GPU programming: Code optimization

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Instruction-level optimization
Asynchronous execution

- By default, most CUDA function calls are *asynchronous*
  - Returns immediately to CPU code
  - GPU commands are queued and executed in-order
- Some commands are synchronous by default
  - cudaMemcpy(..., cudaMemcpyDeviceToHost)
  - Asynchronous version: cudaMemcpyAsync
- Keep it in mind when checking for errors and measuring timing!
  - Error returned by a command may be caused by an earlier command
  - Time taken by kernel<<<>>> launch is meaningless
- To force synchronization: cuThreadSynchronize()
Asynchronous transfers

- Overlap CPU work with GPU work

- Can we do better?
Streams: pipelining commands

- **Command queues** in OpenCL
  - Commands from the same stream run in-order
  - Commands from different streams run out-of-order
Streams: benefits

- Overlap CPU-GPU communication and computation: Direct Memory Access (DMA) copy engine runs CPU-GPU memory transfers in background
  - Requires page-locked memory
  - Some Tesla GPUs have 2 DMA engines: simultaneous send and receive

- Concurrent kernel execution
  - Start next kernel before previous kernel finishes
  - Reduces loss due to load imbalance

Example

Serial kernel execution

<table>
<thead>
<tr>
<th>a block 0</th>
<th>a 3</th>
<th>b 0</th>
<th>b 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 4</td>
<td>b 1</td>
<td></td>
</tr>
<tr>
<td>a 2</td>
<td></td>
<td>b 2</td>
<td></td>
</tr>
</tbody>
</table>

Concurrent kernel execution

<table>
<thead>
<tr>
<th>a block 0</th>
<th>a 3</th>
<th>b 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 4</td>
<td>b 1</td>
</tr>
<tr>
<td>a 2</td>
<td>b 0</td>
<td>b 3</td>
</tr>
</tbody>
</table>
Page-locked memory

- By default, allocated memory is *pageable*
  - Can be swapped out to disk, moved by the OS...
- DMA transfers are only safe on *page-locked* memory
  - Fixed virtual $\rightarrow$ physical mapping
  - `cudaMemcpy` needs an intermediate copy: slower, synchronous only
- `cudaMallocHost` allocates page-locked memory
  - Mandatory when using streams
- Warning: page-locked memory is a limited resource!
Streams: example

- Send data, execute, receive data

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```
Streams: alternative implementation

cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);

- Which one is better?
Events

- Schedule synchronization of one stream with another
  - Specifies dependencies between tasks

```c
cudaEvent_t e;
cudaEventCreate(&e);
kernel1<<<,,,a>>>();
cudaEventRecord(e, a);
cudaStreamWaitEvent(b, e);
kernel2<<<,,,b>>>();
cudaEventDestroy(e);
```

- Measure timing

```c
cudaEventRecord(start, 0);
kernlel<<<>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```
NVIDIA Compute capabilities

- Newer GPUs introduce additional features

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>G92</th>
<th>GT200</th>
<th>GT21x</th>
<th>GF100</th>
<th>GF104</th>
<th>GK104</th>
<th>GK110</th>
<th>GM107</th>
<th>GM204</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td>5.0</td>
<td>5.2</td>
</tr>
</tbody>
</table>

- Compute capability means both
  - Set of features supported
    Who can do more can do less: \( x > y \) \( \rightarrow \) CC \( x \) includes CC \( y \)
  - Native instruction set
    Not always backward-compatible
    e.g. GPU of CC 2.0 cannot run binary for CC 1.3
Compiler targets

- Compiler flags: `--generate-code arch=<arch>, code=<code>, ...`
  - `arch=CC`: directs PTX generation
    - my code requires features of CC
  - `code=CC`: directs native code gen.
    - generate code for GPU CC
  - Multiple targets possible

- CC can be
  - `compute_xx` for PTX
  - `sm_xx` for native

- Example

```
nvcc --generate-code arch=compute_10, code=sm_10
  --generate-code arch=compute_11, code='sm_12,sm_13'
  -o hello hello.cu
```
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Instruction-level optimization
Memory access patterns

In traditional vector processing

<table>
<thead>
<tr>
<th>Easy</th>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tₙ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td>Registers</td>
<td>Memory</td>
</tr>
<tr>
<td>T₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tₙ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Scalar load & broadcast
- Reduction & scalar store

<table>
<thead>
<tr>
<th>Hard</th>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tₙ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hard</td>
<td>Registers</td>
<td>Memory</td>
</tr>
<tr>
<td>T₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tₙ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (Non-unit) strided load
- (Non-unit) strided store

On GPUs

- Every load is a gather, every store is a scatter
Breakdown of memory access patterns

- Vast majority: uniform or unit-strided
  - And even aligned vectors

“In making a design trade-off, favor the frequent case over the infrequent case.”
Memory coalescing

- In hardware: compare the address of each vector element
- Coalesce memory accesses that fall within the same segment

- Dynamically detects parallel memory regularity
Coalescing concurrent requests

- **Unit-strided detection (NVIDIA CC 1.0-1.1 coalescing)**
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

- **Minimal coverage (NVIDIA CC 1.2 coalescing)**
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

Impact on work distribution, data structures?
Consequences: threading granularity

- **Coarse-grained threading**
  - **Decouple** tasks to reduce conflicts and inter-thread communication
  - e.g. MPI, OpenMP

- **Fine-grained threading**
  - **Interleave** tasks
  - Exhibit **locality**: neighbor threads share memory
  - Exhibit **regularity**: neighbor threads have a similar behavior
  - e.g. CUDA, OpenCL
Array of structures (AoS)

- Programmer-friendly memory layout
  - Group data logically
- Memory accesses not coalesced
  - Bad performance on GPU

```
struct Pixel {
    float r, g, b;
};
Pixel image_AoS[480][640];

kernel void luminance(Pixel img[][], float luma[][]) {
    int x=tid.x; int y=tid.y;
    luma[y][x]=.59*img[y][x].r
             + .11*img[y][x].g
             + .30*img[y][x].b;
}
```

- Need to rethink data structures for fine-grained threading
Structure of Arrays (SoA)

- Transpose the data structure
  - Group together similar data for different threads
- Benefits from memory coalescing
  - Best performance on GPU

```c
struct Image {
    float R[480][640];
    float G[480][640];
    float B[480][640];
};
Image image_SoA;

kernel void luminance(Image img, float luma[16][16]) {
    int x=tid.x; int y=tid.y;
    luma[y][x]=.59*img.R[y][x] + .11*img.G[y][x] + .30*img.B[y][x];
}
```
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Instruction-level optimization
Vector loads

- We can load more data at once with vector types
  - float2, float4, int2, int4...
  - More memory parallelism
  - Allows to reach peak throughput with fewer threads

Multiple outstanding loads

- Multiple independent loads from the same thread can be pipelined
  - More memory parallelism
  - Peak throughput with yet fewer threads

```c
__global__ void luminance(Image img, float luma[][[]]) {
    int x=threadIdx.x, y=threadIdx.y;
    luma[y][x]=.59*img.R[y][x]
        + .11*img.G[y][x]
        + .30*img.B[y][x];
}
```
Buffer accesses through shared memory

- Global memory accesses are the most expensive
  - Focus on optimizing global memory accesses

- Strategy: use shared memory as a temporary buffer

1. Load with regular accesses
2. Read and write shared memory with original pattern
3. Store back to global memory with regular accesses
Example: matrix transpose

- $B = A^T$

- Naive algorithms
  - Option 1
    Thread $i,j$:
    $B[j,i] = A[i,j]$
  
  - Option 2
    Thread $i,j$:
    $B[i,j] = A[j,i]$

- Which one is better?
  - What is the problem?
Example: matrix transpose

- $B = A^T$

- Naive algorithms
  - Option 1
    - Coalesced
  - Option 2
    - Non-coalesced

- Both are equally bad
  - Access to one array is non-coalesced
Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back

Example with 16×16 blocks

Block bx, by, Thread tx, ty:

\[
a[ty,tx]=A[by\times16+ty,by\times16+tx]
\]

Sync threads

\[
b[ty,tx]=a[tx,ty]
\]

Sync threads

\[
B[by\times16+ty,bx\times16+tx]=b[ty,tx]
\]

Coalesced
Objection

- Isn't it just moving the problem to shared memory?
- Yes: shared memory has access restrictions too
- But
  - Shared memory is much faster, even for irregular accesses
  - We can optimize shared memory access patterns too
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities

- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication

- Instruction-level optimization
Shared memory: banked

- Inside each SM, shared memory is distributed between multiple banks
  - 16 or 32 banks
Shared memory bank assignment

- Interleaved on a word-by-word basis: Modulo placement of data

![Diagram showing memory bank assignment](image_url)
Shared memory: the good

- Threads access contiguous locations: no conflict
  - All threads can be served concurrently
Shared memory: the bad

- Threads access random locations: some conflicts
  - Some threads have to wait for a bank
Shared memory: the ugly

- Threads access locations spaced by 16: systematic conflict
  - All threads have to wait for the same bank
Example: matrix transpose

- Where are bank conflicts?

Block bx, by, Thread tx, ty:

\[ a[ty*16+tx] = A[by*16+ty, by*16+tx] \]

Syncthreads

\[ b[ty*16+tx] = a[tx*16+ty] \]

Syncthreads

\[ B[by*16+ty, bx*16+tx] = b[ty*16+tx] \]
Example: matrix transpose

**Where are bank conflicts?**

Block $b_x, b_y$, Thread $t_x, t_y$:
- $a[ty*16+tx]=A[by*16+ty,by*16+tx]$  
  Syncthreads
- $b[ty*16+tx]=a[tx*16+ty]$  
  Syncthreads
- $B[by*16+ty,bx*16+tx]=b[ty*16+tx]$

**How to avoid them?**
Remapping data

- Solution 1: pad with empty cells

Block bx, by, Thread tx, ty:
\[ a[ty*17+tx] = A[by*16+ty, by*16+tx] \]
Sync threads
\[ b[ty*16+tx] = a[tx*17+ty] \]
Sync threads
\[ B[by*16+ty, bx*16+tx] = b[ty*17+tx] \]

- No bank conflicts
- Memory overhead
Remapping data

- Solution 2: different mapping function
  - Example: map \([y,x]\) to \(y \times 16 + (x + y \mod 16)\)
  - Or \(y \times 16 + (x \land y)\)

Block \(bx, by, Thread tx, ty: \)
\[
a[ty\times16+(tx+ty)\mod16]=A[by\times16+ty, by\times16+tx]
\]
Syncthreads
\[
b[ty\times16+tx]=a[tx\times16+(ty+tx)\mod16]
\]
Syncthreads
\[
B[by\times16+ty, bx\times16+tx]=b[ty\times17+tx]
\]

- No bank conflicts
- No memory overhead
Recap

- Overlap long-latency communications with computations
- Avoid global accesses when you can
  - Reuse data to get enough arithmetic intensity
  - Use registers and shared memory whenever possible
- Make consecutive threads access contiguous data
  - Stage data in shared memory if needed
- Avoid bank conflicts in shared memory
- Express locality and regularity
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Instruction-level optimization
Example: back to matrix multiplication

- Run loops on $i, j, i_2, j_2$ in parallel

```cpp
for // $i = 0$ to $n-1$ step 16
    for // $j = 0$ to $n-1$ step 16
        c = \{0\}
        for $k = 0$ to $n-1$ step 16
            a = A[i..i+15,k..k+15]
            b = B[k..k+15,j..j+15]
            for $k_2 = 0$ to 15
                for // $i_2 = 0$ to 15
                    for // $j_2 = 0$ to 15
                        c[i2,j2] += a[i2,k2]*b[k2,j2]
            C[i..i+15,j..j+15] = c
```

- Let's focus on threads
Level 1: SIMD (PRAM-style) version

- Each processor has ID (x,y)
  - Loops on i2, j2 are implicit

\[
\begin{align*}
c[x,y] &= 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} & \\
a[x,y] &= A[i+x,k+y] \\
b[x,y] &= B[k+x,j+y] \\
\text{Load submatrices a and b} \\
\text{for } k2 = 0 \text{ to } 15 & \\
c[x,y] &+= a[x,k2] \times b[k2,y] \\
\text{Multiply submatrices c = a \times b} \\
C[i+x,j+y] &= c[x,y] \\
\text{Store submatrix c} \\
\text{Private writes: no conflict} \\
\text{Read from other processors}
\end{align*}
\]

- How to translate to SPMD (BSP-style)?
SPMD version

- Place synchronization barriers

\[
c[x,y] = 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
a[x,y] = A[i+x,k+y] \\
b[x,y] = B[k+x,j+y] \\
\textbf{Barrier} \\
\text{for } k2 = 0 \text{ to } 15 \\
\quad c[x,y] += a[x,k2]*b[k2,y] \\
\textbf{Barrier} \\
C[i+x,j+y] = c[x,y]
\]

- Why do we need the second barrier?
Data allocation

- 3 memory spaces: **Global, Shared, Local**

  - Where should we put: A, B, C, a, b, c?

```plaintext
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```
Data allocation

- Memory spaces: **Global**, **Shared**, **Local**
  - As local as possible

\[
\begin{align*}
  c &= 0 \\
  \text{for } k &= 0 \text{ to } n-1 \text{ step 16} \\
  a[x,y] &= A[i+x, k+y] \\
  b[x,y] &= B[k+x, j+y] \\
  \text{Barrier} \\
  \text{for } k2 &= 0 \text{ to } 15 \\
  c &= c + a[x, k2] \times b[k2, y] \\
  \text{Barrier} \\
  C[i+x, j+y] &= c
\end{align*}
\]

**Local**: private to each thread (indices are implicit)

**Global**: shared between blocks / inputs and outputs

**Shared**: shared between threads, private to block
CUDA version

- Straightforward translation

```c
float Csub = 0;

for(int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
    for(int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

matrixMul example: cuda/samples/0_Simple/matrixMul
Memory access patterns

- On a block of 256 threads

  \[
  \begin{array}{cccccccc}
  T0 & T1 & T2 & T3 & T16 & T17 & \cdots & T255 \\
  x & 0 & 1 & 2 & 3 & 0 & \cdots & 15 \\
  y & 0 & 0 & 0 & 0 & 0 & \cdots & 15 \\
  \end{array}
  \]

  \[
c = 0 \\
  \text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
  a[x,y] = A[i+x,k+y] \\
  b[x,y] = B[k+x,j+y] \\
  \text{Barrier} \\
  \text{for } k2 = 0 \text{ to } 15 \\
  c += a[x,k2] * b[k2,y] \\
  \text{Barrier} \\
  C[i+x,j+y] = c
  \]

- Which accesses are coalesced?
- Are there bank conflicts?
Memory access patterns

- On a block of 256 threads

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T16</th>
<th>T17</th>
<th>T255</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

\[ c = 0 \]
for \( k = 0 \) to \( n-1 \) step 16
\[ a[x,y] = A[i+x,k+y] \]
\[ b[x,y] = B[k+x,j+y] \]
Barrier
for \( k2 = 0 \) to 15
\[ c += a[x,k2] * b[k2,y] \]
Barrier
\[ C[i+x,j+y] = c \]
- No coalesced access
- Massive bank conflicts

\[
c = 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
a[x,y] = A[i+x,k+y] \\
b[x,y] = B[k+x,j+y] \\
\text{Barrier} \\
\text{for } k2 = 0 \text{ to } 15 \\
c += a[x,k2] \times b[k2,y] \\
\text{Barrier} \\
C[i+x,j+y] = c
\]

- Can we improve it?
Memory optimization

- Exchange x and y

```plaintext
c = 0
for k = 0 to n-1 step 16
    a[y,x] = A[i+y,k+x]
    b[y,x] = B[k+y,j+x]
    Barrier
    for k2 = 0 to 15
        c += a[y,k2]*b[k2,x]
    Barrier
C[i+y,j+x] = c
```

- Success!
- Now can we improve memory parallelism?
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Workload partitioning
- Instruction-level optimization
Workload partitioning

How to choose grid dimensions?

- **Number of blocks per grid**
  - Linear with data size, or constant
  - Min: at least number of SMs * blocks per SM
  - No max in practice

- **Number of threads per block**
  - Constant: should not depend on dataset size
  - Max: hardware limitation, 512 or 1024 threads
  - Min: size of a warp: 32 threads

- **Iterations per thread**
  - Constant or variable
  - Min: enough to amortize thread creation overhead
  - No max, but shorter-lived threads reduce load imbalance
Multiple grid/block dimensions

- Grid and block size are of type dim3
  - Support up to 3 dimensions
    
    ```
    dim3 dimBlock(tx, ty, tz);
    dim3 dimGrid(bx, by, bz);
    my_kernel<<<dimGrid, dimBlock>>>(arguments);
    ```

  - Implicit cast from int to dim3
    y and z sizes are 1 by default

- On device side, threadIdx, blockDim, blockIdx, gridDim are also of type dim3
  - Access members with .x, .y, .z

- Some early GPUs (CC 1.x) only support 2-D grids
Occupancy metric

- Threads per SM / max threads per SM
- Resource usage may cause non-ideal occupancy
  - Register usage
  - Shared memory usage
  - Non-dividable block size

Available shared memory: 16KB
Usage: 12KB/block
→ Only 1 block / SM

Available registers: 32768
Usage: 64 registers/thread, blocks of 256 threads
→ Only 2 blocks / SM

Max threads/SM: 768 threads
Block size: 512 threads
→ Only 1 block / SM
Could run 3 blocks of 256 threads
GPU: on-chip memory

- Conventional wisdom
  - Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:

- But... if we include registers:

<table>
<thead>
<tr>
<th>GPU</th>
<th>Register files + caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GM204 GPU</td>
<td>8.3 MB</td>
</tr>
<tr>
<td>AMD Hawaii GPU</td>
<td>15.8 MB</td>
</tr>
<tr>
<td>Intel Core i7 CPU</td>
<td>9.3 MB</td>
</tr>
</tbody>
</table>

- GPU/accelerator internal memory exceeds desktop CPUs
How many threads?

- As many as possible (maximize occupancy)?
  - Maximal data-parallelism
    - Latency hiding
  - Locality
    - Store private data of each thread
  - Thread management overhead
    - Initialization, redundant operations

- Trade-off between parallelism and memory locality
Multiple elements per thread

- Block size \((16, 16) \rightarrow (8, 16)\)
- 2 elements per thread: \((x, y)\) and \((x+8, y)\)

\[
\begin{align*}
c[0] &= 0 \\
c[1] &= 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step } 16 \\
a[y,x] &= A[i+y,k+x] \\
b[y,x] &= B[k+y,j+x] \\
a[y+8,x] &= A[i+y+8,k+x] \\
b[y+8,x] &= B[k+y+8,j+x] \\
\text{Barrier} \\
\text{for } k2 = 0 \text{ to } 15 \\
c[0] &=+ a[y,k2]\cdot b[k2,x] \\
c[1] &=+ a[y+8,k2]\cdot b[k2,x] \\
\text{Barrier} \\
C[i+y,j+x] &= c[0] \\
C[i+y+8,j+x] &= c[1]
\end{align*}
\]

- What about shared memory?
Data reuse

- Share reads to submatrix b
  - Fewer shared memory accesses
  - Exchange data through registers

\[
\begin{align*}
  c[0] &= 0 \\
  c[1] &= 0 \\
  \text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
  a[y,x] &= A[i+y,k+x] \\
  b[y,x] &= B[k+y,j+x] \\
  a[y+8,x] &= A[i+y+8,k+x] \\
  b[y+8,x] &= B[k+y+8,j+x] \\
  \text{Barrier} \\
  \text{for } k2 = 0 \text{ to } 15 \\
  bl &= b[k2,x] \\
  c[0] &= a[y,k2] * bl \\
  c[1] &= a[y+8,k2] * bl \\
  \text{Barrier} \\
  C[i+y,j+x] &= c[0] \\
  C[i+y+8,j+x] &= c[1]
\end{align*}
\]

- Improves register usage too. Why?
True story: SGEMM from CUBLAS 1.1


- 512 threads / CTA, 15 registers / thread
- 9 registers / 15 contain redundant data
- Only 2 registers really needed

 Addresses, indices (linear increase)  

 Duplicated data  

 Useful data  

 Temporary data  

 15 registers / thread  

 512 threads / CTA
Fewer threads, more computations

- New SGEMM in CUBLAS 2.0
  - 8 elements computed / thread
  - Unrolled loops
  - Less traffic through shared memory, more through registers

- Overhead amortized
  - 1920 registers vs. 7680 for the same amount of work
  - Works for redundant computations too

- Instruction-level parallelism is still relevant

Diagram:
- Adresses, indices
- Useful data
- Duplicated data
- Temporary data

- 30 registers / thread
- 64 threads / CTA
Re-expressing parallelism

- Converting types of parallelism
  - ILP
  - TLP
  - DLP

- General strategy
  - Design phase: focus on thread-level parallelism
  - Optimization phase: convert TLP to Instruction-level or Data-level parallelism
Outline

- Host-side task and memory management
  - Asynchronism and streams
  - Compute capabilities
- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Back to matrix multiplication
- Workload partitioning
- Instruction-level optimization
Device functions

- Kernel can call functions
- Need to be marked for GPU compilation

```c
__device__ int foo(int i) {
}
```

- A function can be compiled for both host and device

```c
__host__ __device__ int bar(int i) {
}
```

- Device functions can call device functions
  - Older GPUs do not support recursion
Local memory

- Registers are fast but
  - Limited in size
  - Not addressable

- Local memory used for
  - Local variables that do not fit in registers (*register spilling*)
  - Local arrays accessed with indirection

```c
int a[17];
b = a[i];
```

- **Warning**: local is a misnomer!
  - Physically, local memory usually goes off-chip
  - About same performance as coalesced access to global memory
Loop unrolling

- Can improve performance
  - Amortizes loop overhead over several iterations
  - May allow constant propagation, common sub-expression elimination...

- Unrolling is **necessary** to keep arrays in registers

  Not unrolled
  ```c
  int a[4];
  for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
  }
  ```

  Unrolled
  ```c
  int a[4];
  a[0] = 3 * 0;
  a[1] = 3 * 1;
  a[2] = 3 * 2;
  a[3] = 3 * 3;
  ```

  Indirect addressing:
  a in local memory

  Static addressing:
  a in registers

- The compiler can unroll for you

  ```c
  #pragma unroll
  for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
  }
  ```
Warp-based execution

- Threads in a warp run in lockstep
- On NVIDIA architectures, warp is 32 threads
- A block is made of warps (warps do not cross block boundaries)
  - Block size multiple of 32 for best performance
Branch divergence

- Conditional block
  ```c
  if(c) {
    // A
  }
  else {
    // B
  }
  ```

- All threads of a warp take the same path

With imaginary 4-thread warps
Branch divergence

- Conditional block
  
  ```
  if(c) {
    // A
  }
  else {
    // B
  }
  ```

- Threads in a warp take different paths

- Warps have to go through both A and B: lower performance
Avoiding branch divergence

- Hoist identical computations and memory accesses outside conditional blocks

```c
if(tid % 2) {
    s += 1.0f/tid;
} else {
    s -= 1.0f/tid;
}
```

```c
float t = 1.0f/tid;
if(tid % 2) {
    s += t;
} else {
    s -= t;
}
```

- When possible, re-schedule work to make non-divergent warps

```c
// Compute 2 values per thread
int i = 2 * tid;
s += 1.0f/i - 1.0f/(i+1);
```

- What if I use C's ternary operator (?:) instead of if? (or tricks like ANDing with a mask, multiplying by a boolean...)
Ternary operator ? good : bad

- Run both branches and select: \[ R = c \ ? \ A \ : \ B; \]
  - No more divergence?

- All threads have to take both paths
  - No matter whether the condition is divergent or not

```
Warp 0
A
B
Warp 1
A
B
```

- Does **not** solve divergence: we lose in all cases!
- Only benefit: fewer instructions
  - May be faster for short, often-divergent branches
- Compiler will choose automatically when to use predication
  - Advice: keep the code readable, let the compiler optimize
Recap

- Beware of local arrays
  use static indices and loop unrolling
- Keep in mind branch divergence when writing algorithm
  but do not end up in managing divergence yourself
Takeaway

- Distribute work and data
  - Favor SoA
  - Favor locality and regularity
  - Use common sense (avoid extraneous copies or indirections)

- More threads ≠ higher performance
  - Saturate instruction-level parallelism first (almost free)
  - Complete with data parallelism (expensive in terms of locality)

- Usual advice applies
  - First write correct code
  - Profile
  - Optimize
  - Repeat
References and further reading/watching

- CUDA C Programming Guide