Using The CUDA Programming Model

Leveraging GPUs for Application Acceleration

Dan Ernst, Brandon Holt
University of Wisconsin – Eau Claire
What is (Historical) GPGPU?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

- Applications – see GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Why GPGPU Processing?

- A quiet revolution
  - Calculation: TFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: ~10x

Figure 1.1. Enlarging Performance Gap between GPUs and CPUs.

- GPU in every PC—massive volume and potential impact

Courtesy: John Owens
Intel P4 Northwood

Buffer Allocation & Register Rename
- Instruction Queue (for less critical fields of the uOps)
- General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)
- Floating Point, MMX, SSE2
- Renamed Register File
- 128 entries of 128 bit.

uOp Schedulers
- FP Move Scheduler: (8x8 dependency matrix)
- Parallel (Matrix) Scheduler for the two double pumped ALU's
- General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)
- Load / Store uOp Scheduler: (8x8 dependency matrix)
- Load / Store Linear Address Collision History Table

Integer Execution Core
1. uOp Dispatch unit & Replay Buffers
2. Integer Renamed Register File
3. 128 entries of 32 bit + 6 status flags
4. 12 real ports and six write ports
5. Datathrough switch & Bypasses to and from the Integer Register File.
6. Flags, Write Back
7. Double Pumped ALU 0
8. Double Pumped ALU 1
9. Load Address Generator Unit
10. Store Address Generator Unit
11. Load Buffer (148 entries)
12. Store Buffer (24 entries)

Execution Pipeline Start
- Register Alias History Tables (2x 208)
- uOp Queue

Instruction Trace Cache
- Micro code Sequencer Micro code ROM & Flash
- Trace Cache Fill Buffers
- Distributed Tag comparators
- 25 bit virtual tag

Instruction Fetch
- from L2 cache and Branch Prediction
- 32x64 entry, fully associative
- 4096 entries in total
- Instruction TLBs 2x64 entry, fully associative for 4k and 4M pages
- In: Virtual address [31:12]
- Out: Physical address [35:12] + 2 page level bits

Instruction Decode
- Up to 4 decoded uOps/cycle output
- (from max. one x86 instr/cycle)
- Instructions with more than four are handled by Microsequencer
- Trace Cache LRU bits
- Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Trace Cache Access, next Address Predict
- Trace Cache Branch Prediction Table (BTB), 512 entries
- Return Stacks (2x16 entries)
- Trace Cache next IP's (2x)
- Miscellaneous Tag Data

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Front Side Bus Interface, 400..800 MHz

April 19, 2003 www.chip-architect.com
NVIDIA GT200
NVIDIA GT200

GeForce GTX 280

Penryn
GeForce 8800 (2007)
G80 Characteristics

- **367 GFLOPS** peak performance (25-50 times of current high-end microprocessors)
- **265 GFLOPS** sustained for apps such as VMD
- Massively parallel, 128 cores, **90W**
- Massively threaded, sustains 1000s of threads per app
- **30-100 times speedup** over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

“I think they're right on the money, but the huge performance differential (currently 3 GPUs ~ 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publicly until I triple check those numbers.”

John Stone, VMD group, Physics, UIUC
Fermi (Earlier this year)

~1.5TFLOPS (SP)/~800GFLOPS (DP)
140+ GB/s DRAM Bandwidth

ASCI Red – Sandia National Labs – 1997
NVIDIA Tesla C2050 Card Specs

- 448 GPU cores
- 1.15 GHz
- Single precision floating point performance: 1030.4 GFLOPs
  (2 single precision flops per clock per core)
- Double precision floating point performance: 515.2 GFLOPs
  (1 double precision flop per clock per core)
- Internal RAM: 3 GB DDR5
- Internal RAM speed: 144 GB/sec (compared 21-25 GB/sec for regular RAM)
- Has to be plugged into a PCIe slot (at most 8 GB/sec)
NVIDIA Tesla S2050 Server Specs

- 4 C2050 cards inside a 1U server
  (looks like a Sooner node)
- 1.15 GHz
- Single Precision (SP) floating point performance: 4121.6 GFLOPs
- Double Precision (DP) floating point performance: 2060.8 GFLOPs
- Internal RAM: 12 GB total (3 GB per GPU card)
- Internal RAM speed: 576 GB/sec aggregate
- Has to be plugged into two PCIe slots
  (at most 16 GB/sec)
## Compare x86 vs S2050

Let’s compare the best dual socket x86 server today vs S2050.

<table>
<thead>
<tr>
<th></th>
<th>Dual socket, AMD 2.3 GHz 12-core</th>
<th>NVIDIA Tesla S2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak DP FLOPs</td>
<td>220.8 GFLOPs DP</td>
<td>2060.8 GFLOPs DP (9.3x)</td>
</tr>
<tr>
<td>Peak SP FLOPS</td>
<td>441.6 GFLOPs SP</td>
<td>4121.6 GFLOPs SP (9.3x)</td>
</tr>
<tr>
<td>Peak RAM BW</td>
<td>25 GB/sec</td>
<td>576 GB/sec (23x)</td>
</tr>
<tr>
<td>Peak PCIe BW</td>
<td>N/A</td>
<td>16 GB/sec</td>
</tr>
<tr>
<td>Needs x86 server to attach to?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Power/Heat</td>
<td>~450 W</td>
<td>~900 W + ~400 W (~2.9x)</td>
</tr>
<tr>
<td>Code portable?</td>
<td>Yes</td>
<td>No (CUDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes (PGI, OpenCL)</td>
</tr>
</tbody>
</table>
Compare x86 vs S2050

Here are some interesting measures:

<table>
<thead>
<tr>
<th></th>
<th>Dual socket, AMD 2.3 GHz 12-core</th>
<th>NVIDIA Tesla S2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP GFLOPs/Watt</td>
<td>~0.5 GFLOPs/Watt</td>
<td>~1.6 GFLOPs/Watt (~3x)</td>
</tr>
<tr>
<td>SP GFLOPS/Watt</td>
<td>~1 GFLOPs/Watt</td>
<td>~3.2 GFLOPs/Watt (~3x)</td>
</tr>
<tr>
<td>DP GFLOPs/sq ft</td>
<td>~590 GFLOPs/sq ft</td>
<td>~2750 GFLOPs/sq ft (4.7x)</td>
</tr>
<tr>
<td>SP GFLOPs/sq ft</td>
<td>~1180 GFLOPs/sq ft</td>
<td>~5500 GFLOPs/sq ft (4.7x)</td>
</tr>
<tr>
<td>Racks per PFLOP DP</td>
<td>142 racks/PFLOP DP</td>
<td>32 racks/PFLOP DP (23%)</td>
</tr>
<tr>
<td>Racks per PFLOP SP</td>
<td>71 racks/PFLOP SP</td>
<td>16 racks/PFLOP SP (23%)</td>
</tr>
</tbody>
</table>

OU’s Sooner is 34.5 TFLOPs DP, which is just over 1 rack of S2050.
Previous GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
  - Essentially – re-write entire program as a collection of shaders and polygons
CUDA

“Compute Unified Device Architecture”

General purpose programming model
- User kicks off batches of threads on the GPU
- GPU = dedicated super-threaded, massively data parallel co-processor

Targeted software stack
- Compute oriented drivers, language, and tools

Driver for loading computation programs onto GPU
Parallel Computing on a GPU

- 400-series GPUs deliver 450 to 1,400+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters

- GPU parallelism is doubling every year
- Programming model scales transparently

- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism
Overview

- CUDA programming model
  - Basic concepts and data types

- CUDA application programming interface (API) basics

- A couple of simple examples

- Performance features will be covered this afternoon
CUDA Devices and Threads

- A CUDA compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device

- Data-parallel portions of an application are expressed as device kernels which run on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few (and is hurt by having too many)
CUDA – C with a Co-processor

- One program, two devices
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device kernel C code

Serial Code (host)

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

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
Extended C

Diagram:

- C CUDA Key Kernels
- Rest of C Application
- NVCC
- CPU Code
- CUDA object files
- CPU object files
- CPU-GPU Executable
- Linker
Buzzword: Kernel

- In CUDA, a kernel is code (typically a function) that can be run inside the GPU.

- The kernel code runs on the many stream processors in the GPU *in parallel*.
  - Each processor runs the code over different data (SPMD)
In CUDA, a thread is an execution of a kernel with a given index. Each thread uses its index to access a specific subset of the data, such that the collection of all threads cooperatively processes the entire data set.

Think: MPI Process ID

These are very much like threads in OpenMP

they even have shared and private variables.

So what’s the difference with CUDA?

Threads are free
Buzzword: Block

- In CUDA, a block is a group of threads.

- Blocks are used to organize threads into manageable chunks.
  - Can organize threads in 1D, 2D, or 3D arrangements
  - What best matches your data?
  - Some restrictions, based on hardware

- Threads within a block can do a bit of synchronization, if necessary.
Buzzword: Grid

- In CUDA, a grid is a group of blocks
  - no synchronization at all between the blocks.

- Grids are used to *organize* blocks into manageable chunks.
  - Can organize blocks in 1D or 2D arrangements
  - What best matches your data?

- A Grid is the set of threads created by a call to a CUDA kernel
Mapping Buzzwords to GPU Hardware

- **Grids** map to GPUs
- **Blocks** map to the MultiProcessors (MP)
  - Blocks are never split across MPs, but an MP can have multiple blocks
- **Threads** map to Stream Processors (SP)
- **Warps** are groups of (32) threads that execute simultaneously
  - Completely forget about these until later

Image Source: NVIDIA CUDA Programming Guide
Transparent Scalability

- Hardware is free to assign blocks to any SM (processor)
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - BlockIdx: 1D or 2D
  - ThreadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
CUDA Memory Model Overview

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- **We will focus on global memory for now**
  - Other memories will come later

Note: This is not hardware!
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device Global Memory
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object

- **cudaFree()**
  - Frees object from device Global Memory
    - Pointer to freed object
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to pointer named Md
  - “d” is often used in naming to indicate a device data structure

```c
TILE_WIDTH = 64;
float* Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);

cudaFree(Md);
```
The Physical Reality Behind CUDA

CPU (host)

GPU w/ local DRAM (device)
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Asynchronous transfer
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a $64 \times 64$ single precision float array
  - $M$ is in host memory and $Md$ is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are symbolic constants

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```
CUDA Kernel Template

- In C:
  ```c
  void foo(int a, float b)
  {
      // slow code goes here
  }
  ```

- In CUDA:
  ```c
  __global__ void foo(int a, float b)
  {
      // fast code goes here!
  }
  ```
Calling a Kernel Function

- A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block
KernelFunc(...); // invoke a function
```
A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);

dim3 DimGrid(100, 50);   // 5000 thread blocks
dim3 DimBlock(4, 8, 8);   // 256 threads per block

KernelFunc(...);         // invoke a function
```

Declare the dimensions for grid/blocks
A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);

dim3 DimGrid(100, 50);    // 5000 thread blocks

dim3 DimBlock(4, 8, 8);   // 256 threads per block

KernelFunc<<<DimGrid, DimBlock>>>(...);
    // invoke a kernel
```

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

Declare the dimensions for grid/blocks
C SAXPY

void saxpy_serial(int n, float a, float *x, float *y)
{
    int i;
    for(i=0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }
}

... //invoke the kernel
saxpy_serial(n, 2.0, x, y);
SAXPY on a GPU

- Doing anything across an entire vector is perfect for massively parallel computing.

- Instead of *one* function *looping* over the data set, we’ll use *many* threads, each doing *one* calculation.

\[
y[tid] = a \times x[tid] + y[tid];
\]
CUDA SAXPY

__global__ void
saxpy_cuda(int n, float a, float *x, float *y)
{
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;
    if(i < n)
        y[i] = a*x[i] + y[i];
}

...  

int nblocks = (n + 255) / 256;

//invoke the kernel with 256 threads per block
saxpy_cuda<<<nblocks, 256>>>(n, 2.0, x, y);
Matrix Multiplication in CUDA

A case study
Matrix Multiplication: A Case Study

- Matrix multiplication illustrates many of the basic features of memory and thread management in CUDA
  - Usage of thread/block IDs
  - Memory data transfer between host and device
  - Motivates some performance issues:
    - shared memory usage
    - register usage

- Assumptions:
  - Basic unoptimized sgemm
  - Matrices are square (for simplicity)
Programming Model: Square Matrix Multiplication Example

- \( P = M \times N \)
  - Each is of size WIDTH x WIDTH

- Basic Idea:
  - One thread calculates one element of \( P \)
    - \( M \) and \( N \) are loaded \( \text{WIDTH} \) times from global memory
Step 1: Matrix Multiplication
A Simple Host Version in C

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int WIDTH)
{
    int i, j, k;
    double a, b, sum;
    for (i = 0; i < WIDTH; ++i)
        for (j = 0; j < WIDTH; ++j) {
            sum = 0;
            for (k = 0; k < WIDTH; ++k) {
                a = M[i * WIDTH + k];
                b = N[k * WIDTH + j];
                sum += a * b;
            }
            P[i * WIDTH + j] = sum;
        }
}
Step 2: Input Matrix Data Transfer (Host-side Code)

```c
void MatrixMulOnDevice(float* M, float* N, float* P, int WIDTH) {
    int size = WIDTH * WIDTH * sizeof(float);
    float* Md, Nd, Pd;

    // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
}
```
Step 3: Output Matrix Data Transfer (Host-side Code)

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

// 3. Read P from the device
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

// Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
Step 4: Kernel Function

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int WIDTH) {
    float Pvalue = 0;

    for (int k = 0; k < WIDTH; ++k) {
        float Melement = Md[threadIdx.y*WIDTH+k];
        float Nelement = Nd[k*WIDTH+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    Pd[threadIdx.y*WIDTH+threadIdx.x] = Pvalue;
}
```
Step 5: Kernel Invocation (Host-side Code)

// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(WIDTH, WIDTH);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, WIDTH);
Only One Thread Block Used

- One Block of threads compute the matrix \( Pd \)
  - Each thread computes one element of the matrix \( Pd \)
- Each thread
  - Loads a row of matrix \( Md \)
  - Loads a column of matrix \( Nd \)
  - Perform one multiply and addition for each pair of \( Md \) and \( Nd \) elements
- Compute to off-chip memory access ratio close to 1:1 (not very good)
- Size of matrix limited by the number of threads allowed in a thread block (512)
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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

    Pd[Row*Width+Col] = Pvalue;
}

G80 Block Granularity Considerations

Q: 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 768 threads, there are 12 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 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

- For 32x32, we have 1024 threads per Block. Not even one can fit into an SM!
Exercise: Area Under the Curve

cp -r ~ernstdj/NCSI2010 .
go to “cuda_trap” directory.

less README.txt