All threads executing concurrently share the same code.</p>
}


// question: 511185  name: Early GPU Adopters
::Early GPU Adopters::[html]<p>What were the early adapters of graphics processing hardware?</p>{
    ~%25%<p>Gaming Consoles</p>
    ~%25%<p>Workstations</p>
    ~%25%<p>Desktops</p>
    ~<p>Cellphones</p>
    ~%25%<p>Mainframes</p>
    ~<p>Vehicles</p>
}


// question: 511217  name: Embarrassingly Parallel Problem Properties
::Embarrassingly Parallel Problem Properties::[html]<p>What are the properties of a problem that is
 embarrassingly parallel?</p>{
    =<p>Can be subdivided into smaller identical independent subproblems.</p>
    ~<p>A problem that consists of a large set independent subproblems.</p>
    ~<p>A problem where each subproblem is dependent upon other subproblems.</p>
    ~<p>An indivisible problem.</p>
}


// question: 511187  name: Fixed Pipeline Properties
::Fixed Pipeline Properties::[html]<p>What are the properties of a fixed pipeline on a GPU?</p>{
    ~%50%<p>Pipeline that has discreet components suited for the different steps in processing vertex
 and raster data.</p>
    ~%50%<p>The pipeline is configurable.</p>
    ~<p>The pipeline is programmable.</p>
```

```
    ~<p>Dynamically allocates resources depending on the nature of the graphics data.</p>
}


// question: 511213  name: Gustafsons Law Calculation 1
::Gustafsons Law Calculation 1::[html]<p>You have a program with a parallel fraction of 0.5 that
 is executing on a machine with 10 processors. What is the speedup?</p>{
    ~<p>10</p>
    ~<p>9.5</p>
    ~<p>7.5</p>
    ~<p>8.5</p>
    ~<p>6.5</p>
    =<p>5.5</p>
    ~<p>4.5</p>
    ~<p>3.5</p>
}


// question: 511214  name: Gustafsons Law Calculation 2
::Gustafsons Law Calculation 2::[html]<p>You have a program with a parallel fraction of 0.5 that
 is executing on a machine with 100 processors. What is the speedup?</p>{
    ~<p>70.5</p>
    ~<p>65.5</p>
    =<p>50.5</p>
    ~<p>45.5</p>
    ~<p>30.5</p>
    ~<p>25.5</p>
    ~<p>10.5</p>
    ~<p>5.5</p>
}


// question: 511215  name: Gustafsons Law Calculation 3
::Gustafsons Law Calculation 3::[html]<p>You have a program with a parallel fraction of 0.9 that
 is executing on a machine with 10 processors. What is the speedup?</p>{
    ~<p>2.0</p>
    ~<p>1.0</p>
    =<p>1.9</p>
    ~<p>2.9</p>
    ~<p>0.9</p>
}
```

```
// question: 511216  name: Gustafsons Law Calculation 4
::Gustafsons Law Calculation 4::[html]<p>You have a program with a parallel fraction of 0.9 that
 is executing on a machine with 100 processors. What is the speedup?</p>{
    ~<p>11.9</p>
    =<p>10.9</p>
    ~<p>9.9</p>
    ~<p>8.9</p>
    ~<p>7.9</p>
}


// question: 511189  name: Programmable Pipeline Properties
::Programmable Pipeline Properties::[html]<p>Select the properties associated to a programmable
 pipeline.</p>{
    ~%33.33333%<p>Customization of individual steps in the pipeline.</p>
    ~%33.33333%<p>Programmable vertex processor.</p>
    ~%33.33333%<p>Programmable fragment processor</p>
    ~<p>Access to image data on the host machine.</p>
    ~<p>Networking capabilities.</p>
}


// question: 511222  name: Properties of the CPU
::Properties of the CPU::[html]<p>What are some properties of a CPU bound parallelism?</p>{
    ~%14.28571%<p>Low latency</p>
    ~<p>High latency</p>
    ~%14.28571%<p>Low throughput</p>
    ~<p>High throughput</p>
    ~%14.28571%<p>Support for interrupts</p>
    ~<p>No support for interrupts</p>
    ~%14.28571%<p>Branch prediction</p>
    ~%14.28571%<p>Out-of-order execution</p>
    ~%14.28571%<p>Speculative execution</p>
    ~%14.28571%<p>Asynchronous events</p>
}


// question: 511229  name: Properties of the GPGPU
::Properties of the GPGPU::[html]<p>What are some properties of a GPGPU bound parallelism?</p>{
```

```
    ~<p>Low latency</p>
    ~%33.33333%<p>High latency</p>
    ~<p>Low throughput</p>
    ~%33.33333%<p>High throughput</p>
    ~<p>Support for interrupts</p>
    ~%33.33333%<p>No support for interrupts</p>
    ~<p>Branch prediction</p>
    ~<p>Out-of-order execution</p>
    ~<p>Speculative execution</p>
    ~<p>Asynchronous events</p>
}


// question: 511114  name: Raster Graphics
::Raster Graphics::[html]<p>What are raster graphics?</p>{
    ~<p>Graphics displayed using a series of lines represented by mathematical formulas.</p>
    =<p>Graphics data represented as a grid of data points.</p>
    ~<p>Graphics visualized as a set of audio sounds.</p>
    ~<p>Graphics visualized as a set of textures that can be felt.</p>
}


// question: 511228  name: Structure of GPGPU
::Structure of GPGPU::[html]<p>Select all that are true about the architecture of a GPGPU.</p>{
    ~%25%<p>Single die with hundreds of massively parallel processors.</p>
    ~%25%<p>Each processor is simpler than a standard CPU core.</p>
    ~%25%<p>Dedicated memory.</p>
    ~%25%<p>Can execute thousands of concurrent threads.</p>
    ~<p>Couple of simple processors running concurrently.</p>
    ~<p>No dedicated memory.</p>
}


// question: 511190  name: Unified Pipeline Properties
::Unified Pipeline Properties::[html]<p>Select the terms that best describe a unified pipeline.</p>{
    ~%33.33333%<p>Array of generalized processors.</p>
    ~%33.33333%<p>Perform multiple passes on the data before sending it to the frame buffer.</p>
    ~%33.33333%<p>load balancing</p>
    ~<p>Direct communication with the north bridge on the motherboard.</p>
    ~<p>None of the above.</p>
}
```

```
// question: 511113  name: Vector Graphics
::Vector Graphics::[html]<p>What are vector graphics?</p>{
	=<p>Graphics displayed using a series of lines represented by mathematical formulas.</p>
	~<p>Graphics data represented as a grid of data points.</p>
	~<p>Graphics visualized as a set of audio sounds.</p>
	~<p>Graphics visualized as a set of textures that can be felt.</p>
}


// question: 511170  name: What is Flynn's Taxonomy
::What is Flynn's Taxonomy::[html]<p>What is Flynn's Taxonomy?</p>{
	~%50%<p>A classification of computer architectures.</p>
	~%50%<p>A classification of different concurrent programming paradigms.</p>
	~<p>A processor instruction set.</p>
	~<p>A database.</p>
}


// question: 511205  name: Amdahls Law Dataset Assumption
::Amdahls Law Dataset Assumption::[html]<p>Amdahl's law assumes a dataset with a fixed
 size.</p>{TRUE}


// question: 511224  name: CPU General Operations
::CPU General Operations::[html]<p>The CPU has been optimized to perform general system
 operations.</p>{TRUE}


// question: 511225  name: CPU Multiple Threads
::CPU Multiple Threads::[html]<p>It isinefficient to run thousands of threads concurrently on a
 CPU with 2 to 4 cores.</p>{TRUE}


// question: 511227  name: CPU Shared Memory
::CPU Shared Memory::[html]<p>The RAM that the CPU is also shared amongst other system
 peripherals.</p>{TRUE}


// question: 511269  name: CUDA Blocks
```

```
::CUDA Blocks::[html]<p>Blocks are independent of other blocks.</p>{TRUE}


// question: 511243  name: CUDA SM & Threads Question
::CUDA SM & Threads Question::[html]<p>Relative to the CUDA architecture. Even though an SM can
 handle hundreds to thousands of threads. It can only execute a subset of those threads
 concurrently.</p>{TRUE}


// question: 511271  name: CUDA SM Blocks
::CUDA SM Blocks::[html]<p>A streaming multiprocessor is capable of dealing with multiple
 blocks.</p>{TRUE}


// question: 511252  name: CUDA Thread ID
::CUDA Thread ID::[html]<p>Each thread has an ID associated with it.</p>{TRUE}


// question: 511261  name: CUDA Thread ID
::CUDA Thread ID::[html]<p>Every thread and block has at least an xz-coordinate pair. These are
 accessible via <code> threadIdx </code> and <code> blockIdx</code>. </p>{TRUE}


// question: 511249  name: CUDA Threads
::CUDA Threads::[html]<p>A thread is the smallest unit that is executed by the GPGPU.</p>{TRUE}


// question: 511196  name: Double Precision Speed
::Double Precision Speed::[html]<p>Performing arithmetic using double precision values is slower
 than single precision arithmetic.</p>{TRUE}


// question: 511115  name: Fixed-Pipeline
::Fixed-Pipeline::[html]<p>A graphics processing unit with a fixed-pipeline is configurable but not
 customizable.</p>{TRUE}


// question: 511230  name: GPGPU High Throughput
::GPGPU High Throughput::[html]<p>At a the sacrifice of latency, the GPGPU is optimized for high
 throughput.</p>{TRUE}
```

```
// question: 511231  name: GPGPU Taking Orders
::GPGPU Taking Orders::[html]<p>A GPGPU requires a CPU to supply it with data and
 instructions.</p>{TRUE}
```

```
// question: 511206  name: Gustafsons Law Dataset Assumption
::Gustafsons Law Dataset Assumption::[html]<p>Gustafsons Law assumes that as more processors
 are added then the size of the dataset will also be increased in order to fully utilize every
 processor. </p>{TRUE}
```

```
// question: 511116  name: Programmable Pipeline
::Programmable Pipeline::[html]<p>A graphics processing unit with a programmable pipeline allows the
 customization of certain steps in the pipeline via special languages such as GLSL.</p>{TRUE}
```

```
// question: 511199  name: Systolic Array Property
::Systolic Array Property::[html]<p>A systolic architecture is data-stream-driven by data counters
 while a Von-Neumann architecture is an instruction-steam-driven program counter.</p>{TRUE}
```

```
// question: 511184  name: Systolic Arrays
::Systolic Arrays::[html]<p>Were systolic arrays designed for the cryptanalysis of the
 enigma?</p>{TRUE}
```

```
// question: 511194  name: Unified Pipeline Academic Research
::Unified Pipeline Academic Research::[html]<p>Academic researchers started using the shading
 languages to perform general computations. As a result CUDA and OpenCL were created to address
 their needs.</p>{TRUE}
```

```
// question: 511186  name: Vector Graphics Memory
::Vector Graphics Memory::[html]<p>Representing graphics in a vector format requires less memory
 than that of raster graphics.</p>{TRUE}
```

9.4.2 Concurrency Quiz Questions

Concurrency-Quizzes.gift

```
// question: 784515  name: Four Conditions for a Deadlock
::Four Conditions for a Deadlock::[html]<p>Match each description with its respective term.</p>{
    =<p>There exists at least a single resource that can be held in a non-shareable mode.</p> -> Mutual
 Exclusion
    =<p>Thread holds onto a resource and waits for another resource to be freed.</p> -> Hold and Wait
    =<p>Threads cannot be stripped of their resources by other threads.</p> -> No Preemption
    =<p>When two or more threads are holding and waiting on shared resources.</p> -> Circular Wait
}


// question: 784514  name: Resource-Allocation Graph Key
::Resource-Allocation Graph Key::[html]<p>Match each number in the image with the relevant
 item.</p>\n<p><img src\="@@PLUGINFILE@@/resource-allocation-graph-key.png" alt\="" width\="282"
 height\="611" /></p>{
    =<p>1</p> -> Process
    =<p>2</p> -> Resource with Instances
    =<p>3</p> -> P holds instance R
    =<p>4</p> -> P requests instance R
}


// question: 784369  name: Threading Models
::Threading Models::[html]<p>Match the numbers in the image with the appropriate threading
 model.</p>\n<p><img style\="vertical-align\: middle;" src\="@@PLUGINFILE@@/ThreadingModels.png"
 alt\="Threading Models" width\="1057" height\="400" /></p>\n<p></p>{
    =<p>2</p> -> One-to-One
    =<p>1</p> -> Many-to-One
    =<p>3</p> -> Many-to-Many
}


// question: 784376  name: Critical Section Problem Solution
::Critical Section Problem Solution::[html]<p>When faced with a critical section what are the three
 basic requirements of the solution?</p>{
    ~%33.33333%<p>Mutual Exclusion</p>
    ~%33.33333%<p>Progress</p>
    ~%33.33333%<p>Bounded Waiting</p>
    ~<p>Process Synchronization</p>
    ~<p>Thread Coordination</p>
}
```

```
// question: 784373  name: Detaching a Thread.
::Detaching a Thread.::[html]<p>What does detaching a thread do?</p>{
    =<p>Allows the thread to run independently of the program until it terminates or the program exits
 and it is cleaned up by the operating system.</p>
    ~<p>Blocks the calling thread until the target thread finishes executing.</p>
    ~<p>Joins the instructions of the calling and target threads.</p>
    ~<p>Tells the target thread to terminate it's execution instantly.</p>
}


// question: 784368  name: Early Multithreading & Multitasking
::Early Multithreading & Multitasking::[html]<p>How did Berkeley Timesharing System handle
 multiprocessing?</p>{
    =<p>Gave processes<strong>time-slots</strong>of execution.</p>
    ~<p>Had a multi-core processor where each core was dedicated to a single process.</p>
    ~<p>The system didn't support multitasking.</p>
    ~<p>The system permitted multiple batches of work to be queued up for processing.</p>
}


// question: 784370  name: Hyper-Threading
::Hyper-Threading::[html]<p>What are the properties of a single-core processor with
 Hyper-Threading?</p>{
    =<p>Duplicate Registers</p>
    ~<p>Duplicate Caches (to match the number of duplicated registers).</p>
    ~<p>128bit bus interface.</p>
    ~<p>Larger L1 cache.</p>
}


// question: 784372  name: Joining a thread
::Joining a thread::[html]<p>What does joining a thread do?</p>{
    ~<p>Allows the thread to run independently of the program until it terminates or the program exits
 and is cleaned up by the operating system.</p>
    =<p>Blocks the calling thread until the target thread finishes executing.</p>
    ~<p>Joins the instructions of the calling and target threads.</p>
    ~<p>Tells the target thread to terminate it's execution instantly.</p>
}
```

```
// question: 784517  name: Lock-Free Programming Guarantees
::Lock-Free Programming Guarantees::[html]<p>A lock-free programming guarantees,..</p>{
    ~<p>No thread will ever starve when accessing the critical section.</p>
    ~%50%<p>Throughput will always be guaranteed.</p>
    ~<p>Faster performance.</p>
    ~%50%<p>Ensures deadlocks will never occur.</p>
}


// question: 784375  name: Preventing race conditions.
::Preventing race conditions.::[html]<p>What can be used to manage threads such that race conditions
 are prevented?</p>{
    ~%33.33333%<p>atomic operations</p>
    ~%33.33333%<p>mutexes</p>
    ~%33.33333%<p>semaphores</p>
    ~<p>memcpy</p>
    ~<p>putting the thread to sleep periodically.</p>
}


// question: 784510  name: Properties of a mutex?
::Properties of a mutex?::[html]<p>What are the properties of a mutex that is shared amongst a group
 of threads?</p>{
    ~%50%<p>Allows one thread at a time into a critical section.</p>
    ~%50%<p>Can be used to lock and unlock a critical section.</p>
    ~<p>Allows multiple threads to access a critical section.</p>
    ~<p>Mutexes cannot control thread access to a critical section.</p>
}


// question: 784511  name: Properties of a semaphore?
::Properties of a semaphore?::[html]<p>What are the properties of a semaphore that is shared amongst
 a group of threads?</p>{
    ~%33.33333%<p>Allows one thread at a time into a critical section.</p>
    ~%33.33333%<p>Can be used to lock and unlock a critical section.</p>
    ~%33.33333%<p>Allows multiple threads to access a critical section.</p>
    ~<p>Mutexes cannot control thread access to a critical section.</p>
}


// question: 784371  name: Spawning a thread in C++11
```

```
::Spawning a thread in C++11::[html]<p>Which item creates a C++11 thread?</p>{
    =<p>std\:\:thread( Function&amp;&amp; f, Args&amp;&amp;... args );</p>
    ~<p>pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *),
 void *arg);</p>
    ~<pre>CreateThread(LPSECURITY_ATTRIBUTES, SIZE_T, LPTHREAD_START_ROUTINE, LPVOID, DWORD,
 LPDWORD);</pre>
}


// question: 784518  name: Wait-Free Programming Guarantees
::Wait-Free Programming Guarantees::[html]<p>Wait-free programming guarantees...</p>{
    ~%33.33333%<p>No thread will ever starve when accessing the critical section.</p>
    ~%33.33333%<p>Throughput will always be guaranteed.</p>
    ~<p>Faster performance.</p>
    ~%33.33333%<p>Ensures deadlocks will never occur.</p>
}


// question: 784374  name: What is a race condition?
::What is a race condition?::[html]<p>What is a race condition?</p>{
    =<p>When two or more threads access a critical section and modify shared data in unexpected
 ways.</p>
    ~<p>When two or more threads execute at the same time of different processor cores.</p>
    ~<p>When two or more threads recursively spawn more threads.</p>
    ~<p>When the main thread completes its work before all of the other threads in a program.</p>
}


// question: 784519  name: Forking
::Forking::[html]<p>You come across a simple program. How many child processes will have spawned its
 lifetime?</p>\n<p></p>\n<pre><code>int main() \{\n    pid_t pid0 \= fork();\n    pid_t pid1 \=
 fork();\n    if(pid0) \{\n        pid_t pid2 \= fork();\n    \}\n    return 0;\n\} <br
 /></code></pre>\n<p></p>{#
    =%100%5:0#
    =%50%6:0#<p>The main process is not a child process.</p>
}


// question: 784512  name: A Mutex is a Binary Semaphore
::A Mutex is a Binary Semaphore::[html]<p>A mutex is a binary semaphore.</p>{TRUE}
```

```
// question: 784522  name: FIFO Communication
::FIFO Communication::[html]<p>FIFOs can be shared between processes.</p>{TRUE}


// question: 784521  name: Pipe Communication
::Pipe Communication::[html]<p>Pipes provide unidirectional communication between
 processes.</p>{TRUE}


// question: 784520  name: Pipe Sharing
::Pipe Sharing::[html]<p>Pipes can only be shared between parent and child processes.</p>{TRUE}


// question: 784513  name: Semaphores and Deadlocks
::Semaphores and Deadlocks::[html]<p>Using semaphores will eliminate all potential
 deadlocks.</p>{FALSE}


// question: 784516  name: Slides Readers-Writers Example
::Slides Readers-Writers Example::[html]<p>The writer thread can starve in the readers-writers
 example provided in the lectures.</p>{TRUE}
```

10 References

[1] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*, 1 ed. Morgan Kaufmann, 2008.

[2] Kirk, D. B. *Programming Massively Parallel Processors: A Hands-on Approach*, 1 ed. Applications of GPU Computing Series. Morgan Kaufmann, 2010.

[3] Krathwohl, D. R., Bloom, B. S., and Masia, B. B. *Taxonomy of Educational Objectives.* David McKay Company, 1964.

[4] Luebke, D., and Owens, J. Intro to parallel programming, 2014.

[5] Mager, R. F. *Preparing Instructional Objectives*, second ed. Lake Publishing Company, 1984.

[6] Merrill, D., and Grimshaw, A. Revisiting sorting for gpgpu stream architectures. Tech. rep., University of Virginia, February 2010.

[7] Microsoft Corp. *Rise of the Graphics Processor* (Redmond, WA 98052 USA, July 2008), vol. 96, IEEE.

[8] Nguyen, H. *GPU Gems 3*, 3 ed. Addison-Wesley Professional, 2008.

[9] nVidia. Nvidia cuda architecture. Tech. rep., nVidia, April 2009.

[10] nVidia. Cuda c programming guide, 2014.

[11] Owens, J. Gpgpu reduce, scan, and sort. Lectures.

[12] Sanders, J., and Kandrot, E. *CUDA by Example*, 1 ed. Addison-Wesley Professional, 2011.

[13] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing. In *Graphics Hardware* (2007), vol. 2007, pp. 97–106.

[14] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, 8 ed. Wiley, December 2012.

[15] Williams, A. *Concurrency in Action*, 1 ed. Manning Publications, February 2012.