

# 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  $X = [x_0, x_1, \dots, x_{n-1}]$  and  $Y = [x_0^3, x_1^3, \dots, x_{n-1}^3]$ .
- $X$  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(B):429-440.

## Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.

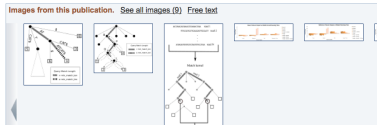
Trapnell C<sup>1</sup>, Salzberg MC.

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 genotyping. 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 layout footprint and running time and conclude higher occupancy has greater impact than reduced latency. MUMmerGPU <http://mummergpu.sourceforge.net>.

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



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

Sushil K. Prasad  
Georgia State University  
Atlanta, GA  
sprasad@gsu.edu

Xun Zhou  
University of Minnesota  
Minneapolis, MN  
xun@cs.umn.edu

Shashi Shekhar  
University of Minnesota  
Minneapolis, MN  
shekhar@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.

### Keyword

Big Data, CUDA-C,

### 1. INT 1.1 Wi tat

The computation, with (with doze GPGPU) computer Parallel ar for data-as for poly [10, 7], and 16] (some efficient ut the GIS p one to two strong-sca perspective, computing - most lap opportunity



# History

# ENIAC

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

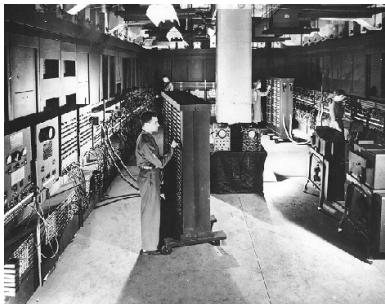


Figure 1: Unidentified Photographer [photo]. Retrieved from <http://en.wikipedia.org/wiki/ENIAC>

# 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.

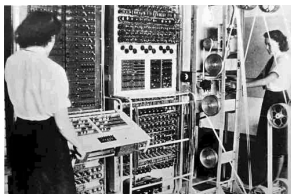


Figure 2: Retrieved from [http://en.wikipedia.org/wiki/Colossus\\_Mark\\_II](http://en.wikipedia.org/wiki/Colossus_Mark_II)

# 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

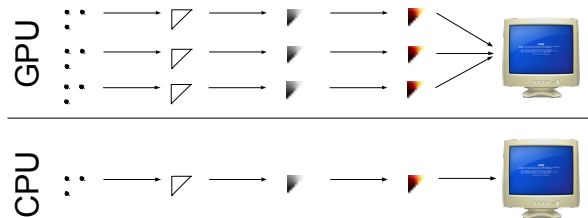


Figure 3: Simple example of the GPU vs CPU when rendering an image.

# 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)

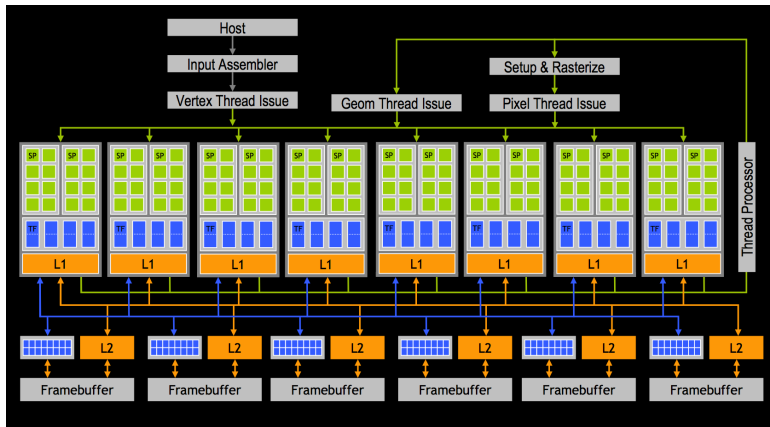


Figure 4: Nvidia's Unified Pipeline: Adopted from *Programming Massively Parallel Processors*.

# 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.<sup>1</sup>

## 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.

---

<sup>1</sup>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

Theoretical GFLOP/s

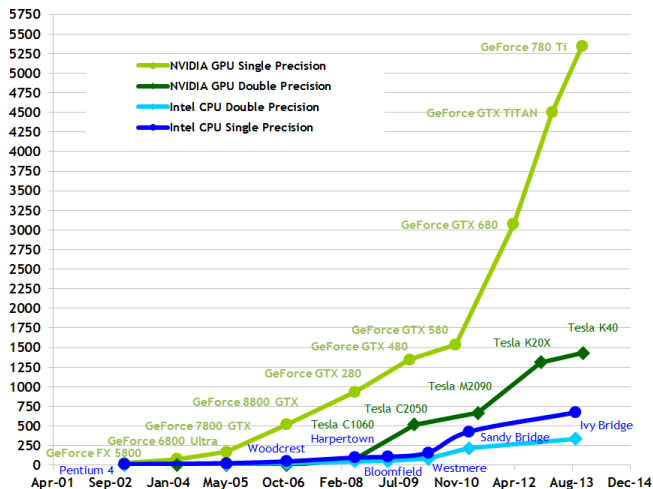


Figure 6: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

# 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-stream-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

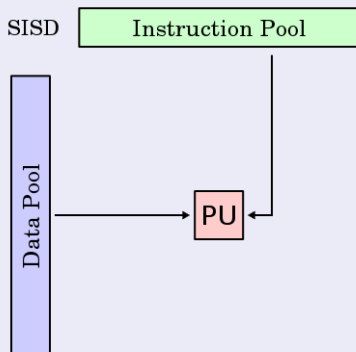


Figure 7: Wikipedia: SISD. PU stands for processing unit.

# Flynn's Taxonomy

## Single Instruction Multiple Data

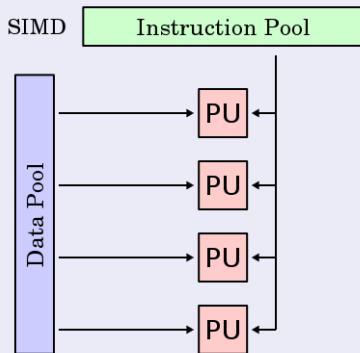


Figure 8: Wikipedia: SIMD. Keep this diagram in mind. GPU parallelism falls into this category

# Flynn's Taxonomy

## Multiple Instruction Single Data

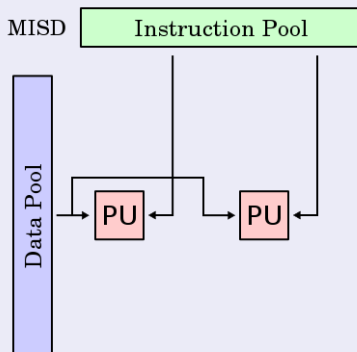


Figure 9: Wikipedia: MISD

# Flynn's Taxonomy

## Multiple Instruction Multiple Data

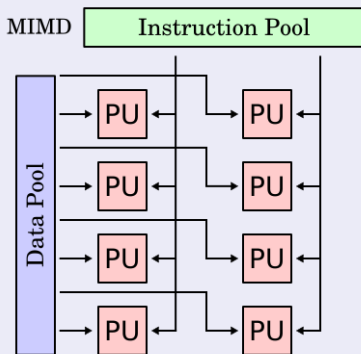


Figure 10: Wikipedia: MIMD

# 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 extremely 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

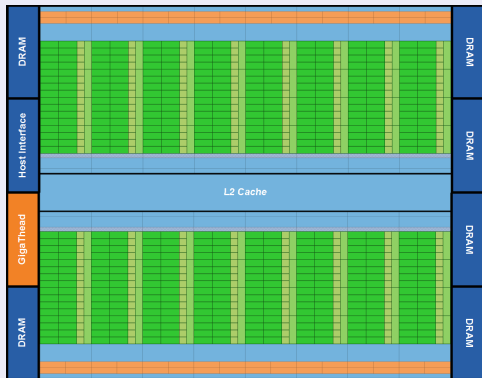


Figure 11: Retrieved from Nvidia's Fermi Whitepaper

# Nvidia GPGPU Architecture

## Fermi Streaming Multiprocessor

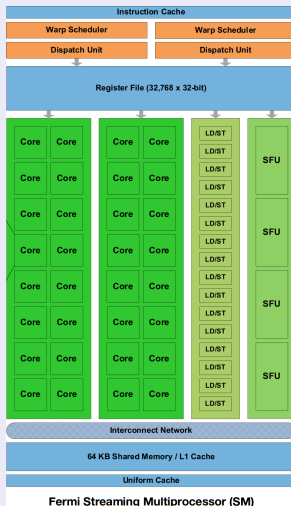


Figure 12: Retrieved from Nvidia's Fermi Whitepaper



# 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

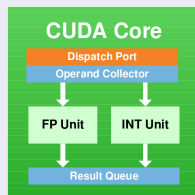


Figure 13: Retrieved from Nvidia's Fermi Whitepaper

# 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 transferred 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

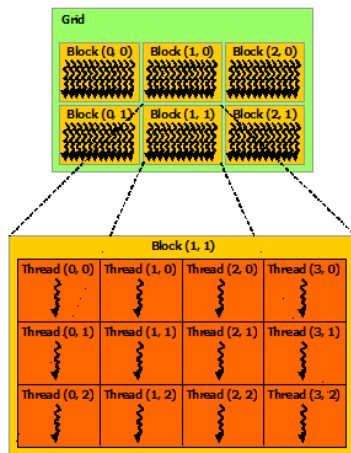


Figure 14: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation

# Blocks

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

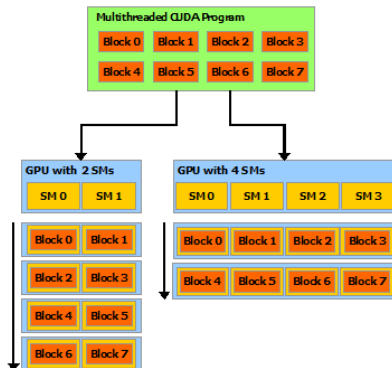


Figure 15: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit Documentation



# 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 tid = threadIdx.x + blockIdx.x * blockDim.x;  
  
    __shared__ int shared_mem[LENGTH];  
    shared_mem[tid] = dev_array[tid];  
    __syncthreads();  
  
    int b = shared_mem[tid];  
    shared_mem[tid] = cube(b);  
    __syncthreads();  
  
    dev_array[tid] = shared_mem[tid];  
}
```

# 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 tid = 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

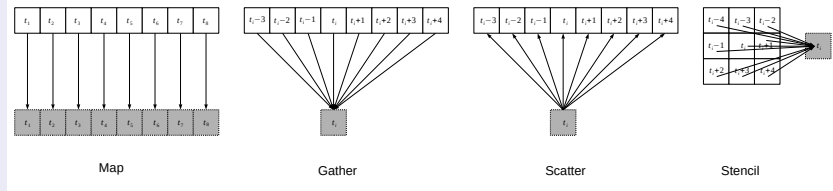


Figure 16: White boxes represent memory addresses. Grey boxes represent cells. The arrows represent the direction of communication (read / write).

# 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

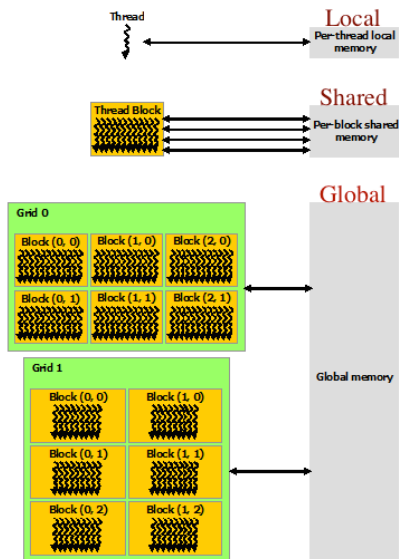


Figure 17: Retrieved from the Programmers Guide in Nvidia's CUDA Toolkit

# 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

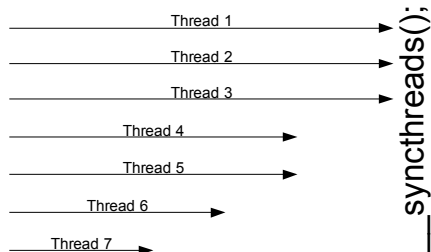
Part of Table 5.1 in *Programming Massively Parallel Processors*.

# 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 `kernel1_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. Therefore it can execute  $14 * 64 * 32 = 28672$  threads concurrently!



# Compilation

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

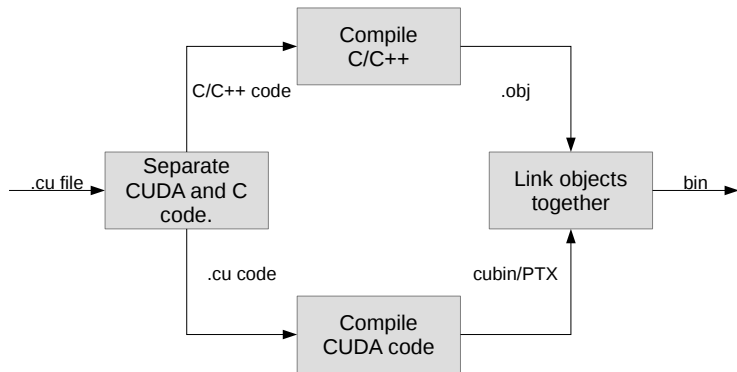


Figure 18: Simplified version of Nvidia's compilation procedure.

# 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  $i$  in  $A$  and  $\mathbf{b_j}$  is column  $j$  in  $B$ .
  - ▶  $1 \leq i \leq m$  and  $1 \leq j \leq o$
- 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

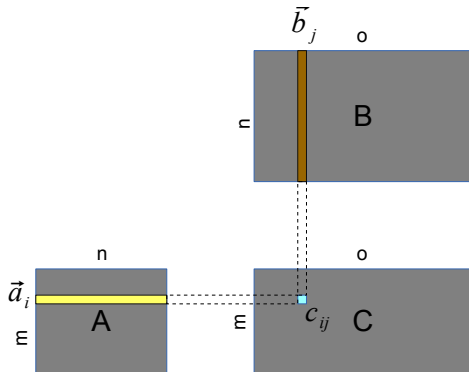


Figure 19: Reproduced from Nvidia's CUDA Toolkit Documentation

# 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  $6 \times 6$  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 Synchronization
  - ▶ `__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 Synchronization

- 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

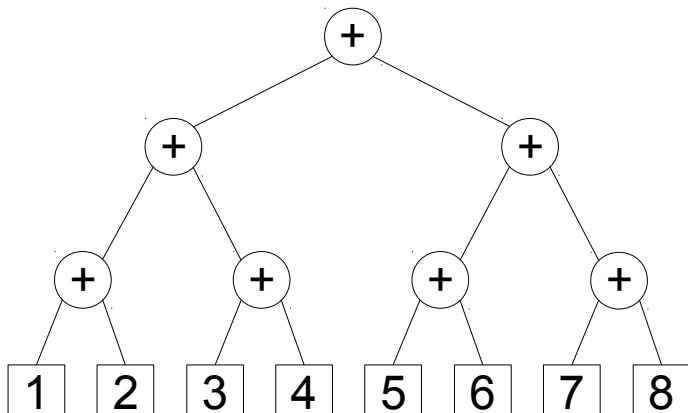
- Step Complexity
  - ▶ The number of iterations a parallel device needs to do before completing an algorithm.
- Work Complexity
  - ▶ The amount of work a parallel device does when running a algorithm.
- Work Efficient
  - ▶ A parallel algorithm that is asymptotically the same as its serial implementation.

# 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 let's 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, stride *= 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:  $S = [a_0, a_1, a_2, a_3, \dots, a_n]$ ,  $i = I$ ,  $\oplus$  where  $S$  is the sequence and  $i$  is the identity for the operator  $\oplus$ .
  - ▶ OUTPUT:
  - ▶ Exclusive:  $[I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), (I \oplus a_0 \oplus a_1 \oplus a_2), \dots, (I \oplus \dots \oplus a_{n-1})]$

# 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 < in.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.

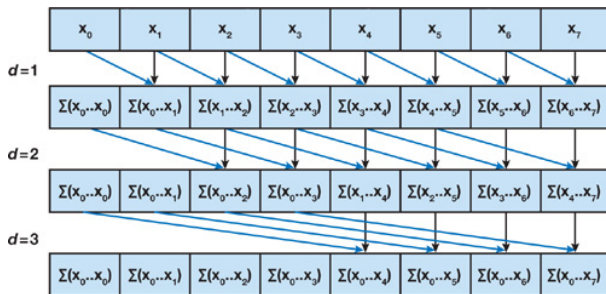


Figure 21: Hillis-Steele Scan: Adopted from *GPU Gems 3*

# 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

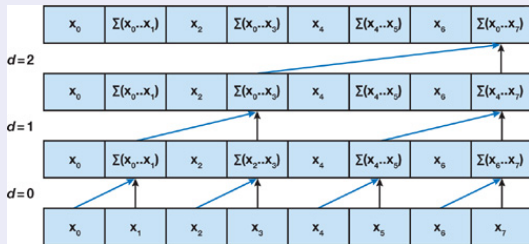


Figure 22: Blelloch Down-Sweep: Adopted from *GPU Gems 3*

## Down-Sweep Complexity

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



## 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();
    }
}
```

## Up-Sweep

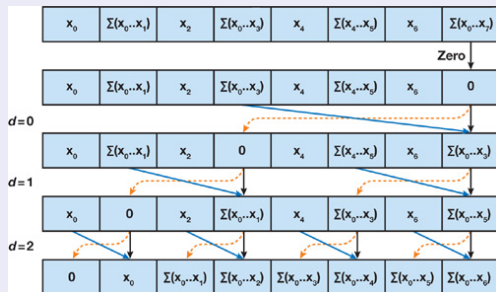


Figure 23: Blelloch Up-Sweep: Adopted from *GPU Gems 3*

## Up-Sweep Complexity

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

## 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

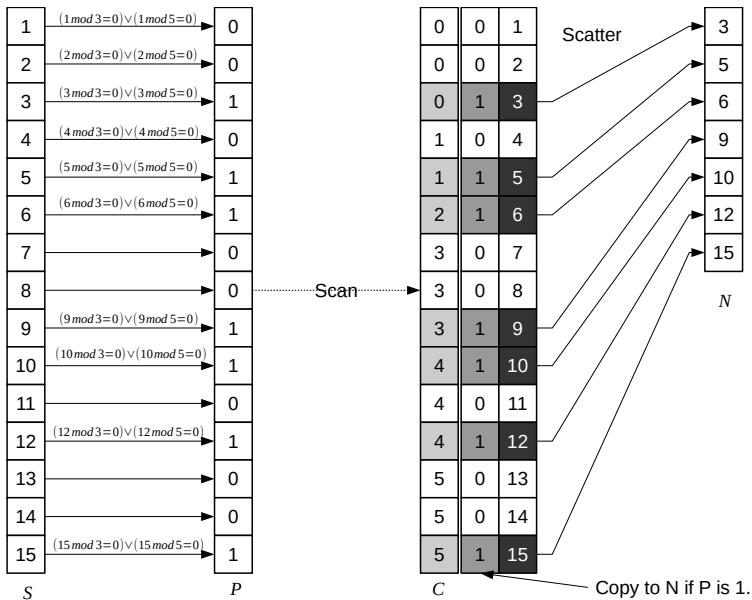
- What is compacting?
  - ▶ The process of extracting a subset from a set of items given a condition
  - ▶ Items in a set are compared against a condition. (usually an array)
  - ▶ The output of the comparison is then stored in a dataset called the predicate.
  - ▶ Using the predicate and scan operator 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  $(s \bmod 3 = 0) \vee (s \bmod 5 = 0)$
- 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](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html) for more information.