ASCI A24: GPU COMPUTING

Rob van Nieuwpoort
rob@cs.vu.nl

The way it's meant to be played

nVidia

nRADEON

GRAPHICS
Schedule

- **Morning**
  - Introduction
  - GPU hardware
  - Cuda basics
  - Optimizing your Cuda program

- **Afternoon**
  - Cuda hand-on session
Who am I

- 10 years of Grid computing
- 5 years of many-core computing, radio astronomy
- Netherlands eScience center
  - Many-core solutions, astronomy, green computing
- ASTRON: Netherlands institute for radio astronomy
  - LOFAR
  - SKA
  - Many-core
- VU Amsterdam
  - CUDA teaching center
- Enhanced science
- Collaboration between cross-disciplinary researchers
- Apply ICT in the broadest sense
- Data-driven research across all scientific disciplines
- Develop generic Big Data tools
Novel instruments and applications produce lots of data

- LHC, telescopes, climate simulations, genetics, medical scanners, facebook, twitter, ...

- These applications cannot work without computer science anymore

- A lot of unexploited knowledge that we can only find if the data across disciplines is accessible and usable...

- Challenges: data handling, processing
  - Need large-scale parallelism
What are many-cores?
What are many-cores

- From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — largely because of issues with congestion in supplying instructions and data to the many processors.”
What are many-cores

- How many is many?
  - Several tens of cores

- How are they different from multi-core CPUs?
  - Non-uniform memory access (NUMA)
  - Private memories
  - Network-on-chip

- Examples
  - Multi-core CPUs (48-core AMD magny-cours)
  - Graphics Processing Units (GPUs)
    - GPGPU = general purpose programming on GPUs
  - Cell processor (PlayStation 3)
  - Server processors (Sun Niagara)
Why do we need many-cores?
Why do we need many-cores?
Why do we need many-cores?
Why do we need many-cores?
China's Tianhe-1A

#5 in top500 list

4.701 pflops peak
2.566 pflops max

14,336 Xeon X5670 processors
7168 Nvidia Tesla M2050 GPUs x 448 cores = 3,211,264 cores
Power efficiency

The World’s Most Efficient GPU

Outpacing Moore’s Law

GFLOPS/W

GFLOPS/mm²

0.42 1.07 2.01 2.21 4.50 4.56 7.90

Nov 05 Jan 06 Sep 07 Nov 07 Jan 06 Oct 09

ATI Radeon™ X1800 XT ATI Radeon™ X1900 XTX ATI Radeon™ HD 2900 PRO ATI Radeon™ HD 3870 ATI Radeon™ HD 4870 ATI Radeon™ HD 5870
Graphics in 1980
Graphics in 2000

You fragged Ares
2nd place with 28
Graphics now: Crysis 3
Realism of modern GPUs

ITV accused of using ArmA 2 game footage in IRA doc

Updated: WHOOPS

By Kate Solomon

ITV seems to have broadcast a documentary about Colonel Gaddafi's support for the IRA, passing gameplay action taken from ArmA 2 off as genuine documentary footage.

Update: ITV has now sent us a comment on the situation. A spokesman said, "The events featured in Exposure: Gaddafi and the IRA were genuine but it would appear that during the editing process the correct clip of the 1988 incident was not selected and other footage was mistakenly included in the film by producers. This was an unfortunate case of human error for which we apologise."

http://www.youtube.com/watch?v=bJDeipvpjGQ&feature=player_embedded#t=49s

Courtesy techradar.com
Why do we need many-cores?

- Performance
  - Large scale parallelism
- Power Efficiency
  - Use transistors more efficiently
- Price (GPUs)
  - Huge market, bigger than Hollywood
  - Mass production, economy of scale
  - “spotty teenagers” pay for our HPC needs!
How to cheat with speedups

How can this be? Core i7 CPU has 154 GFLOPs NVIDIA GTX 580 GPU has 1581 GFLOPs (10.3 X more)
How to cheat with speedups

- Heavily Optimize GPU version
  - Coalescing, Shared memory
  - Tiling, Loop unrolling
  - ...

- Do not optimize CPU version
  - 1 core only
  - No SSE
  - Cache unaware
  - No loop unrolling and tiling, ...

- Result: very high speedups!
- Exception: kernels that do interpolations (texturing hardware)
- Solution
  - Optimize CPU version
  - Use efficiencies: % of peak performance, Roofline
GPU hardware introduction
It's all about the memory
Integration into host system

- Typically PCI Express 2.0 x16
- Theoretical speed 8 GB/s
  - protocol overhead → 6 GB/s
- In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead
Lessons from Graphics Pipeline

- Throughput is paramount
  - must paint every pixel within frame time
  - scalability

- Create, run, & retire lots of threads very rapidly
  - measured 14.8 billion thread/s on increment() kernel

- Use multithreading to hide latency
  - 1 stalled thread is OK if 100 are ready to run
CPU vs GPU

- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA’s NVISION 2008
Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not

- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads
Chip area CPU vs GPU
Flynn’s taxonomy revisited

<table>
<thead>
<tr>
<th></th>
<th>Single Data</th>
<th>Multiple Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single instruction</td>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Multiple Instruction</td>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

GPUs don’t fit!
Key architectural Ideas

- Data parallel, like a vector machine
  - There, 1 thread issues parallel vector instructions

- SIMT (Single Instruction Multiple Thread) execution
  - Many threads work on a vector, each on a different element
  - They all execute the same instruction
  - HW automatically handles divergence

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
  - Context switching is (basically) free
ATI GPUs
Latest generation ATI

- Southern Islands
  - 1 chip: HD 7970
    - 2048 cores
    - 264 GB/sec memory bandwidth
    - 3.8 tflops single, 947 gflops double precision
    - Maximum power: 250 Watts
    - 399 euros!
  
- 2 chips: HD 7990
  - 4096 cores, 7.6 tflops

- Comparison: entire 72-node DAS-4 VU cluster has 4.4 tflops
ATI programming models

- Low-level: CAL (assembly)
- High-level: Brook+
  - Originally developed at Stanford University
  - Streaming language
  - Performance is not great
- Now: OpenCL
GPU Hardware: NVIDIA
Reading material

- Recommended reading:
  - CUDA: Compute Unified Device Architecture
Fermi

- Consumer: GTX 480, 580
- GPGPU: Tesla C2050
  - More memory, ECC
  - 1.0 teraflop single
  - 515 megaflop double
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
- 768 KB L2 cache
Fermi Streaming Multiprocessor (SM)

- 32 cores per SM (512 cores total)
- 64KB configurable L1 cache / shared memory
- 32,768 32-bit registers
Memory Hierarchy

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared

- Shared 768KB L2 cache
Multiple Memory Scopes

- Per-thread private memory
  - Each thread has its own local memory
  - Stacks, other private data
  - registers

- Per-SM shared memory
  - Small memory close to the processor, low latency

- Device memory
  - GPU frame buffer
  - Can be accessed by any thread in any SM
Atomic Operations

- Device memory is not coherent!

- Share data between streaming multiprocessors

- Read / Modify / Write

- Fermi increases atomic performance by 5x to 20x
  - Still, much slower than non-atomic access
ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache
- DRAM protected by ECC (on Tesla only)
- ECC is a must have for many computing applications
Programming NVIDIA GPUs
NVIDIA GPUs become more generic

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - Double precision floating point
  - C++

- Full integration in modern software development environment
  - Debugging
  - Profiling

- Bring more users, more applications to the GPU
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions

- Provide straightforward mapping onto hardware
  - Good fit to GPU architecture
  - Maps well to multi-core CPUs too

- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization
Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads
Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the kernel

- Threads are grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

- All thread blocks are organized in a Grid

- Threads/blocks have unique IDs
Grids, Thread Blocks and Threads

Grid

Thread Block 0, 0

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3

Thread Block 0, 1

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3

Thread Block 0, 2

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3

Thread Block 1, 0

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3

Thread Block 1, 1

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3

Thread Block 1, 2

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
2,0
2,1
2,2
2,3
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - Devices have
    - Different numbers of SMs
    - Different compute capabilities (Fermi = 2.0)
  - block is a virtualized streaming multiprocessor (threads, shared memory)
  - thread is a virtualized scalar processor (registers, PC, state)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent
Memory Spaces in CUDA

Host

Grid

Block (0, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Block (1, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Device Memory

Constant Memory
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Additional memories

- **Textures**
  - Read-only
  - Data resides in device memory
  - Different read path, includes specialized caches

- **Constant memory**
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Use when all threads in a block read the same address
    - Serializes otherwise
Host (CPU) manages device (GPU) memory:

- `cudaMalloc(void **pointer, size_t nbytes)`
- `cudaMemset(void *pointer, int val, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = n * sizeof(int);
int* data = 0;
cudaMalloc(&data, nbytes);
cudaMemset(data, 0, nbytes);
cudaFree(data);
```
Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete

- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking copies are also available
  - DMA transfers, overlap computation and communication
### CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int var;</code></td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><code>int array_var[10];</code></td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><code>__shared__ int shared_var;</code></td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><code>__device__ int global_var;</code></td>
<td>device</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><code>__constant__ int constant_var;</code></td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>
Philosophy: provide minimal set of extensions necessary

Function qualifiers:

```c
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

Execution configuration:

```c
dim3 gridDim(100, 50);  // 5000 thread blocks
dim3 blockDim(4, 8, 8);  // 256 threads per block (1.3M total)
my_kernel <<< gridDim, blockDim >>> (...);  // Launch kernel
```

Built-in variables and functions valid in device code:

```c
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx;  // Thread index
void syncthreads();  // Thread synchronization
“global” thread index:

\[
\text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\]
Calculating the global thread index

“global” thread index:

\[ \text{blockDim}.x \times \text{blockIdx}.x + \text{threadIdx}.x; \]

\[ 4 \times 2 + 1 = 9 \]
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
// compute vector sum \( c = a + b \)

// each thread performs one pair-wise addition

```c
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main() {
    // initialization code here ... 

    // launch N/256 blocks of 256 threads each
    vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);

    // cleanup code here ... 
}
```

(Host code)

(GPU code)
int main(int argc, char** argv) {
    int size = N * sizeof(float);

    // allocate host memory
    hostA = malloc(size);
    hostB = malloc(size);
    hostC = malloc(size);

    // initialize A, B arrays here...

    // allocate device memory
    cudaMemcpy(&deviceA, size);
    cudaMemcpy(&deviceB, size);
    cudaMemcpy(&deviceC, size);
// transfer the data from the host to the device
cudaMemcpy(deviceA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, size, cudaMemcpyHostToDevice);

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);

// transfer the result back from the GPU to the host
cudaMemcpy(hostC, deviceC, size, cudaMemcpyDeviceToHost);
CUDA: Scheduling, Synchronization and Atomics
Thread Scheduling

- Order in which thread blocks are scheduled is undefined!
  - any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially

- Order of threads within a block is also undefined!
Q: How do we do global synchronization with these scheduling semantics?
Global synchronization

Q: How do we do global synchronization with these scheduling semantics?
A1: Not possible!
Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!
Q: How do we do global synchronization with these scheduling semantics?

A1: Not possible!

A2: Finish a grid, and start a new one!

```
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

We don't have to copy the data back and forth!
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
  - Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Can be done on device memory and shared memory
- Much more expensive than load + operation + store
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter.

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter.

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically.

__global__ void histogram(int* colors, int* buckets) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
CUDA: optimizing your application

Shared Memory
Matrix multiplication example

- \( C = A \times B \)
- Each element \( C_{i,j} \)
  \[ = \text{dot}(\text{row}(A,i),\text{col}(B,j)) \]
- Parallelization strategy
  - Each thread computes element in \( C \)
  - 2D kernel
Matrix multiplication implementation

```c
__global__ void mat_mul(float *a, float *b,
    float *c, int width)
{
    // calc row & column index of output element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float result = 0;

    // do dot product between row of a and column of b
    for(int k = 0; k < width; k++) {
        result += a[row*width+k] * b[k*width+col];
    }
    c[row*width+col] = result;
}
```
# Matrix multiplication performance

<table>
<thead>
<tr>
<th>Loads per dot product term</th>
<th>2 (a and b) = 8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS</td>
<td>2 (multiply and add)</td>
</tr>
<tr>
<td>Arithmetic Intensity (AI)</td>
<td>2 / 8 = 0.25</td>
</tr>
<tr>
<td>Performance GTX 580</td>
<td>1581 GFLOPs</td>
</tr>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>192 * 0.25 = 48 GFLOPS</td>
</tr>
<tr>
<td>Maximum efficiency</td>
<td>3.0 % of theoretical peak</td>
</tr>
</tbody>
</table>
Data reuse

- Each input element in A and B is read WIDTH times
- Load elements into shared memory
- Have several threads use local version to reduce the memory bandwidth
Using shared memory

- Partition kernel loop into phases
- A thread block will compute a tile
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result
Matrix multiply with shared memory

```c
__global__ void mat_mul(float *a, float *b, float *c, int width) {

    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in shared memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & column index
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
```
Matrix multiply with shared memory

// loop over input tiles in phases
for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
        result += s_a[ty][k] * s_b[k][tx];
    }
    __syncthreads();
}

c[row*width+col] = result;
Use of Barriers in mat_mul

- Two barriers per phase:
  - `__syncthreads` after all data is loaded into shared memory
  - `__syncthreads` after all data is read from shared memory
  - Second `__syncthreads` in phase $p$ guards the load in phase $p+1$

- Use barriers to guard data
  - Guard against using uninitialized data
  - Guard against corrupting live data
Matrix multiplication performance

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global loads</td>
<td>$2N^3 \times 4 \text{ bytes}$</td>
<td>$(2N^3 / \text{TILE}_\text{WIDTH}) \times 4 \text{ bytes}$</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>Arithmetic Intensity (AI)</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE}_\text{WIDTH}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance GTX 580</th>
<th>1581 GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>AI needed for peak</td>
<td>$1581 / 192 = 8.23$</td>
</tr>
</tbody>
</table>
| TILE\_WIDTH required to achieve peak | $0.25 \times \text{TILE}\_\text{WIDTH} = 8.23,$  
|                         | $\text{TILE}\_\text{WIDTH} = 32.9$ |
A Common Programming Strategy

- Partition data into subsets that fit into shared memory
A Common Programming Strategy

- Handle each data subset with one thread block
A Common Programming Strategy

- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism
Perform the computation on the subset from shared memory
A Common Programming Strategy

- Copy the result from shared memory back to device memory
CUDA: optimizing your application

Coalescing
Coalescing

traditional multi-core optimal memory access pattern

thread 0
- $t = 0$ → address 0
- $t = 1$ → address 0

thread 1
- $t = 0$ → address 1
- $t = 1$ → address 2

thread 2
- $t = 0$ → address 4
- $t = 1$ → address 5

thread 3
- $t = 0$ → address 6
- $t = 1$ → address 7

many-core GPU optimal memory access pattern

thread 0
- $t = 0$ → address 0
- $t = 1$ → address 1

thread 1
- $t = 0$ → address 2
- $t = 1$ → address 3

thread 2
- $t = 0$ → address 4
- $t = 1$ → address 5

thread 3
- $t = 0$ → address 6
- $t = 1$ → address 7
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // Stride 1, OK!
    int a = input[i];

    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];

    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void***)&d_records, ...);
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
Example: SoA vs AoS

__global__ void bar(record* AoS_data, SoA SoA_data) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // AoS wastes bandwidth
    int key1 = AoS_data[i].key;

    // SoA efficient use of bandwidth
    int key2 = SoA_data.keys[i];
}
Memory Coalescing

- Structure of arrays is often better than array of structures
- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
- Can lose a factor of 10 – 30!
CUDA: optimizing your application

Optimizing Occupancy
Thread Scheduling

- SM implements zero-overhead warp scheduling
  - A warp is a group of 32 threads that runs concurrently on an SM
  - At any time, only one of the warps is executed by an SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - All threads in a warp execute the same instruction when selected

- Thread block sizes should be a multiple of the warp size

Instruction: 1 2 3 4 5 6 1 2 1 2 3 4 1 2 7 8 1 2 1 2 3 4

TB = Thread Block, W = Warp
Stalling warps

- What happens if all warps are stalled?
  - No instruction issued $\rightarrow$ performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy
Occupyancy

- What determines occupancy?
- Limited resources!
  - Register usage per thread
  - Shared memory per thread block
Pool of registers and shared memory per SM

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized → no more thread blocks
Resource Limits (2)

- Can only have 8 thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words

- **Y-axis**: GB/s
- **X-axis**: Threads Per Multiprocessor

The graph shows the throughput of 32-bit words in GB/s as the number of threads per multiprocessor increases. Initially, the throughput increases rapidly with the number of threads, reaching a peak before stabilizing.
How do you know what you’re using?

- Use “nvcc -Xptxas -v” to get register and shared memory usage

- Plug those numbers into CUDA Occupancy Calculator
1. Select Compute Capability (click): 1.3

2. Enter your resource usage:
   - Threads Per Block: 128
   - Registers Per Thread: 256
   - Shared Memory Per Block (bytes): 640

3. GPU Occupancy Data is displayed here and in the graphs:
   - Occupancy of each Multiprocessor: 50%
   - Multiprocessor Warp Occupancy: 128

4. Physical Limits for GPU Compute Capability: 1.3
   - Threads per Warp: 32
   - Warp per Multiprocessor: 32
   - Threads per Multiprocessor: 1024
   - Thread Blocks per Multiprocessor: 8
   - Total # of 32-bit registers per Multiprocessor: 16384
   - Register allocation unit size: 512
   - Register allocation granularity: block
   - Shared Memory per Multiprocessor (bytes): 16384
   - Shared Memory Allocation Unit size: 512
   - Warp allocation granularity (for register allocation): 2

5. Allocation Per Thread Block:
   - Warps: 4
   - Registers: 3684
   - Shared Memory: 1024

6. These data are used in computing the occupancy data in blue

7. Maximum Thread Blocks Per Multiprocessor Blocks
   - Limited by Max Warps / Blocks per Multiprocessor: 8
   - Limited by Registers per Multiprocessor: 4
   - Limited by Shared Memory per Multiprocessor: 16

8. Thread Block Limit Per Multiprocessor highlighted RED

9. CUDA Occupancy Calculator
   - Version: 2.0
   - Copyright and License
CUDA: optimizing your application

Shared memory bank conflicts
Shared Memory Banks

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value

- Consecutive words are in different banks

- If two or more threads access the same bank but different value, we get bank conflicts
Bank Addressing Examples: OK

- No Bank Conflicts

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

No Bank Conflicts

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15
Bank Addressing Examples: BAD

- 2-way Bank Conflicts
- 8-way Bank Conflicts
Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for shared memory writes
  - So, replace shared memory array indices with `threadIdx.x`
  - (Could also be done for the reads)
Generic programming models

OpenCL
Portability

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, Xeon Phi, CPUs

- Parallelism portability
  - Different architecture requires different granularity
  - Task vs data parallel

- Performance portability
  - Can we express platform-specific optimizations?
The Khronos group

Over 100 companies creating visual computing standards

Board of Promoters
OpenCL: Open Compute Language

- Architecture independent
- Explicit support for many-cores
- Low-level host API
  - Uses C library, no language extensions
- Separate high-level kernel language
  - Explicit support for vectorization
- Run-time compilation
- Architecture-dependent optimizations
  - Still needed
  - Possible
## Cuda vs OpenCL Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Device memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong></td>
<td>__kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>(no qualifier needed)</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constant</strong></td>
<td>__constant</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>__global</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>__local</td>
</tr>
</tbody>
</table>
Cuda vs OpenCL Indexing

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>get_num_groups()</td>
</tr>
<tr>
<td>blockDim</td>
<td>get_local_size()</td>
</tr>
<tr>
<td>blockIdx</td>
<td>get_group_id()</td>
</tr>
<tr>
<td>threadIdx</td>
<td>get_local_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_size()</td>
</tr>
</tbody>
</table>

__syncthreads() $\rightarrow$ barrier()
Vector add: Cuda vs OpenCL kernel

CUDA

```c
__global__ void vectorAdd(float* a, float* b, float* c) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

OpenCL

```c
__kernel void vectorAdd(__global float* a, __global float* b, __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0
};

cl_context context = clCreateContextFromType(props,
                                              CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1, 
    &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd",0);
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
   CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
// setup parameter values
clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
    &totalSize, &workGroupSize, 0, 0, 0); // execute kernel

// copy results from device back to host, blocking
clEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
    totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup
clReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
clReleaseMemObject(deviceC);
Summary and Conclusions
Summary and conclusions

- Higher performance cannot be reached by increasing clock frequencies anymore
- Solution: introduction of large-scale parallelism
- Multiple cores on a chip
  - Today:
  - Up to 48 CPU cores in a node
  - Up to 4096 cores on a single GPU
  - Host system can contain multiple GPUs: 10,000+ cores
  - We can build clusters of these nodes!
- Future: 100,000s – millions of cores?
Summary and conclusions

- Many different types of many-core hardware and software
- Very different properties
  - Performance, Programmability, Portability
- It's all about the memory
- Most models are hardware-induced, low-level
  - double buffering
  - Coalescing
  - Explicit cache (shared memory on GPU)
- Future
  - Cuda? OpenCL?
  - high-level models on top of OpenCL?
- Many-cores are here to stay
Hands-on session
Hostname: fs0.das4.cs.vu.nl

Add to .bashrc:

- module load cuda42/toolkit prun

Try nvcc --version

Everything is in: /home/rob/asci-a24-gpu

- Slides, handouts
- Cuda documentation
- Example programs
- Template for the assignment, input images

Use **display** to look at images
GPU hands-on session (2)

Run with:

```
prun -v -np 1 -native '-l gpu=GTX480' EXECUTABLE
```

Try, for instance:

```
$CUDA_SDK/C/bin/linux/release/deviceQuery
```

Check queue status:

```
preserve -long-list
```
GPU hands-on session (3)

- Vector add:
  - `/home/rob/asci-a24-gpu/vector-add`

- Assignment:
  - `/home/rob/asci-a24-gpu/sequential`

- CUDA template is already there
  - `/home/rob/asci-a24-gpu/cuda`