PARALLEL PROGRAMMING
MANY-CORE COMPUTING:
ADVANCED CUDA (4/5)
Schedule

1. Introduction, performance metrics & analysis
2. Many-core hardware, low-level optimizations
3. GPU hardware and Cuda class 1: basics
4. Cuda class 2: optimizations ; OpenCL
5. Case study: LOFAR telescope with many-cores
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
// compute vector sum \( \mathbf{c} = \mathbf{a} + \mathbf{b} \)
// 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];
}
```

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

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

    // cleanup code here ...
}
```
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?
Q: How do we do global synchronization with these scheduling semantics?

A1: Not possible!
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!

```c
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
// 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
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 + blockIdx.x * blockDim.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 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

Coalescing
Coalescing

traditional multi-core optimal memory access pattern

```
thread 0
  t = 0
  address 0
  t = 1
thread 1
  t = 0
  address 1
  t = 1
thread 2
  t = 0
  address 2
  t = 1
thread 3
  t = 0
  address 3
  t = 1
```

many-core GPU optimal memory access pattern

```
thread 0
  t = 0
  address 0
  t = 1
thread 1
  t = 0
  address 1
  t = 1
thread 2
  t = 0
  address 2
  t = 1
thread 3
  t = 0
  address 3
  t = 1
```
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;
}
```
struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void**)&d_records, ...);
Example: Structure of Arrays (SoA)

```c
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

```c
__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

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
__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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads per dot product term</td>
<td>2 (a and b) = 8 bytes</td>
</tr>
<tr>
<td>FLOPS</td>
<td>2 (multiply and add)</td>
</tr>
<tr>
<td>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>
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
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result

![Diagram showing shared memory usage](image)
__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;
Two barriers per phase:
- \texttt{syncthreads} after all data is loaded into shared memory
- \texttt{syncthreads} after all data is read from shared memory
- Second \texttt{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>$2N^3 \times 4$ bytes</td>
<td>$(2N^3 / \text{TILE_WIDTH}) \times 4$ bytes</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>AI</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE_WIDTH}$</td>
</tr>
</tbody>
</table>

Performance GTX 580            | 1581 GFLOPs               |
Memory bandwidth GTX 580        | 192 GB/s                  |
AI needed for peak             | $1581 / 192 = 8.23$       |
TILE\_WIDTH required to achieve peak | $0.25 \times \text{TILE\_WIDTH} = 8.23$,
                                          $\text{TILE\_WIDTH} = 32.9$ |
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
A Common Programming Strategy

- 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

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

TB1 W1
TB2 W1
TB3 W1
TB3 W2
TB2 W1
TB1 W1
TB1 W2
TB1 W3
TB3 W2

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
What determines occupancy?

Limited resources!

- Register usage per thread
- Shared memory per thread block
Pooling 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

GB/s

Threads Per Multiprocessor

0 128 256 384 512 640 768 896 1024
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
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
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 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

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

Bank 15
Bank 15
Bank 15

- No Bank Conflicts

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

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

Bank 15
Bank 15
Bank 15
Bank Addressing Examples: BAD

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 8
  - Thread 9
  - Thread 10
  - Thread 11
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15

- **8-way 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 7
  - Bank 8
  - Bank 9
  - Bank 15
  - x8
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>
## Cuda vs OpenCL Qualifiers

### 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() → barrier()
```
Vector add: Cuda vs OpenCL kernel

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

__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);
```c
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);

// 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);

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

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

delete[] A, B, C; // cleanup
clReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
cReleaseMemObject(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?
Many different types of many-core hardware

Very different properties
  - Performance
  - Programmability
  - Portability

It's all about the memory

Choose the right platform for your application
  - Arithmetic intensity / Operational intensity
  - Roofline model
Summary and conclusions

- Many different many-core programming models
- Most models are hardware-induced, low-level
  - DMA, double buffering
  - Vectorization
  - Coalescing
  - Explicit cache (LS on Cell, shared memory on GPU)
- Future
  - Cuda? OpenCL?
  - high-level models on top of OpenCL?
- Many-cores are here to stay