Massively Parallel Hardware

Introduction

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

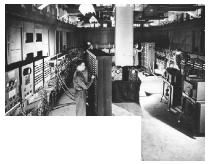
NOTE: To whom ever is reading this document. Pandoc processes divs that inherit from the 'notes' class as a note. These will not show up in your slides. To compile slides with notes run `make notes`. You will need to be running pandoc –version $>=$ 1.12.

- 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.

Massively Parallel Hardware
History
ENIAC

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.

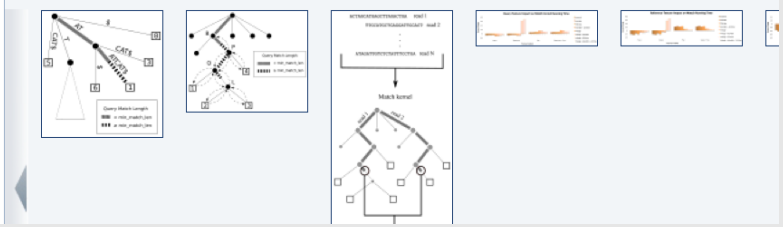
Author information

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. MUMmerG
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 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.

Keywor
Big Data,
CUDA-C,

1. INT
1.1 WI
tat

The compu
brid, with
(with doze
GPGPU) r
computer
Parallel ar
for data-ar
as for poly
[10, 7], and
16] (some
efficient ut
the GIS p
one to two
strong-sca
spective, a
computing
- most lapt
opportunit

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.

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.

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*

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.

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`

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)

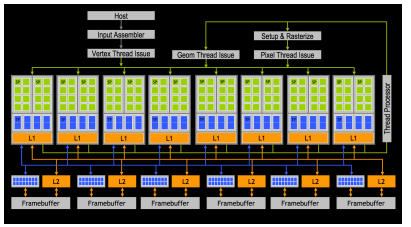


Figure 4: Nvidia's Unified Pipeline: Adopted from *Programming Massively Parallel Processors*.

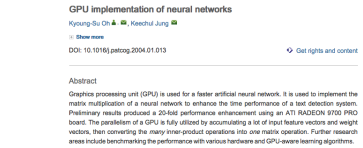
- SP stands for streaming processor. SP is also referred to as a CUDA core.
- The blocks of SPs in the images are streaming multiprocessors (SMs). NVidia's unified programmable processor array of the GeForce 8800 GTX graphics pipeline.
- First send the data in through the *Host* interface, next the data 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.

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.

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

Speed Evolution

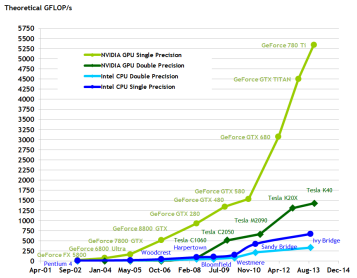


Figure 6: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

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).

Recall, the first couple

- 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.

	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.

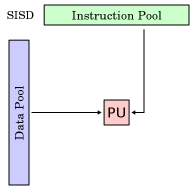


Figure 7: Wikipedia: SISD. PU stands for processing unit.

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.

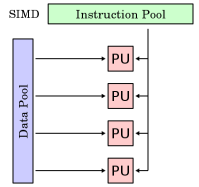


Figure 8: Wikipedia: SIMD. Keep this diagram in mind. GPU parallelism falls into this category

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.

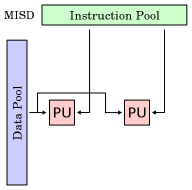


Figure 9: Wikipedia: MISD

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

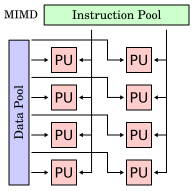


Figure 10: Wikipedia: MIMD

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

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.

- 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.

- 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.

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?

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.

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.

- 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`.

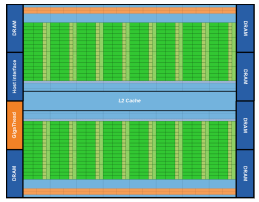


Figure 11: Retrieved from Nvidia's Fermi Whitepaper

- 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`

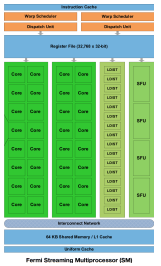


Figure 12: Retrieved from Nvidia's Fermi Whitepaper

- 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`

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

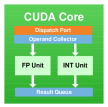


Figure 13: Retrieved from Nvidia's Fermi Whitepaper

- 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.

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.

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

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.

Blocks

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

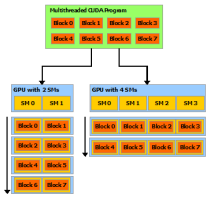


Figure 15: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

- 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*

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.

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.

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 Access Patterns

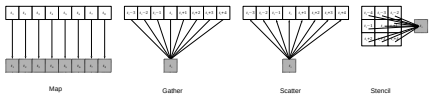


Figure 16: White boxes represent memory addresses. Grey boxes represent cells. The arrows represent the direction of communication (read / write).

- 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.

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.

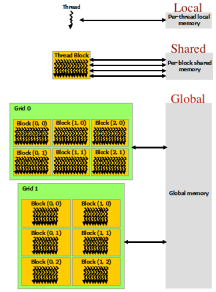


Figure 17: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

- Basic breakdown of the GPGPU memory hierarchy.

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.

Compilation

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

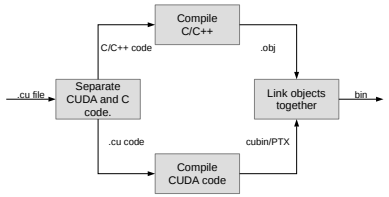


Figure 18: Simplified version of Nvidia's compilation procedure.

- `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.

- 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.

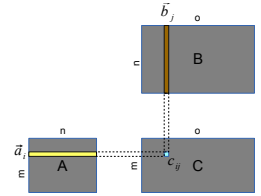


Figure 19: Reproduced from Nvidia's CUDA Toolkit Documentation

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

- 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.

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]`

- 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.

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`

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.

- 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.

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

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>>>();
  ...
}
```

- 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.

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
}
```

- 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

- 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 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`

- 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=V8TTrUdfplY

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.

Scan

- Scan applies concepts similar to reduce.
- Take a sequence of values, apply a binary operator while keeping a cumulative output.
- Example:
  - INPUT: $S = [a_0, a_1, a_2, a_3, ...a_n]$, $i = I$, $\oplus$ 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.

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.

- 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*

- 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*

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`