INTELLECTUAL PROPERTY RIGHTS NOTICE:

- The User may only download, make and retain a copy of the materials for his/her use for non-commercial and research purposes.
- The User may not commercially use the material, unless has been granted prior written consent by the Licensor to do so; and cannot remove, obscure or modify copyright notices, text acknowledging or other means of identification or disclaimers as they appear.
- For further details, please contact BSC-CNS patc@bsc.es

PRACE TRAINING COURSE under PRACE Advance Training Centre at BSC

BSC-CNS http://www.bsc.es/
PRACE project http://www.prace-ri.eu/
PRACE Training Portal http://www.training.prace-ri.eu/
PATC @ BSC Training Program http://www.bsc.es/marenostrum-support-services/hpc-trainings/prace-trainings
PRACE Training Course: Introduction to CUDA Programming

Area: Core HPC Curriculum

Level: BEGINNERS: for trainees from different background and very little knowledge

Prerequisites:
Basic knowledge of C/C++ programming
Attendees will need to bring their own laptops with a SSH client

Convener: Isaac Gelado

Objectives:
The aim of this course is to provide students with knowledge and hands-on experience in developing applications software for processors with massively parallel computing resources. In general, we refer to a processor as massively parallel if it has the ability to complete more than 64 arithmetic operations per clock cycle. Many commercial offerings from NVIDIA, AMD, and Intel already offer such levels of concurrency. Effectively programming these processors will require in-depth knowledge about parallel programming principles, as well as the parallelism models, communication models, and resource limitations of these processors. The target audiences of the course are students who want to develop exciting applications for these processors, as well as those who want to develop programming tools and future implementations for these processors.

Learning Outcomes:
The students who finish this course will learn how to program massively parallel processors and achieve high performance, functionality, maintainability, and scalability across future generations.
The students who finish this course will acquire technical knowledge required to achieve the above goals by learning principles and patterns of parallel algorithms, processor architecture features and constraints, and programming API, tools and techniques.
Introduction to CUDA Programming

Timetable

Day 1 / Session 1 / 9am - 1 pm: (3h lectures with 5 min breaks on the hour)
   1. Introduction to CUDA
   2. CUDA Threading Model (I)
   3. CUDA Threading Model (II)

   Session 2 / 2 pm- 6 pm: 3h practical session – lab exercises

Day 2 / Session 3 / 9am- 1 pm: (3h practical session)
   1. CUDA Memory Model
   2. Matrix Multiplication – Shared Memory
   3. 2D Convolution – Constant Memory

   Session 4 / 2 pm- 6 pm: 3h practical session – lab exercises

Day 3 / Session 5 / 9am- 1 pm: (3h practical session)
   1. CUDA Memory Model
   2. Matrix Multiplication – Shared Memory
   3. 2D Convolution – Constant Memory

   Session 6 / 2 pm- 6 pm: 3h practical session – lab exercises

Day 4 / Session 7 / 9am- 1 pm: (3h practical session)
   1. Parallel Reductions
   2. Memory Bandwidth Considerations
   3. Prefix Scan

   Session 8 / 2 pm- 6 pm: 3h practical session – lab exercises

END of COURSE
Disclaimer

All the material of this Seminar is based on the ECE408 course imparted at the University of Illinois.

All the credit for this material goes to:

- **Prof. Wen-mei W. Hwu**
  - Full Professor at the ECE and CS Departments in the University of Illinois
  - Director of the Blue-Waters Supercomputer

- **David Kirk**
  - NVIDIA Fellow
  - Professor of ECE
Course Goals

- Learn how to program massively parallel processors and achieve
  - high performance
  - functionality and maintainability
  - scalability across future generations

- Acquire technical knowledge required to achieve the above goals
  - principles and patterns of parallel algorithms
  - processor architecture features and constraints
  - programming API, tools and techniques


3. Lecture notes and recordings will be posted at the class web site
THE ADVENT OF GPUs
Performance Advantage of GPUs

An enlarging peak performance advantage:

- Calculation: 1 TFLOPS vs. 100 GFLOPS
- Memory Bandwidth: 100-150 GB/s vs. 32-64 GB/s

Courtesy: John Owens
GPU computing is catching on

280 submissions to GPU Computing Gems
- 110 articles included in two volumes
CPUs and GPUs have fundamentally different design philosophies.
CPUs: Latency Oriented Design

Large caches
- Convert long latency memory accesses to short latency cache accesses

Sophisticated control
- Branch prediction for reduced branch latency
- Data forwarding for reduced data latency

Powerful ALU
- Reduced operation latency
GPUs: Throughput Oriented Design

- **Small caches**
  - To boost memory throughput

- **Simple control**
  - No branch prediction
  - No data forwarding

- **Energy efficient ALUs**
  - Many, long latency but heavily pipelined for high throughput

- Require massive number of threads to tolerate latencies
Stretching Traditional Architectures

Traditional parallel architectures cover some super-applications
- DSP, GPU, network apps, Scientific

The game is to grow mainstream architectures “out” or domain-specific architectures “in”
- CUDA is latter
Winning Applications Use Both CPU and GPU

**CPUs for sequential parts where latency matters**
- CPUs can be 10+X faster than GPUs for sequential code

**GPUs for parallel parts where throughput wins**
- GPUs can be 10+X faster than CPUs for parallel code
A Common GPU Usage Pattern

- A desirable approach considered impractical
  - Due to excessive computational requirement
  - But demonstrated to achieve domain benefit
  - Convolution filtering (e.g. bilateral Gaussian filters), De Novo gene assembly, etc.

- Use GPUs to accelerate the most time-consuming aspects of the approach
  - Kernels in CUDA
  - Refactor host code to better support kernels

- Rethink the domain problem
Integrated host+device app C program

- Serial or modestly parallel parts in **host** C code
- Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
THE GPU MODEL
From Natural Language to Electrons

Compiler →

Natural Language (e.g., English)

Algorithm

High-Level Language (C/C++…)

Instruction Set Architecture

Microarchitecture

Circuits

Electrons

©Yale Patt and Sanjay Patel, From bits and bytes to gates and beyond
An Instruction Set Architecture (ISA) is a contract between the hardware and the software.

As the name suggests, it is a set of instructions that the architecture (hardware) can execute.
A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.

Program instructions operate on data stored in memory or provided by Input/Output (I/O) device.
The Von-Neumann Model

Memory

I/O

Processing Unit

ALU

Reg File

Control Unit

PC

IR
A CUDA kernel is executed by a grid (array) of threads

- All threads in a grid run the same kernel code (SPMD)
- Each thread has an index that it uses to compute memory addresses and make control decisions

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

Arrays of Parallel Threads
Divide thread array into multiple blocks
– Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
– Threads in different blocks cannot cooperate

Thread Block 0
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

Thread Block 1
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

Thread Block N-1
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```
• Each thread uses indices to decide what data to work on
  – blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  – threadIdx: 1D, 2D, or 3D

• Simplifies memory addressing when processing multidimensional data
  – Image processing
  – Solving PDEs on volumes
  – …
Vector Addition – Conceptual View

vector A

\[
\begin{align*}
\end{align*}
\]

vector B

\[
\begin{align*}
\end{align*}
\]

vector C

\[
\begin{align*}
\end{align*}
\]
// Compute vector sum \( C = A+B \)

```c
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0; i < n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

```c
int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```
void vecAdd(float* A, float* B, float* C, int n) {
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...
    1. // Allocate device memory for A, B, and C
       // copy A and B to device memory
    2. // Kernel launch code – to have the device
       // to perform the actual vector addition
    3. // copy C from the device memory
       // Free device vectors
}
Device code can:
- R/W per-thread registers
- R/W per-grid global memory

Host code can
- Transfer data to/from per grid global memory

We will cover more later.
CUDA Device Memory Management API functions

cudaMalloc()
- Allocates object in the device global memory
- Two parameters
  - Address of a pointer to the allocated object
  - Size of of allocated object in terms of bytes

cudaFree()
- Frees object from device global memory
  - Pointer to freed object
**cudaMemcpy()**

- Memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type of transfer

- Transfer to device is asynchronous
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size);

    // Allocate device memory for
    cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code – to be shown later
...

3. // Transfer C from device to host
    cudaMemcpy (C, C_d, size);
    // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
Example: Vector Addition Kernel

// Compute vector sum \( C = A + B \)
// Each thread performs one pair-wise addition

```c
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run \( \lceil n/256 \rceil \) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}
```

Device Code
Example: Vector Addition Kernel

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
int vecAdd(float* A, float* B, float* C, int n) {
  // A_d, B_d, C_d allocations and copies omitted
  // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(ceil(n/256), 1, 1);
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}

Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
More on CUDA Function Declarations

- **__global__** defines a kernel function
- Each “__” consists of two underscore characters
- A kernel function must return **void**
- **__device__** and **__host__** can be used together

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

**Example:**

```c
__host__ float HostFunc()
```
Kernel execution in a nutshell

__host__
Void vecAdd()
{
    dim3 DimGrid = (ceil(n/256,1,1));
    dim3 DimBlock = (256,1,1);
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d,B_d,C_d,n);
}

__global__
void vecAddKernel(float *A_d, float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

GPU

Schedule onto multiprocessors
Compiling A CUDA Program

Integrated C programs with CUDA extensions

NVCC Compiler

Host Code
Host C Compiler/Linker

Device Code (PTX)
Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs
Each thread uses IDs to decide what data to work on
- Block ID: 1D, 2D, or 3D
- Thread ID: 1D, 2D, or 3D

Simplifies memory addressing when processing multidimensional data
- Image processing
- Solving PDEs on volumes
- …
A simple illustration of the basic features of memory and thread management in CUDA programs

- Thread index usage
- Memory layout
- Register usage
- Assume square matrix for simplicity
- Leave shared memory usage until later
Square Matrix-Matrix Multiplication

\[ P = M \times N \text{ of size WIDTH x WIDTH} \]

- Each thread calculates one element of \( P \)
- Each row of \( M \) is loaded WIDTH times from global memory
- Each column of \( N \) is loaded WIDTH times from global memory
Memory Layout of a Matrix in C

\[
\begin{array}{cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3}
\end{array}
\]
void MatrixMul(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
Have each 2D thread block to compute a $(\text{TILE_WIDTH})^2$ sub-matrix (tile) of the result matrix
- Each has $(\text{TILE_WIDTH})^2$ threads

Generate a 2D Grid of $(\text{WIDTH}/\text{TILE_WIDTH})^2$ blocks

Kernel Function - A Small Example

- WIDTH = 4; TILE_WIDTH = 2
- Each block has $2 \times 2 = 4$ threads
- WIDTH/TILE_WIDTH = 2
- Use $2 \times 2 = 4$ blocks
A Slightly Bigger Example

<table>
<thead>
<tr>
<th>Block(0,0)</th>
<th>Block(1,0)</th>
<th>Block(2,0)</th>
<th>Block(3,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀,₀ P₁,₀</td>
<td>P₂,₀ P₃,₀</td>
<td>P₄,₀ P₅,₀</td>
<td>P₆,₀ P₇,₀</td>
</tr>
<tr>
<td>P₀,₁ P₁,₁</td>
<td>P₂,₁ P₃,₁</td>
<td>P₄,₁ P₅,₁</td>
<td>P₆,₁ P₇,₁</td>
</tr>
<tr>
<td>P₀,₂ P₁,₂</td>
<td>P₂,₂ P₃,₂</td>
<td>P₄,₂ P₅,₂</td>
<td>P₆,₂ P₇,₂</td>
</tr>
<tr>
<td>P₀,₃ P₁,₃</td>
<td>P₂,₃ P₃,₃</td>
<td>P₄,₃ P₅,₃</td>
<td>P₆,₃ P₇,₃</td>
</tr>
<tr>
<td>P₀,₄ P₁,₄</td>
<td>P₂,₄ P₃,₄</td>
<td>P₄,₄ P₅,₄</td>
<td>P₆,₄ P₇,₄</td>
</tr>
<tr>
<td>P₀,₅ P₁,₅</td>
<td>P₂,₅ P₃,₅</td>
<td>P₄,₅ P₅,₅</td>
<td>P₆,₅ P₇,₅</td>
</tr>
<tr>
<td>P₀,₆ P₁,₆</td>
<td>P₂,₆ P₃,₆</td>
<td>P₄,₆ P₅,₆</td>
<td>P₆,₆ P₇,₆</td>
</tr>
<tr>
<td>P₀,₇ P₁,₇</td>
<td>P₂,₇ P₃,₇</td>
<td>P₄,₇ P₅,₇</td>
<td>P₆,₇ P₇,₇</td>
</tr>
</tbody>
</table>

WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4*4 = 16 blocks
A Slightly Bigger Example (cont.)

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 = 16 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks
/* Setup the execution configuration */
/* TILE_WIDTH is a #define constant */
dim3 dimGrid(Width/TILE_WIDTH Width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

/* Launch the device computation threads! */
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
/* Matrix multiplication kernel - per thread code */
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    /* Pvalue is used to store the element of the matrix
     that is computed by the thread */
    float Pvalue = 0;
Block (0,0) in a TILE_WIDTH = 2 Configuration

Col = 0 * (TILE_WIDTH) + threadIdx.x
Row = 0 * (TILE_WIDTH) + threadIdx.y

Row = 0
Row = 1
Work for Block (1,0)

Col = 1 * (TILE_WIDTH) + threadIdx.x
Row = 0 * (TILE_WIDTH) + threadIdx.y

Row = 0
Row = 1
Work for Block (0,1)

Col = 0 \times (\text{TILE}_\text{WIDTH}) + \text{threadIdx.x}

Row = 1 \times (\text{TILE}_\text{WIDTH}) + \text{threadIdx.y}

Row = 2

Row = 3
Work for Block (1,1)

Col = 1 \times (TILE\_WIDTH) + threadIdx.x

Row = 1 \times (TILE\_WIDTH) + threadIdx.y

Col = 2

Row = 2

Row = 3
A Simple Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    /* Calculate the row index of the Pd element and M */
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    /* Calculate the column index of Pd and N */
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    /* Each thread computes one element of the block sub-matrix */
    for (int k = 0; k < Width; ++k)
        Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];

    d_P[Row*Width+Col] = Pvalue;
}
```
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)

- Programmer declares block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads

- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data

- Threads in the same block share data and synchronize while doing their share of the work

- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!
REVIEWS OF PARALLEL EXECUTION
1\textsuperscript{st} gen - Instructions are executed sequentially in program order, one at a time.

Example:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
</tr>
</tbody>
</table>
2nd gen - Instructions are executed sequentially, in program order, in an assembly line fashion. (pipeline)

Example:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction3</td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
</tbody>
</table>
History – Instruction Level Parallelism

- 3rd gen - Instructions are executed in parallel

- Example code 1:
  - \( c = b + a; \)
  - \( d = c + e; \)  
    Non-parallelizable

- Example code 2:
  - \( a = b + c; \)
  - \( d = e + f; \)  
    Parallelizable
Two forms of ILP:

- **Superscalar**: At runtime, fetch, decode, and execute multiple instructions at a time. Execution may be out of order.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction1</th>
<th>Instruction2</th>
<th>Instruction3</th>
<th>Instruction4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
</tr>
<tr>
<td>2</td>
<td>Decode</td>
<td>Decode</td>
<td>Decode</td>
<td>Decode</td>
</tr>
<tr>
<td>3</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>4</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **VLIW**: At compile time, pack multiple, independent instructions in one large instruction and process the large instructions as the atomic units.
4th gen – Multi-threading: multiple threads are executed in an alternating or simultaneous manner on the same processor/core. (will revisit)

5th gen - Multi-Core: Multiple threads are executed simultaneously on multiple processors
Hardware is free to assigns blocks to any processor at any time
- A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
Threads are assigned to Streaming Multiprocessors in block granularity

- Up to 8 blocks to each SM as resource allows
- Fermi SM can take up to 1536 threads (256 (threads/block) * 6 blocks or 512 (threads/block) * 3 blocks, etc.)

Threads run concurrently
- SM maintains thread/block id #s
- SM manages/schedules thread execution
Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
- An implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps
Going back to the program

Every instruction needs to be fetched from memory, decoded, then executed.

Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.

An example instruction cycle is the following:

Fetch | Decode | Execute | Memory
INSTRUCTIONS AND PERFORMANCE
Operate Instructions

Example of an operate instruction:
ADD R1, R2, R3

Instruction cycle for an operate instruction:
Fetch | Decode | Execute | Memory
Data Transfer Instructions

Examples of data transfer instruction:

- LDR R1, R2, #2
- STR R1, R2, #2

Instruction cycle for an operate instruction:

- Fetch | Decode | Execute | Memory
Control Flow Operations

- Example of control flow instruction:
  BRp #-4
  if the condition is positive, jump back four instructions

- Instruction cycle for an arithmetic instruction:
  Fetch | Decode | Execute | Memory
How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

Partitioning is always the same
- Thus you can use this knowledge in control flow
- However, the exact size of warps may change from generation to generation
- (Covered next)

However, DO NOT rely on any ordering between warps
- If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).
Main performance concern with branching is divergence

- Threads within a single warp take different paths
- Different execution paths are serialized in current GPUs

A common case: avoid divergence when branch condition is a function of thread ID

- Example with divergence: if(threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

- Example without divergence: if(threadIdx.x / WARP_SIZE > 2) { }
  - Also creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
SM implements zero-overhead warp scheduling
- At any time, 1 or 2 of the warps is executed by SM
- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a warp execute the same instruction when selected

TB = Thread Block, W = Warp
**Block Granularity Considerations**

For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?

- For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, there are 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

- For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.

- For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM. Also, this works for CUDA 3.0 and beyond but too large for some early CUDA versions.
The API is an extension to the C programming language.

It consists of:

- **Language extensions**
  - To target portions of the code for execution on the device.

- **A runtime library split into:**
  - A *common component* providing built-in vector types and a subset of the C runtime library in both host and device codes.
  - A *host component* to control and access one or more devices from the host.
  - A *device component* providing device-specific functions.
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`

- When executed on the host, a given function uses the C runtime implementation if available
- These functions are only supported for scalar types, not vector types
Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. __sin(x))

- __pow
- __log, __log2, __log10
- __exp
- __sin, __cos, __tan
Introduction to CUDA Programming

Lecture 7: Data Transfer and CUDA Streams
Objective

To understand the major factors that dictate performance when using GPU as an compute accelerator for the CPU

– The feeds and speeds of the traditional CPU world
– The feeds and speeds when employing a GPU
– To form a solid knowledge base for performance programming in modern GPU’s

Knowing yesterday, today, and tomorrow

– The PC world is becoming flatter
– CPU and GPU are being Fused together
– Outsourcing of computation is becoming easier…
Allocate/Free Pinned Memory (a.k.a. Page Locked Memory)

- **cudaHostAlloc()**
  - Three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use cudaHostAllocDefault for now

- **cudaFreeHost()**
  - One parameter
  - Pointer to the memory to be freed
Using Pinned Memory

- Use the allocated memory and its pointer the same way those returned by malloc();
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy function should be about 2X faster with pinned memory
So far, the way we use cudaMemCpy serializes data transfer and GPU computation.

- Trans. A
- Trans. B
- Vector Add
- Transfer

Only use one direction, GPU idle

PCIe Idle

Only use one direction, GPU idle
Some CUDA devices support *device overlap* – *Simultaneously execute a kernel while performing a copy between device and host memory*

```c
int Device;
cudaDeviceProp prop;

cudaGetDevice(&Device);
cudaGetDeviceProperties(&prop, Device);

if (prop.deviceOverlap) ...
```
Overlapped (Pieplined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

Trans A.1

Trans B.1

Comp C.1 = A.1 + B.1

Trans A.2

Trans B.2

Comp C.2 = A.2 + B.2

Trans A.3

Trans B.3

Comp C.3 = A.3 + B.3

Trans A.4

Trans B.4
CUDA supports parallel execution of kernels and MemCpy with “Streams”

Each stream is a queue of operations (kernels and MemCpys)

Operations in different streams can go in parallel
  – “Task parallelism”
Conceptual View of Streams

Operations (Kernels, MemCpys)
A Simple Multi-Stream Host Code

cudaStream_t stream0, stream1;
cudaStreamCreate( &stream0);
cudaStreamCreate( &stream1);
float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),.., stream0);
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),.., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>;
    cudaMemcpyAsync(d_C0, h_C+i; SegSize*sizeof(float),.., stream0);
for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),.., stream0);
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),.., stream0);
    VecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemcpyAsync(d_C0, h_C+i; SegSize*sizeof(float),.., stream0);

    cudaMemcpyAsync(d_A1, h_A+i+SegSize; SegSize*sizeof(float),.., stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize; SegSize*sizeof(float),.., stream1);
    VecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize; SegSize*sizeof(float),.., stream1);
}

A View Closer to Reality

Operations (Kernels, MemCpys)
Not quite the overlap want

C.1 blocks A.2 and B.2 in the copy engine queue

Trans A.1  Trans B.1  Comp C.1 = A.1 + B.1  Trans C.1

Trans A.2  Trans B.2  Comp C.2 = A.2 + B.2
A Better Multi-Stream Host Code (Cont.)

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemCpyAsync(d_A0, h_A+i; SegSize*sizeof(float),.., stream0);
    cudaMemCpyAsync(d_B0, h_B+i; SegSize*sizeof(float),.., stream0);
    cudaMemCpyAsync(d_A1, h_A+i+SegSize; SegSize*sizeof(float),.., stream1);
    cudaMemCpyAsync(d_B1, h_B+i+SegSize; SegSize*sizeof(float),.., stream1);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemCpyAsync(d_C0, h_C+i; SegSize*sizeof(float),.., stream0);
    cudaMemCpyAsync(d_C1, h_C+i+SegSize; SegSize*sizeof(float),.., stream1);
}

A View Closer to Reality

Copy Engine

MemCpy A.1
MemCpy B.1
MemCpy A.2
MemCpy B.2
MemCpy C.1
MemCpy C.2

Kernel Engine

Kernel 1
Kernel 2

Operations (Kernels, MemCpys)

Stream 0
Stream 1
Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

Trans A.1
Trans B.1
Comp C.1 = A.1 + B.1
Trans C.1
Trans A.2
Trans B.2
Comp C.2 = A.2 + B.2
Trans C.2
Trans A.3
Trans B.3
Comp C.3 = A.3 + B.3
Trans C.3
Trans A.4
Trans B.4
Introduction to CUDA Programming

Lecture 3: Tiled Matrix-Matrix Multiplication
**Transparent Scalability**

- Hardware is free to assign blocks to any processor at any time.
- A kernel scales across any number of parallel processors.

Each block can execute in any order relative to other blocks.
Threads are assigned to Streaming Multiprocessors in block granularity

- Up to 8 blocks to each SM as resource allows
- Fermi SM can take up to 1536 threads
  - Could be 256 (threads/block) * 6 blocks
  - Or 512 (threads/block) * 3 blocks, etc.
The Von-Neumann Model
The Von-Neumann Model with SIMD units
Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
- An implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps
How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)

- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must \_\_syncthreads() to get correct results (more later).
Example of control flow instruction:

BRp #-4

if the condition is positive, jump back four instructions

Instruction cycle for an arithmetic instruction:

Fetch | Decode | Execute | Memory
Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in current GPUs

- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence: `If(threadIdx.x > 2) { }`
    • This creates two different control paths for threads in a block
    • Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence: `If(threadIdx.x / WARP_SIZE > 2) { }`
    • Also creates two different control paths for threads in a block
    • Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
SM implements zero-overhead warp scheduling

- At any time, 1 or 2 of the warps is executed by SM
- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a warp execute the same instruction when selected

TB = Thread Block, W = Warp

Instruction:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB1</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB2</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB2</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB1</td>
<td>W3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB3</td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Thread Scheduling (Cont.)
Outline of Tiling Technique

- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile
Idea: Use Shared Memory to reuse global memory data

- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth.
- Tiled algorithms
Work for Block (0,0)

Col = $0 \times (\text{blockDim.x}) + \text{threadIdx.x}$
Row = $0 \times (\text{blockDim.y}) + \text{threadIdx.y}$
Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of $d_M$ and $d_N$. 
Loading a Tile

All threads in a block participate
  – Each thread loads one Md element and one Nd element in based tiled code

Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).
Work for Block (0,0)
Work for Block (0,0)

\[
\begin{array}{cccc}
N_{0,0} & N_{1,0} & N_{2,0} & N_{3,0} \\
N_{0,1} & N_{1,1} & N_{2,1} & N_{3,1} \\
N_{0,2} & N_{1,2} & N_{2,2} & N_{3,2} \\
N_{0,3} & N_{1,3} & N_{2,3} & N_{3,3} \\
\end{array}
\]

\[
\begin{array}{cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3} \\
\end{array}
\]
Work for Block (0,0)
Work for Block (0,0)
Barrier Synchronization

An API function call in CUDA
- `__synchthreads()`

All threads in the same block must reach the `__synchthreads()` before any can move on

Best used to coordinate tiled algorithms
- To ensure that all elements of a tile are loaded
- To ensure that all elements of a tile are consumed
Loading an M Tile

Upper left corner of the M tile at step m:
by * TILE_WIDTH * WIDTH + m * TILE_WIDTH

Each thread uses ty and tx to load an element
Upper left corner + ty * Width + tx
= by * TILE_WIDTH * Width + m * TILE_WIDTH +
ty * Width + tx
= (by * TILE_WIDTH + ty) * Width +
m * TILE_WIDTH + tx
= Row * Width +
m * TILE_WIDTH + tx
Row = by * TILE_WIDTH + ty
Tiled Matrix Multiplication Kernel

1. \_\_global\_\_ void MatrixMul(float* d\_M, float* d\_N, float* d\_P, int Width)
2. {
3. \_\_shared\_\_ float ds\_M[TILE\_WIDTH][TILE\_WIDTH];
4. \_\_shared\_\_ float ds\_N[TILE\_WIDTH][TILE\_WIDTH];
5. int bx = blockIdx.x; int by = blockIdx.y;
6. int tx = threadIdx.x; int ty = threadIdx.y;
7. // Identify the row and column of the Pd element to work on
8. int Row = by \* TILE\_WIDTH + ty;
9. int Col = bx \* TILE\_WIDTH + tx;
10. float Pvalue = 0;
11. // Loop over the Md and Nd tiles required to compute the Pd element
12. for (int m = 0; m < Width/TILE\_WIDTH; ++m) {
13.   // Collaborative loading of Md and Nd tiles into shared memory
14.   ds\_M[ty][tx] = d\_M[Row\*Width + m\*TILE\_WIDTH+tx];
15.   ds\_N[ty][tx] = d\_N[Col+(m\*TILE\_WIDTH+ty)*Width];
16.   \_\_syncthreads();
17.   for (int k = 0; k < TILE\_WIDTH; ++k)
18.     Pvalue += ds\_M[ty][k] \* ds\_N[k][tx];
19.   \_\_syncthreads();
20. }
21. d\_P[Row\*Width+Col] = Pvalue;
22. }
Loading an N Tile

Upper left corner of N tile at step m:
\[ bx \times \text{TILE\_WIDTH} + m \times \text{TILE\_WIDTH} \times \text{Width} \]

Each thread uses \( ty \) and \( tx \) to load an element

Upper left corner + \( ty \times \text{Width} + tx \)

\[ = bx \times \text{TILE\_WIDTH} + m \times \text{TILE\_WIDTH} \times \text{Width} + ty \times \text{Width} + tx \]

\[ = bx \times \text{TILE\_WIDTH} + tx + (m \times \text{TILE\_WIDTH} + ty) \times \text{Width} \]

\[ = \text{Col} + (m \times \text{TILE\_WIDTH} + ty) \times \text{Width} \]
First-order Size Considerations

Each thread block should have many threads
  – TILE_WIDTH of 16 gives 16*16 = 256 threads
  – TILE_WIDTH of 32 gives 32*32 = 1024 threads

For 16, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.

For 32, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations
Shared Memory and Threading

- Each SM in Fermi has 16KB or 48KB shared memory*
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses \(2 \times 256 \times 4B = 2KB\) of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to \(8 \times 512 = 4,096\) pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to \(2 \times 32 \times 32 \times 4B = 8KB\) shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time

- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 150 GB/s bandwidth can now support: \((150/4) \times 16 = 600\) GFLOPS!
Summary - Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__ ...

- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)

- Main()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemcpy(d_GlblVarPtr, h_GlblVarPtr,…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>( args…);
  - transfer results from device to host – cudaMemcpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution

- Kernel – void kernelOne(type args,…)
  - variables declaration - auto, __shared__
    • automatic variables transparently assigned to registers
  - syncthreads()…

- Other functions
  - float handyFunction(int inVar…);
Introduction to CUDA Programming

Lecture 4: Convolution, Constant Memory and Caching
2D Convolution – Inside Cells

\[
\begin{array}{ccccccc}
N & & & & & & \\
3 & 4 & 5 & 6 & 7 & & \\
2 & 3 & 4 & 5 & 6 & & \\
1 & 2 & 3 & 4 & 5 & & \\
2 & 3 & 5 & 6 & 7 & & \\
0 & 1 & 1 & 3 & 1 & & \\
\end{array}
\]

\[
\begin{array}{ccccccc}
P & & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
M & & & \\
1 & 2 & 3 & 2 & 1 \\
2 & 3 & 4 & 3 & 2 \\
3 & 4 & 5 & 4 & 3 \\
2 & 3 & 4 & 3 & 2 \\
1 & 2 & 3 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
3 & 8 & 15 & 12 & 7 & \\
4 & 9 & 16 & 15 & 12 & \\
3 & 8 & 15 & 16 & 15 & \\
4 & 9 & 20 & 18 & 14 & \\
0 & 2 & 3 & 6 & 1 & \\
\end{array}
\]
2D Convolution – Halo Cells

N

P

M

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 3 & 4 & 5 \\
0 & 2 & 3 & 4 \\
0 & 3 & 5 & 6 \\
0 & 1 & 1 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 2 \\
2 & 3 & 4 & 3 \\
3 & 4 & 5 & 4 \\
2 & 3 & 4 & 3 \\
1 & 2 & 3 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 9 & 16 & 15 & 12 \\
0 & 8 & 15 & 16 & 15 \\
0 & 9 & 20 & 18 & 14 \\
0 & 2 & 3 & 6 & 1 \\
\end{array}
\]

179
Access Pattern for M

- M is referred to as mask (a.k.a. kernel, filter, etc.)
  - Elements of M are called mask (kernel, filter) coefficients
- Calculation of all output P elements need M
- M is not changed during kernel

- Bonus - M elements are accessed in the same order when calculating all P elements
- M is a good candidate for Constant Memory
Each thread can:

- Read/write per-thread registers (~1 cycle)
- Read/write per-block shared memory (~5 cycles)
- Read/write per-grid global memory (~500 cycles)
- Read/only per-grid constant memory (~5 cycles with caching)
Memory Hierarchies

If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything.

On today’s processors, main memory accesses take hundreds of cycles.

One solution: Caches.
In order to keep cache fast, it needs to be small, so we cannot fit the entire data set in it.
Cache - Cont’d

- Cache is unit of volatile memory storage
- A cache is an “array” of cache lines
- Cache line can usually hold data from several consecutive memory addresses
- When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests
Some definitions:

- **Spatial locality**: is when the data elements stored in consecutive memory locations are accessed consecutively.
- **Temporal locality**: is when the same data element is accessed multiple times in a short period of time.

Both spatial locality and temporal locality improve the performance of caches.
Scratchpad vs. Cache

Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.

In terms of distance from the processor, scratchpad is similar to L1 cache.

Unlike cache, scratchpad does not necessarily hold a copy of data that is in main memory.

It requires explicit data transfer instructions, whereas cache doesn’t.
Cache Coherence Protocol

A mechanism for caches to propagate updates by their local processor to other caches (processors)
CPU and GPU have different caching philosophy

CPU L1 caches are usually coherent
- L1 is also replicated for each core
- Even data that will be changed can be cached in L1
- Updates to local cache copy invalidates (or less commonly updates) copies in other caches
- Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

GPU L1 caches are usually incoherent
- Avoid caching data that will be modified
How to Use Constant Memory

- Host code allocates, initializes variables the same way as any other variables that need to be copied to the device.

- Use `cudaMemcpyToSymbol(dest, src, size)` to copy the variable into the device memory.

- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.
More on Constant Caching

- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads
- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence

This is not a problem if a variable is NOT modified by a kernel.
Some Header File Stuff for M

#define KERNEL_SIZE 5

// Matrix Structure declaration
typedef struct {
    unsigned int width;
    unsigned int height;
    unsigned int pitch;
    float* elements;
} Matrix;
AllocateMatrix()

// Allocate a device matrix of dimensions height*width
//   If init == 0, initialize to all zeroes.
//   If init == 1, perform random initialization.
//   If init == 2, initialize matrix parameters, but do not allocate memory

Matrix AllocateMatrix(int height, int width, int init)
{
    Matrix M;
    M.width = M.pitch = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
// Don't allocate memory on option 2
if(init == 2) return M;
M.elements = (float*) malloc(size*sizeof(float));

for(unsigned int i = 0; i < M.height * M.width; i++)
{
    M.elements[i] = (init == 0) ? (0.0f) :
        (rand() / (float)RAND_MAX);
    if(rand() % 2) M.elements[i] = - M.elements[i]
}

return M;
Host Code

// global variable, outside any function
__constant__ float Mc[KERNEL_SIZE][KERNEL_SIZE];

... 

// allocate N, P, initialize N elements, copy N to Nd Matrix M;
M = AllocateMatrix(KERNEL_SIZE, KERNEL_SIZE, 1);
// initialize M elements
...

cudaMemcpyToSymbol(Mc, M.elements,
    KERNEL_SIZE*KERNEL_SIZE*sizeof(float));
ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
Tiling P

Use a thread block to calculate a tile of P

- Thread Block size determined by the TILE_SIZE
Each N element is used in calculating up to KERNEL_SIZE * KERNEL_SIZE P elements (all elements in the tile)
High-Level Tiling Strategy

Load a tile of N into shared memory (SM)
- All threads participate in loading
- A subset of threads then use each N element in SM
Input tiles need to be larger than output tiles.
Dealing with Mismatch

- Use a thread block that matches input tile
- Each thread loads one element of the input tile
- Some threads do not participate in calculating output
- There will be if statements and control divergence
Shifting from output coordinates to input coordinates
Shifting from output coordinates to input coordinate

```c
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y * TILE_SIZE + ty;
int col_o = blockIdx.x * TILE_SIZE + tx;

int row_i = row_o - 2;
int col_i = col_o - 2;
```
Threads that loads halos outside N should return 0.0
float output = 0.0f;

if((row_i >= 0) && (row_i < N.height) &&
   (col_i >= 0) && (col_i < N.width) ) {
    Ns[ty][tx] = N.elements[row_i*N.width + col_i];
}
else{
    Ns[ty][tx] = 0.0f;
}
Some threads do not participate in calculating output

```c
if(ty < TILE_SIZE && tx < TILE_SIZE){
    for(i = 0; i < 5; i++) {
        for(j = 0; j < 5; j++) {
            output += Mc[i][j] * Ns[i+ty][j+tx];
        }
    }
}
```
Some threads do not write output

```c
if(row_o < P.height && col_o < P.width)
    P.elements[row_o * P.width + col_o] = output;
```
Setting Block Size

#define BLOCK_SIZE (TILE_SIZE + 4)

dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);

In general, block size should be tile size + (kernel size -1)
Tiling Benefit Analysis

- Start with KERNEL_SIZE = 5
- Each point in an input tile is used multiple times.
  - Each boundary point (blue) is used 9 times
  - Each second boundary point (yellow) is used 16 times
  - Each inner boundary point (red) is used 25 times
For TILE_SIZE = 12
- 44 boundary points
- 36 boundary points
- 64 inside points
- Total uses $44 \times 9 + 36 \times 16 + 64 \times 25 = 396 + 576 + 1600 = 2572$
- Average reuse = $2572/144 = 17.9$

As TILE_SIZE increases, the average reuse approach 25
In General

- The number of boundary layers is proportional to the \texttt{KERNEL\_SIZE}
- The maximal reuse of each data point is \((\texttt{KERNEL\_SIZE})^2\)
- \texttt{BLOCK\_SIZE} is limited by the maximal number of threads in a thread block
- Input tile sizes could be \(\texttt{N\times TILE\_SIZE} + (\texttt{KERNEL\_SIZE}-1)\)
  - By having each thread to calculate \(\texttt{N}\) input points (thread coarsening)
  - \(\texttt{N}\) is limited is limited by the shared memory size
- \texttt{KERNEL\_SIZE} is decided by application needs
Applied CUDA Programming
Lecture 5: Reductions
Partition and Summarize

A commonly used strategy for processing large input data sets
- There is no required order of processing elements in a data set (associative and commutative)
- Partition the data set into smaller chunks
- Have each thread to process a chunk
- Use a reduction tree to summarize the results from each chunk into the final answer

We will focus on the reduction tree step for now.

Google and Hadoop MapReduce frameworks are examples of this pattern
Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations

- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation

Summarize a set of input values into one value using a “reduction operation”
- Max
- Min
- Sum
- Product
- Often with user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
A sequential reduction algorithm performs N operations

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value
A parallel reduction performs N-1 Operations in log(N) steps
For N input values, the reduction tree performs
- \((1/2)N + (1/4)N + (1/8)N + \ldots + (1/N) = (1 - (1/N))N = N-1 \) operations
- In Log (N) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism \((N-1)/\text{Log}(N))\)
  - For \(N = 1,000,000\), average parallelism is 50,000
  - However, peak resource requirement is 500,000!

This is a work-efficient parallel algorithm
- The amount of work done is comparable to sequential
- Many parallel algorithms are not work efficient
A Sum Reduction Example

Parallel implementation:
- Recursively halve # of threads, add two values per thread in each step
- Takes log(n) steps for n elements, requires n/2 threads

Assume an in-place reduction using shared memory
- The original vector is in device global memory
- The shared memory is used to hold a partial sum vector
- Each step brings the partial sum vector closer to the sum
- The final sum will be in element 0
- Reduces global memory traffic due to partial sum values
Vector Reduction with Branch Divergence

Thread 0 | Thread 1 | Thread 2 | Thread 3 | Thread 4 | Thread 5
---|---|---|---|---|---
0..3 | 2+3 | 4+5 | 6+7 | 10+11 | 8+9
0+1 | 4..7 | | 8..11 | | 8..15
0 | 4 | 5 | 6 | 7 | 8
A Sum Example

Thread 0  Thread 1  Thread 2  Thread 3

Data

3 1 7 0 4 1 6 3

1 4 7 5 9

2 11 14

3 25

Active Partial Sum elements

steps
Simple Thread Index to Data Mapping

Each thread is responsible of an even-index location of the partial sum vector
  – One input is the location of responsibility

After each step, half of the threads are no longer needed

In each step, one of the inputs comes from an increasing distance away
A Simple Thread Block Design

Each thread block takes 2* BlockDim input elements
Each thread loads 2 elements into shared memory

```c
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] =
    input[start + t];
partialSum[blockDim+t] =
    input[start+ blockDim.x+t];
```
The Reduction Steps

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

Why do we need `syncthreads()`?
Barrier Synchronization

- syncthreads() are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

- Why do we not need another syncthread() at the end of the reduction loop?
Thread 0 in each thread block write the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x 

There can be a large number of such sums if the original vector is very large
  – The host code may iterate and launch another kernel 

If there are only a small number of sums, the host can simply transfer the data back and add them together.
Some Observations

In each iteration, two control flow paths will be sequentially traversed for each warp
- Threads that perform addition and threads that do not
- Threads that do not perform addition still consume execution resources

No more than half of threads will be executing after the first step
- All odd index threads are disabled after first step
- After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
  • This can go on for a while, up to 5 more steps (1024/32=16= 2^5), where each active warp only has one productive thread until all warps in a block retire
In some algorithms, one can shift the index usage to improve the divergence behavior

- Commutative and associative operators

Example - given an array of values, “reduce” them to a single value in parallel

- Sum reduction: sum of all values in the array
- Max reduction: maximum of all values in the array
- ...
A Better Strategy

- Always compact the partial sums into the first locations in the partialSum[] array

- Keep the active threads consecutive
An Example of 16 threads
A Better Reduction Kernel

```c
for (unsigned int stride = blockDim.x/2;
    stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```
A Quick Analysis

For a 1024 thread block

- No divergence in the first 5 steps
- 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
- The final 5 steps will still have divergence
Introduction to CUDA Programming

Lecture 9: MPI and CUDA Programming
Outline

- MPI for dummies
- MPI meets CUDA
- MPI and CUDA Example: 3D Stencil
- MPI and CUDA 4.0
Message Passing Interface

- MPI is a standard message passing API

- Oriented to cluster machines
  - Distributed memory
  - Hides underlying interconnection network

- Processes execute on different nodes of a network
MPI Model

- Many processes distributed in a cluster
- Each process computes part of the output
- Processes communicate with each other
- Processes can synchronize
MPI Message Types

**Point-to-point communication**
- Send and Receive

**Collective communication**
- Barrier
- Broadcast
- Reduce
- Gather and Scatter
**MPI Initialization, Info and Sync**

```c
int MPI_Init(int *argc, char ***argv)
```
- Initialize MPI

```c
MPI_COMM_WORLD
```
- MPI group with all allocated nodes

```c
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```
- Rank of the calling process in group of comm

```c
int MPI_Comm_size (MPI_Comm comm, int *size)
```
- Number of processes in the group of comm

```c
int MPI_Barrier (MPI_Comm comm)
```
- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.
**MPI Sending Data**

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- **Buf**: Initial address of send buffer (choice)
- **Count**: Number of elements in send buffer (nonnegative integer)
- **Datatype**: Datatype of each send buffer element (handle)
- **Dest**: Rank of destination (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)

**DATA_DISTRIBUTED**: Send to all nodes
MPI Receiving Data

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **Buf**: Initial address of receive buffer (choice)
- **Count**: Maximum number of elements in receive buffer (integer)
- **Datatype**: Datatype of each receive buffer element (handle)
- **Source**: Rank of source (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)
- **Status**: Status object (Status)
MPI Sending and Receiving Data

```c
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

- **Sendbuf**: Initial address of send buffer (choice)
- **Sendcount**: Number of elements in send buffer (integer)
- **Sendtype**: Type of elements in send buffer (handle)
- **Dest**: Rank of destination (integer)
- **Sendtag**: Send tag (integer)
- **Recvcount**: Number of elements in receive buffer (integer)
- **Recvtype**: Type of elements in receive buffer (handle)
- **Source**: Rank of source (integer)
- **Recvtag**: Receive tag (integer)
- **Comm**: Communicator (handle)
- **Recvbuf**: Initial address of receive buffer (choice)
- **Status**: Status object (Status). This refers to the receive operation.
Outline

- MPI for dummies
- MPI meets CUDA
- MPI and CUDA Example: 3D Stencil
- MPI and CUDA 4.0
CUDA-based cluster

- Each node contains $N$ GPUs

```
  GPU 0  |  GPU N
  PCIe   |  PCIe
  CPU 0  |  CPU M
  Host Memory

  GPU 0  |  GPU N
  PCIe   |  PCIe
  CPU 0  |  CPU M
  Host Memory
```
CUDA and MPI Communication

**Source MPI process:**
- `cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost)`
- `MPI_Send()`

**Destination MPI process:**
- `MPI_Recv()`
- `cudaMemcpy(dst, src, cudaMemcpyDeviceToDevice)`
Outline

- MPI for dummies
- MPI meets CUDA
- MPI and CUDA Example: 3D Stencil
- MPI and CUDA 4.0
int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node_stencil(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
Volumes are split into tiles (along the Z-axis)
- 3D-Stencil introduces data dependencies
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np, num_comp_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes = num_points * sizeof(float);
    float *input = 0, *output = 0;
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL) {
        printf("server couldn't allocate memory\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    /* Initialize input data */
    random_data(input, dimx, dimy, dimz, 1, 10);
    /* Calculate number of shared points */
    int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
    int int_num_points = dimx * dimy * (dimz / num_comp_nodes + 8);
    float *send_address = input;
/* Send data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
         DATA_DISTRIBUTE, MPI_COMM_WORLD);
send_address += dimx * dimy * (dimz / num_comp_nodes - 4);

/* Send data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
    MPI_Send(send_address, int_num_points, MPI_REAL, process,
             DATA_DISTRIBUT, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
         DATA_DISTRIBUT, MPI_COMM_WORLD);
/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_comp_nodes; process++)
  MPI_Recv(output + process * num_points / num_comp_nodes,
           num_points / num_comp_nodes, MPI_REAL, process,
           DATA_COLLECT, MPI_COMM_WORLD, &status );

/* Store output data */
store_output(output, dimx, dimy, dimz);

/* Release resources */
free(input);
free(output);
Approach: two-stage execution

- Stage 1: compute the field points to be exchanged
Boundary Exchange Example (II)

Approach: two-stage execution

- Stage 2: Compute the remaining points while exchanging the boundaries
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points * sizeof(float);

    int left_ghost_offset   = 0;
    int right_ghost_offset  = dimx * dimy * (4 + dimz);
    int left_stage1_offset  = 0;
    int right_stage1_offset = dimx * dimy * (dimz - 4);
    int stage2_offset       = num_ghost_points;
}
float *h_input = NULL, *h_output = NULL;
float *d_input = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_left_ghost_own = NULL, *h_right_ghost_own = NULL;
float *h_left_ghost = NULL, *h_right_ghost = NULL;

/* Alloc host memory */
h_input  = (float *)malloc(num_bytes);
    h_output = (float *)malloc(num_bytes);

    /* Alloc host memory for ghost data */
cudaMallocHost((void **)&h_left_ghost_own, num_ghost_bytes);
cudaMallocHost((void **)&h_right_ghost_own, num_ghost_bytes);
cudaMallocHost((void **)&h_left_ghost, num_ghost_bytes);
cudaMallocHost((void **)&h_right_ghost, num_ghost_bytes);

    /* Alloca device memory for input and output data */
cudaMalloc((void **)&d_input, num_bytes);
cudaMalloc((void **)&d_output, num_bytes);
MPI_Status status;
int left_neighbor = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;
int server_process = np - 1;

/* Get the input data from main process */
float *rcv_address = h_input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_REAL, server_process,
         DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);

/* Upload stencil coefficients */
upload_coefficients(coeff, 5);

/* Create streams used for stencil computation */
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; I < nreps; i++) {
    /* Compute values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
                  d_input + left_stage1_offset, dimx, dimy, 12, stream1);
    launch_kernel(d_output + right_stage1_offset,
                  d_input + right_stage1_offset, dimx, dimy, 12, stream1);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                  dimx, dimy, dimz, stream2);

    /* Copy the data needed by other nodes to the host */
    cudaMemcpyAsync(h_left_ghost_own,
                     d_output + num_ghost_points,
                     num_ghost_bytes, cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_right_ghost_own,
                     d_output + right_stage1_offset + num_ghost_points,
                     num_ghost_bytes, cudaMemcpyDeviceToHost, stream1);
    cudaStreamSynchronize(stream1);
/* Send data to left, get data from right */
MPI_Sendrecv(h_left_ghostOwn, num_ghost_points, MPI_REAL,
    left_neighbor, i, h_right_ghost,
    num_ghost_points, MPI_REAL, right_neighbor, i,
    MPI_COMM_WORLD, &status);

/* Send data to right, get data from left */
MPI_Sendrecv(h_right_ghostOwn, num_ghost_points, MPI_REAL,
    right_neighbor, i, h_left_ghost,
    num_ghost_points, MPI_REAL, left_neighbor, i,
    MPI_COMM_WORLD, &status);

cudaMemcpyAsync(d_output + left_ghost_offset,  h_left_ghost,
    num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(d_output + right_ghost_offset, h_right_ghost,
    num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
cudaDeviceSynchronize();

float *temp = d_output;
d_output = d_input; d_input = temp;
/* Wait for previous communications */
MPI_Barrier(MPI_COMM_WORLD);

float *temp = d_output;
d_output = d_input;
d_input = temp;

/* Send the output, skipping ghost points */
cudaMemcpy(h_output, d_output, num_bytes, cudaMemcpyDeviceToHost);
float *send_address = h_output + num_ghost_points;
MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
    server_process, DATA_COLLECT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

/* Release resources */
free(h_input); free(h_output);
cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
cudaFree(d_input); cudaFree(d_output);
Outline

- MPI for dummies
- MPI meets CUDA
- MPI and CUDA Example: 3D Stencil
- MPI and CUDA 4.0
Without GPU Direct

There is an internal copy (not seen by the user) between CUDA buffers and Infinibad buffers
With GPU Direct

- There is no internal copy, increasing performance
- The program code remains unchanged
CUDA 4.0 and MPI

- MPI Processes handle more than one GPU

- Peer GPU to GPU communication without need for MPI
GMAC and MPI

- MPI Processes handle more than one GPU using several CPU (host) threads

- MPI calls use host (shared) memory addresses

- GMAC double-buffers MPI send/receive commands
Applied CUDA Programming

Lecture 6: Prefix Scan
Objective

- To master parallel Prefix Sum (Scan) algorithms
  - frequently used for parallel work assignment and resource allocation
  - A key primitive to in many parallel algorithms to covert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree

Reading –Mark Harris, Parallel Prefix Sum with CUDA
Definition: The all-prefix-sums operation takes a binary associative operator \( \oplus \), and an array of \( n \) elements 
\[
[x_0, x_1, \ldots, x_{n-1}],
\]
and returns the array 
\[
[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].
\]

Example: If \( \oplus \) is addition, then the all-prefix-sums operation on the array 
\[
[3 1 7 0 4 1 6 3],
\]
would return 
\[
[3 4 11 11 15 16 22 25].
\]
A Inclusive Scan Application Example

- Assume that we have a 100-inch sausage to feed 10
- We know how much each person wants in inches
  - \([3 \ 5 \ 2 \ 7 \ 28 \ 4 \ 3 \ 0 \ 8 \ 1]\)
- How do we cut the sausage quickly?
- How much will be left
- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan
  - \([3, \ 8, \ 10, \ 17, \ 45, \ 49, \ 52, \ 52, \ 60, \ 61]\) (39 inches left)
Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- ...

...
A Inclusive Sequential Prefix-Sum

Given a sequence \([x_0, x_1, x_2, \ldots]\)
Calculate output \([y_0, y_1, y_2, \ldots]\)

Such that
\[
y_0 = x_0 \\
y_1 = x_0 + x_1 \\
y_2 = x_0 + x_1 + x_2 \\
\ldots
\]

Using a recursive definition
\[
y_i = y_{i-1} + x_i
\]
A Work Efficient C Implementation

```c
y[0] = x[0];
for (i = 1; i < Max_i; i++) y[i] = y[i-1] + x[i];
```

Computationally efficient:
- N additions needed for N elements - O(N)!
A Naïve Inclusive Parallel Scan

Assign one thread to calculate each y element

Have every thread to add up all x elements needed for the y element

\[ y_0 = x_0 \]
\[ y_1 = x_0 + x_1 \]
\[ y_2 = x_0 + x_1 + x_2 \]

“Parallel programming is easy as long as you do not care about performance.”
Let’s Look at the Reduction Tree Again
Reduction Scan Step

\[ \sum_{i=0}^{3} x_i \]

Time

In place calculation

Final value after reduce
Inclusive Post Scan Step

Move (add) a critical value to a central location where it is needed
Inclusive Post Scan Step
Putting All Together
Reduction Step Kernel Code

/* scan_array[BLOCK_SIZE] is in shared memory */
int stride = 1;
while(stride < BLOCK_SIZE)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < BLOCK_SIZE)
        scan_array[index] += scan_array[index-stride];

    stride = stride*2;
    __syncthreads();
}

threadIdx.x+1 = 1, 2, 3, 4…. stride = 1, index =
Putting All Together
int stride = BLOCK_SIZE >> 1;

while(stride > 0)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < BLOCK_SIZE) {
        scan_array[index+stride] += scan_array[index];
    }
    stride = stride >> 1;
    __syncthreads();
}
**Definition:** *The all-prefix-sums operation takes a binary associative operator \( \oplus \), and an array of \( n \) elements \([a_0, a_1, \ldots, a_{n-1}]\), and returns the array \([0, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]\).*

**Example:** If \( \oplus \) is addition, then the all-prefix-sums operation on the array \([3, 1, 7, 0, 4, 1, 6, 3]\), would return \([0, 3, 4, 11, 11, 15, 16, 22]\).
Why Exclusive Scan

- To find the beginning address of allocated buffers

- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

Exclusive: [0 3 4 11 11 15 16 22]
Inclusive: [3 4 11 11 15 16 22 25]
Exclusive Post Scan Step

\[ x_0 \quad \sum x_{0..1} \quad x_2 \quad \sum x_{0..3} \quad x_4 \quad \sum x_{4..5} \quad x_6 \quad 0 \]

\[ \sum x_{0..3} \quad + \quad 0 \]

\[ \sum x_{0..3} \]
Exclusive Post Scan Step

\[ x_0 \quad \Sigma x_{0..1} \quad x_2 \quad \Sigma x_{0..3} \quad x_4 \quad \Sigma x_{4..5} \quad x_6 \quad 0 \]

\[ 0 \quad \Sigma x_{0..1} \quad 0 \quad \Sigma x_{0..3} \quad \Sigma x_{4..5} \quad \Sigma x_{0..6} \]

\[ 0 \quad x_0 \quad \Sigma x_{0..1} \quad \Sigma x_{0..2} \quad \Sigma x_{0..3} \quad \Sigma x_{0..4} \quad \Sigma x_{0..5} \quad \Sigma x_{0..6} \]
Inclusive Post Scan Step
### Exclusive Scan Example – Reduction Step

Assume array is already in shared memory

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
</table>
Reduction Step (cont.)

Stride 1

Iteration 1, \( n/2 \) threads

Each \( \bigoplus \) corresponds to a single thread.

Iterate \( \log(n) \) times. Each thread adds value \( stride \) elements away to its own value.
Reduction Step (cont.)

Stride 1

Iteration 2, \( \frac{n}{4} \) threads

Stride 2

Iterate \( \log(n) \) times. Each thread adds value \( \text{stride} \) elements away to its own value.

Each \( \text{thread} \) corresponds to a single thread.
Iterate log(n) times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering.
We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.
Post Scan Step from Partial Sums

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |
Post Scan Step from Partial Sums (cont.)

Iterate log(n) times. Each thread adds value stride elements away to its own value, and sets the value stride elements away to its own previous value.

Each corresponds to a single thread.
Iterate \( \log(n) \) times. Each thread adds value \( stride \) elements away to its own value, and sets the value \( stride \) elements away to its own previous value.

Each circle corresponds to a single thread.
Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 \times \log(n)$.
Total work: $2 \times (n-1)$ adds $= O(n)$ **Work Efficient!**
The parallel Inclusive Scan executes $2 \times \log(n)$ parallel iterations

- $\log(n)$ in reduction and $\log(n)$ in post scan
- The iterations do $n/2$, $n/4$,..1, 1, ..., $n/4$. $n/2$ adds
- Total adds: $2 \times (n-1) \to O(n)$ work

The total number of adds is no more than twice of that done in the efficient sequential algorithm

- The benefit of parallelism can easily overcome the 2X work
A Plausible Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

<table>
<thead>
<tr>
<th>In</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.
A Plausible Parallel Scan Algorithm

1. (previous slide)

2. Iterate \( \log(n) \) times: Threads \( \text{stride to } n \): Add pairs of elements \( \text{stride} \) elements apart. Double \( \text{stride} \) at each iteration. (note must double buffer shared mem arrays)

## Iteration #1

### Stride = 1

- Active threads: \( \text{stride to } n-1 \) (\( n-\text{stride} \) threads)
- Thread \( j \) adds elements \( j \) and \( j-\text{stride} \) from T0 and writes result into shared memory buffer T1 (ping-pong)
A Plausible Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads stride to n: Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

**Iteration #2**

Stride = 2
A Plausible Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads \textit{stride} to \textit{n}: Add pairs of elements \textit{stride} elements apart. Double \textit{stride} at each iteration. (note must double buffer shared mem arrays)

In \begin{tabular}{c} 3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3 \end{tabular}

T0 \begin{tabular}{c} 0 \ 3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \end{tabular}

Stride 1

T1 \begin{tabular}{c} 0 \ 3 \ 4 \ 8 \ 7 \ 4 \ 5 \ 7 \end{tabular}

Stride 2

T0 \begin{tabular}{c} 0 \ 3 \ 4 \ 11 \ 11 \ 12 \ 12 \ 11 \end{tabular}

T1 \begin{tabular}{c} 0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \end{tabular}

Iteration #3
Stride = 4
A Plausible Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads *stride* to *n*: Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

3. Write output to device memory.
Work Efficiency Considerations

- The plausible parallel Scan executes \( \log(n) \) parallel iterations
  - The steps do \((n-1), (n-2), (n-4), \ldots (n-n/2)\) adds
  - Total adds: \(n \times \log(n) + (n-1) \rightarrow O(n \times \log(n))\) work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does \(n\) adds
  - A factor of \(\log(n)\) hurts: 20x for \(10^6\) elements!

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency
Working on Arbitrary Length Input

- Build on the scan kernel that handles up to 2*blockDim elements
- Have each section of 2*blockDim elements assigned to each block
- Have each block write the sum of its section into a Sum array indexed by blockIdx.x
- Run parallel scan on the Sum array
  - May need to break down Sum into multiple sections if it is too big for a block
- Add the scanned Sum array values to the elements of corresponding sections
Overall Flow of Complete Scan

Initial Array of Arbitrary Values

Scan Block 0 → Scan Block 1 → Scan Block 2 → Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum \(i\) to All Values of Scanned Block \(i + 1\)

Final Array of Scanned Values
INTELLECTUAL PROPERTY RIGHTS NOTICE:

• The User may only download, make and retain a copy of the materials for his/her use for non-commercial and research purposes.

• The User may not commercially use the material, unless has been granted prior written consent by the Licensor to do so; and cannot remove, obscure or modify copyright notices, text acknowledging or other means of identification or disclaimers as they appear.

• For further details, please contact BSC-CNS patc@bsc.es