Advanced GPU Programming

Samuli Laine
NVIDIA Research
Today

**Code execution on GPU**
- High-level GPU architecture
- SIMT execution model
- Warp-wide programming techniques

**GPU memory system**
- Estimating the cost of a memory access
- Vector types, atomic operations, constant memory

**Asynchronous execution**
- Streams and events
Code execution on GPU
SM = Streaming Multiprocessor

The workhorse in NVIDIA’s GPUs
- Sometimes called SMX, SMM, etc. in marketing

Executes arbitrary code
- Including shaders when doing graphics

Capabilities and SM count depend on architecture and board
- Tesla K2000 (Kepler): 2 SMs (Maari-A)
- GTX Titan X (Maxwell): 24 SMs
- Tesla P100 (Pascal): 56 SMs – but half size
P100 block diagram
Pascal SM

- Scheduler
- Register file
- Single-precision ALUs
- Double-precision ALUs
- L1 cache
- Shared memory
Warps

- Threads are executed in **warps**
  - Warp contains up to **32** threads

- Thread block occupies a number of warps in the same SM
  - E.g., block of **128** threads will be packed into **4** warps

- SM operates at warp granularity
  - Resource allocation
  - Execution
Warps, example

Block of 100 threads
- **blockDim.x = 100**

| Lane index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Warp A     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| Warp B     | 32| 33| 34| 35| 36| 37| 38| 39| 40| 41| 42| 43| 44| 45| 46| 47| 48| 49| 50| 51| 52| 53| 54| 55| 56| 57| 58| 59| 60| 61| 62| 63 |
| Warp C     | 64| 65| 66| 67| 68| 69| 70| 71| 72| 73| 74| 75| 76| 77| 78| 79| 80| 81| 82| 83| 84| 85| 86| 87| 88| 89| 90| 91| 92| 93| 94| 95 |
| Warp D     | 96| 97| 98| 99|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Note: If **blockDim.x == 32**, then **threadIdx.x** will give the lane index
SM view of warps

SM has a pool of resident warps

- Each warp has their own resources

Warp 0: Thread block 10, warp A
Warp 1: Thread block 15, warp B
Warp 2: Thread block 10, warp C
Warp 3: Thread block 10, warp D
Warp 4: Thread block 67, warp C
Warp 5: Thread block 15, warp D
Warp 6: Thread block 15, warp C
Warp 7: Thread block 24, warp A
Warp 8: empty
Warp 9: Thread block 15, warp A
Warp 10: Thread block 15, warp B
Warp 11: Thread block 10, warp B
Warp 12: Thread block 21, warp C
Warp n: Thread block 67, warp A

Register file
Shared memory
Maximum **occupancy**, i.e., number of warps in SM can be limited by various resources

- Register file
- Shared memory
- Number of warp slots (64)
- Number of thread block IDs (16 or 32)

Warps are created when SM has enough resources to spawn an **entire thread block**

Warp terminates when **all of its threads** have terminated

- Resources stay allocated until that
Warp scheduling

At every cycle, SM chooses which warp to execute
- Actually two warps per cycle in current architectures
- Zero overhead in “switching” between warps or threads

Warp is eligible to be executed if all of its threads are free to execute
- Not waiting for memory fetches
- Not waiting for results from ALUs
- Not waiting in a `__syncthreads()` call
Occupancy and latency hiding

- Having a large number of warps is how GPU hides latencies

- Anything that limits occupancy is bad for latency hiding
  - Too high register usage
  - Too high shared memory usage
  - Too small thread blocks → cannot utilize all warps

- Kernel’s occupancy can be queried using CUDA API
  - Or calculated using the “CUDA Occupancy Calculator” spreadsheet
Program counter (PC)

- All threads in a warp have the same PC
  - i.e., they execute the same instruction on a given cycle
**SIMT execution model**

- **How is this possible?**
  - Sounds like SIMD, but how can threads be independent?

  **SIMT = Single Instruction, Multiple Threads**
  - Close to SIMD, but allows free per-thread control flow

- **Built into SM instructions and scheduler**
  - Dedicated hardware is necessary for efficient implementation
SIMT vs SIMD

**SIMD** (Single Instruction Multiple Data)
- Used in CPUs, e.g. Intel’s SSE/AVX extensions
- Programmer sees a scalar thread with access to a wide ALU
  - For example, able to do 4 or 8 additions with a single instruction

**SIMT** (Single Instruction Multiple Thread)
- Programmer sees independent scalar threads with scalar ALUs
- Hardware internally converts independent control flow into convergent control flow
Managing divergence

- How can threads of a warp diverge if they all have the same PC?
  - Partial solution: Per-instruction execution predication
  - Full solution: Execution mask, execution stack in hardware
if (a < 10)
    small++;
else
    big++;
Example: Instruction predication

if (a < 10)
    small++;
else
    big++;
Example: Instruction predication

```c
if (a < 10)
    small++;  // Example
else
    big++;    // Example
```

```
ISETP.LT.AND P0, pt, R6, 10, pt;
@P0  IADD R5, R5, 0x1;
@!P0 IADD R4, R4, 0x1;
```

In threads where \( P0 \) is set, \( R5 = R5 + 1 \)
Example: Instruction predication

if (a < 10)  
    small++;  
else  
    big++;  

ISETP.LT.AND P0, pt, R6, 10, pt;
@P0 IADD R5, R5, 0x1;
@!P0 IADD R4, R4, 0x1;

In threads where P0 is clear, R4 = R4 + 1
What about complex cases?

- Nested if/else blocks, loops, recursion ...

- Solution: Execution mask and execution stack
if (a < 10)
    foo();
else
    bar();

/*0048*/     ISETP.LT.AND P0, pt, R6, 10, pt;
/*0050*/     @!P0 BRA 0x70;
/*0058*/     ...;
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;
/*0078*/     ...;
/*0080*/     code continues here
**Execution mask & stack: Example**

**Case 1:** All threads take the `if` branch

```c
if (a < 10)
    foo();
else
    bar();
```

```c
/*0048*/     ISETP.LT.AND P0, pt, R6, 10, pt;
/*0050*/     @!P0 BRA 0x70; // no thread of the warp wants to jump
/*0058*/     ...;   foo()
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;   bar()
/*0078*/     ...;
/*0080*/     code continues here
```
Execution mask & stack: Example

Case 2: All threads take the `else` branch

```c
if (a < 10)  
    foo();  
else  
    bar();
```

```c
/*0048*/  ISETP.LT.AND P0, pt, R6, 10, pt;
/*0050*/  @!P0 BRA 0x70;  // all threads of the warp want to jump
/*0058*/  ...;  foo();  
/*0060*/  ...;
/*0068*/  BRA 0x80;
/*0070*/  ...;  bar();  
/*0078*/  ...;
/*0080*/  code continues here
```
Case 3: Some threads take the \texttt{if} branch, some take the \texttt{else} branch

```c
if (a < 10)
    foo();
else
    bar();
```

```c
/*0048*/     ISETP.LT.AND P0, pt, R6, 10, pt;
/*0050*/     @!P0 BRA 0x70;
/*0058*/     ...;
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;
/*0078*/     ...;
/*0078*/     bar();
/*0080*/     code continues here
```
Benefits of SIMT

- Supports all structured C++ constructs
  - `if` / `else`, `switch` / `case`, loops, function calls, exceptions
  - `goto` is an abomination – supported, but best to avoid

- Multi-level constructs handled efficiently
  - `break` / `continue` from inside multiple levels of conditionals
  - Function return from inside loops and conditionals
  - Retreating to exception handler from anywhere

- You only need to care about SIMT when tuning for performance
  - Unlike traditional SIMD that gives you nothing unless you explicitly use it
Consequences of SIMT

- An `if` statement takes the same number of cycles for any number of threads greater than zero
  - If nobody participates it’s cheap
  - Also, masked-out threads don’t do memory accesses

- A loop is iterated until all active threads in the warp are done

- A warp stays alive until every thread in it has terminated
  - Terminated threads are dead weight
  - Same as in conditionals when masked out
Straight code tends to be better than branchy

```c
int median5(int v1, int v2, int v3, int v4, int v5)
{
    int a[] = { v1, v2, v3, v4, v5};
    for (int i = 0; i < 4; ++i)
    {
        int b = 0;
        for (int j = 0; j < 5; ++j)
            b += (a[j] < a[i] || (a[i] == a[j] && i < j));
        if (b == 2)
            return a[i];
    }
    return a[4];
}
```

```c
__device__ int median5(int a0, int a1, int a2, int a3, int a4)
{
    int b0 = min(a0, a1); int b1 = max(a0, a1); int b2 = min(a2, a3);
    int b3 = max(a2, a3); int c0 = min(b0, b2); int c2 = max(b0, b2);
    int c1 = min(b1, b3); int d1 = min(c1, c2); int d2 = max(c1, c2);
    int e4 = max(c0, a4); int f2 = min(d2, e4); int g2 = max(d1, f2);
    return g2;
}
```
Be careful with arrays

- Register file cannot be accessed indirectly
  - I.e., assembly code may refer to R2 or R3, but not R[R2] or such
  - Allowing this would lead to scheduling nightmare

- Statically indexed local arrays are converted to registers
  - Fixed loops are unrolled, fixed indices then converted to registers
  - You shouldn’t need to manually unroll anything

- Dynamically indexed local arrays will be stored in local memory
  - Which, unlike the name suggests, is not on chip but in DRAM
  - So in this context, local means thread-local, not physically local
// Forces vec into local memory.
__device__ float select3(float vec[3], int idx)
{
    return vec[idx];
}

// Can keep vec in registers.
__device__ float select3(float vec[3], int idx)
{
    if (idx == 0) return vec[0];
    if (idx == 1) return vec[1];
    return vec[2];
}
#define VEC_SIZE 3

// Forces vec into local memory.
__device__ float select(float vec[VEC_SIZE], int idx)
{
    return vec[idx];
}

// Can keep vec in registers.
__device__ float select(float vec[VEC_SIZE], int idx)
{
    // This loop will be unrolled by the compiler.
    for (int i = 0; i < VEC_SIZE - 1; ++i)
        if (idx == i)
            return vec[i];

    return vec[VEC_SIZE - 1];
}
Questions?
Warp-wide constructs
Warp-wide programming

- Warp structure is exposed in CUDA through intrinsics
- Warp-wide programming is often useful for efficiently implementing complex algorithms on GPU
- Treat warp as a thread, lanes as parts of SIMD register
  - Can also treat warp as 2 independent 16-lane operations, etc.
- Still take advantage of hardware SIMT support
  - For example, automatic masking of threads, handling of loops
Warp-wide operations: Vote and ballot

```
int __all(int predicate)
// Returns 1 iff predicate is non-zero for all active threads in warp

int __any(int predicate)
// Returns 1 iff predicate is non-zero for any active threads in warp

unsigned int __ballot(int predicate)
// Returns a 32-bit mask where bit k is set iff predicate is non-zero on lane k
// Inactive threads contribute a zero
// Get the mask of active threads: mask = __ballot(1);
```
Warp-wide operations: Shuffle

```
int __shfl(int var, int srcLane)
float __shfl(float var, int srcLane)
```

- Returns the value of `var` from another lane

- Also variants `__shfl_up`, `__shfl_down`, and `__shfl_xor`
  - Calculate source lane index based on target lane index and parameter

- Fast sharing of data within warp
  - Single-cycle operation
  - Note: If source lane is not active, result is **undefined**!
Example 1: Sum within a warp

// Note: This would be cleaner to implement with a loop.
// Unrolled manually for clarity.

__device__ int warp_sum(int val)
{
    val += __shfl_xor(val, 1);
    val += __shfl_xor(val, 2);
    val += __shfl_xor(val, 4);
    val += __shfl_xor(val, 8);
    val += __shfl_xor(val, 16);

    return val;
}
Example 2: Prefix sum across warp (inclusive)

// Note: This would be cleaner to implement with a loop.
// Unrolled manually for clarity.

__device__ int warp_scan(int val)
{
    int t;
    t = __shfl_up(val, 1); if (threadIdx.x >= 1) val += t;
    t = __shfl_up(val, 2); if (threadIdx.x >= 2) val += t;
    t = __shfl_up(val, 4); if (threadIdx.x >= 4) val += t;
    t = __shfl_up(val, 8); if (threadIdx.x >= 8) val += t;
    t = __shfl_up(val, 16); if (threadIdx.x >= 16) val += t;
    return val;
}
Example 3: Sum over $32 \times 32$ thread block
(Illustrated with a hypothetical $8 \times 8$ block)
Example 3: Sum over $32 \times 32$ thread block

$$
\begin{array}{cccccccc}
\Sigma_0 & \Sigma_0 & \Sigma_0 & \Sigma_0 & \Sigma_0 & \Sigma_0 & \Sigma_0 & \\
\Sigma_1 & \Sigma_1 & \Sigma_1 & \Sigma_1 & \Sigma_1 & \Sigma_1 & \Sigma_1 & \\
\Sigma_2 & \Sigma_2 & \Sigma_2 & \Sigma_2 & \Sigma_2 & \Sigma_2 & \Sigma_2 & \\
\Sigma_3 & \Sigma_3 & \Sigma_3 & \Sigma_3 & \Sigma_3 & \Sigma_3 & \Sigma_3 & \\
\Sigma_4 & \Sigma_4 & \Sigma_4 & \Sigma_4 & \Sigma_4 & \Sigma_4 & \Sigma_4 & \\
\Sigma_5 & \Sigma_5 & \Sigma_5 & \Sigma_5 & \Sigma_5 & \Sigma_5 & \Sigma_5 & \\
\Sigma_6 & \Sigma_6 & \Sigma_6 & \Sigma_6 & \Sigma_6 & \Sigma_6 & \Sigma_6 & \\
\Sigma_7 & \Sigma_7 & \Sigma_7 & \Sigma_7 & \Sigma_7 & \Sigma_7 & \Sigma_7 & \\
\end{array}
$$
Example 3: Sum over $32 \times 32$ thread block
Example 3: Sum over $32 \times 32$ thread block
Example 3: Sum over $32 \times 32$ thread block
Example 3: Sum over $32 \times 32$ thread block
Example 3: Sum over $32 \times 32$ thread block
Example 3: Sum over $32 \times 32$ thread block
**Example 3: Sum over $32 \times 32$ thread block**

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma_1$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_2$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_3$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_4$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_5$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_6$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma_7$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
</tbody>
</table>
__device__ int block_sum_32_32(int val) {

  __shared__ int sum_temp[32]; // Shared memory space for warp sums.
  val = warp_sum(val); // Compute sum within warp.
  if (threadIdx.x == 0) // Pick one thread in each warp.
    sum_temp[threadIdx.y] = val; // Store warp sum into shared memory.

  __syncthreads(); // Wait until all warp sums are in shared memory.
  if (threadIdx.y == 0) // Pick one warp.
  {
    val = sum_temp[threadIdx.x]; // Read all warp sums into individual threads.
    val = warp_sum(val); // Calculate total sum over them.
    if (threadIdx.x == 0) // Pick one thread.
      sum_temp[0] = val; // Store total sum into shared memory.
  }

  __syncthreads(); // Wait until total sum is computed and stored.
  return sum_temp[0]; // Read total sum and return it.
}
Remarks on warp-wide programming

- It is faster to operate in one thread than between threads
  - Warp-wide sum took 10 instructions per thread \( \rightarrow 32 \times 10 = 320 \) total
  - Doing this in one thread takes 31 instructions, i.e., over \( 10 \times \) as efficient
  - Block-wide sum was even less efficient

- But sometimes you cannot do much per thread
  - Too many registers per thread \( \rightarrow \) bad occupancy \( \rightarrow \) bad performance
  - Too few threads to efficiently utilize GPU with typical workload
  - Too much divergence in warp if all threads work individually

- In these cases warp-wide programming techniques help
Questions?
Memory system
Memory system
Memory system

Per-SM L1 cache
Memory system

Per-chip L2 cache
Memory system

- On-board DRAM

Diagram showing the memory system with various components such as GigaThread Engine, PCI Express 3.0 Host Interface, L2 Cache, and NVLink connections.
Accessing global memory

- Memory bus to DRAM is very wide
  - Partially due to historically coherent workloads

- Cache line size in NVIDIA GPUs is **1024** bits
  - I.e., $32 \times 32$ bits or $16 \times 64$ bits
  - Memory ops that are cached in L1 and L2 are served at this granularity

- In early architectures, L1 is bypassed for global memory access
  - Including the K2000 at Maari-A
  - Global memory is served at **256-byte** granularity (sector size in L2)
    - Still consumes full cachelines in L2
Cost of a global memory operation

- Cost ≈ number of cachelines that warp-wide memory op touches
  - Order of elements vs thread lanes does not matter

Exception: Ops wider than 32 bits per thread are first split into half-warps or quarter-warps
  - This can affect double, long long int, or vector loads / stores

Note that alignment is important
  - 32 threads accessing consecutive 32-bit items will touch 1 or 2 cachelines
    - 1 cacheline if the chunk is cacheline-aligned, 2 otherwise
  - All CUDA memory allocation functions return properly aligned memory
    - Start address aligned to at least 2048 bit boundary
Bypassing L1

That was for ops that cache in both L1 and L2

- If L1 is bypassed, the rules change a little

L2 can serve 4 independent 256-bit sectors in one go

- If L1 is active, it always requests 4 consecutive sectors → everything works effectively at cacheline granularity

For future proofing, best to stick to the 1024-bit rule
Cost of a shared memory operation

- **Shared memory is not cached**
  - It is already inside the SM, so no need

- **Shared memory is split into 32 banks**
  - Each bank can supply one 32-bit word at a time
  - Multiple accesses to same bank, different words → multiple cycles
    - This is called a **bank conflict**
  - Multiple accesses to same word (in same bank) → one cycle
    - I.e., broadcast does not cause a bank conflict
  - Access to all 32 banks from all 32 threads → one cycle
    - Any permutation is equally fast
Vector types (1/2)

- SM supports loads and stores of 64 and 128 bit types

- Built-in vector types in CUDA
  - float2, int2, uint2, float4, int4, uint4, double2
  - Structs with fields x, y, or x, y, z, w
  - Construction helpers: `make_float2(float x, float y)` etc.
  - These are aligned to full width

- Memory operations using these types are slightly more efficient than using 32-bit base types
  - Simply type cast the pointers appropriately
Vector types (2/2)

- There are also 3-component vectors:
  - float3, int3, uint3
  - These have 32-bit alignment
  - Loads and stores become three 32-bit ops

- Smaller data types and vectors:
  - char[1..4], uchar[1..4], short[1..4], ushort[1..4]
  - Internally, e.g., uchar4 load becomes a single 32-bit op
  - Be careful if using these – registers and ALUs are internally 32 / 64 bits

- Full list of type alignments and rules in programming guide
Atomic operations (1/3)

- Memory system supports a set of fast **atomic operations**
  - Hazard-free read-modify-write operation
  - All support 32-bit, some support 64-bit operands

- All return old value at memory location

- Can target global and shared memory

- Often crucial for high-performance implementations of algorithms involving data reduction
Atomic operations (2/3)

- `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`
  - Supported types vary, e.g., `float` is supported only in `atomicAdd()`.
  - `int` and `unsigned int` are supported for all.

- `atomicExch()`, `atomicCAS()`
  - Useful for atomic data structures.
  - `atomicCAS()` (compare-and-set) can emulate any atomic operation.
    - Has danger of starvation.

- `atomicInc()`, `atomicDec()`
  - Increment and decrement that wrap around into range \([0, N - 1]\).
Atomic operations (3/3)

- **atomicAnd()**, **atomicOr()**, **atomicXor()**
  - Bitwise operations
  - Both 32-bit and 64-bit operations supported

Coalescing across warp is often beneficial
- Memory address has to be same

**Example:** doing **atomicAdd()** from every thread in warp
- Compute exclusive prefix sum across warp using **__shfl**
- Have one thread execute the **atomicAdd()** with warp total
- Broadcast old value from that thread with **__shfl**
- Add prefix sum to old value in each thread
Constant memory

- Specified using the `__constant__` decorator, read-only
- Used under the hood for literals, zero overhead in assembly
  - `val = 2.5`
- Supports indexed reads, but these are **fully serialized**
  - `val = arr[idx]` takes 1 cycle if `idx` is consistent over entire warp
  - But **up to 32 cycles** if `idx` varies across the warp
  - Hence **very slow for non-coherent access**
- For incoherent access, put look-up tables in **shared memory**
Other memory types

Textures
- Read-only support for 1D, 2D and 3D arrays
- Support hardware filtering, format conversions
  - Filtering useful for, e.g., advection in grid-based simulations
- Extreme pipelining capability

Surfaces
- Read / write access with pixel format conversions
- Useful for integrating with graphics pipeline
Questions?
Asynchronous execution and events
Asynchronous operation

- Realized through CUDA streams
  - Will not go through the stream API here

- Operations in the same stream are serially dependent

- Operations in different streams can be mixed

- Streams can synchronize with each other and with CPU
  - Synchronize with other streams using events
  - Explicit synchronization with host CPU thread
Asynchronous operation, example (1/3)

Default: Everything is fully synchronous
Asynchronous operation, example (2/3)

- Memory copies overlapped with execution
- Double buffering required
Asynchronous operation, example (3/3)

- Launch kernel on data A
- Launch kernel on data B
- Launch kernel on data C

- Copy data A to GPU
- Copy data B to GPU
- Copy data C to GPU
- Copy data D to GPU

- Copy result A to CPU
- Copy result B to CPU

- Memory copies in both directions overlapped
- Can hide very long transfers
Events

- **GPU records** an event when all prior operations in stream have been completed
  - Internal clock is stored
  - CPU threads and streams waiting for the event are released

- **Useful for benchmarking / profiling**
  - `cudaEventElapsedTime()`: Query elapsed time between two events

- **Can be used for synchronizing multiple streams with each other or with host thread on CPU**
  - `cudaStreamWaitEvent()`: Stream proceeds only after event happens
  - `cudaEventSynchronize()`: Block host thread until event happens
Questions?
Bonus slides

Examining assembly code
GPU as a computing platform
Examining assembly code, how?

Two assembly-like languages in the toolchain
- PTX = Device-independent intermediate assembly (unoptimized)
- SASS = Device-specific final assembly code (optimized)

Extract PTX: `cuobjdump -ptx foo.o`
- Typically finds one PTX representation per kernel
- Not useful for optimization, only for checking correctness

Extract SASS: `cuobjdump -sass foo.o`
- Find the correct SASS for your device
- Can also extract SASS for a specific architecture using the `-arch` option
  - E.g., for Quadro K2000, specify `-arch sm_30` in the command
Examining assembly code, why?

- Check register usage
  - Is it close to what you expected?

- Check for unintended complexity
  - Local memory being used accidentally?
  - Unintended data type conversions due to storing data in wrong format?

- See tons of stuff that doesn’t seem to come from your code?
  - Slow path on a non-hardware supported operation (integer div, modulo)
  - Complex arithmetic operation (sin, cos, ...)

GPU as a computing platform (1/2)

- GPU architecture is undeniably a bit messy
  - Largely because it is exposed in great detail

- Everything is in there for a reason
  - And unless explicitly stated otherwise, the reason is **performance**

- Programming efficient GPU algorithms is a challenge
  - The payoffs can be significant
  - Peak performance per dollar is tremendously good

- Heterogeneous computing is here to stay
  - Heavy lifting vs. latency-sensitive stuff
The kind of details in this lecture are needed only by those that program GPUs on the lowest abstraction level

- Tons of higher-level GPU libraries for various purposes
- End-user doesn’t see any of this complexity, just the value

This is the only course in Aalto that touches GPU programming

- Two lessons should be enough for everyone?

Be prepared for this complexity to stay

- Give up performance to make low-level programming easier? No.
Thank you!

Questions?