GPU Programming Using CUDA C/C++

Ahmad Abdelfattah

PhD Candidate, Computer Science Program
Computer, Electrical, and Mathematical Sciences and Engineering (CEMSE) Division

King Abdullah University of Science and Technology

GPU Tutorial Day, September 18th, 2013

Outline

Outline

Before we begin

- This session introduces basic GPU programming using CUDA
- The audience should be familiar with C/C++
- The session contains CUDA examples that we will run live
- Slides and example source codes will be available online
  - http://www.hpc.kaust.edu.sa/training/2013/GPU/materials/

Outline

What is CUDA?

"CUDA™ is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) ... "

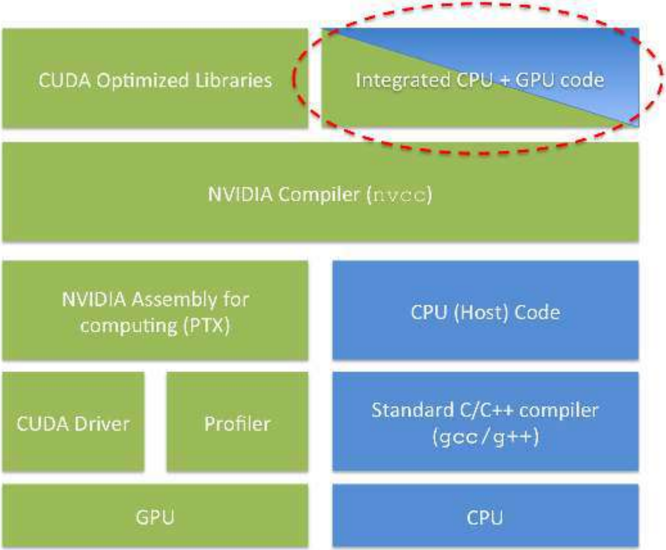
NVIDIA Website

- CUDA = Compute Unified Device Architecture
- It is a model that combines both hardware and software. GPU has to be CUDA-enabled
- Through CUDA, you can program GPUs using C, C++, Fortran, Java, Python, and more
- We will talk about CUDA C/C++
- From now on: $\boxed{\text{CPU = Host}}$ and $\boxed{\text{GPU = Device}}$

CUDA C/C++

- CUDA provides some extensions to the standard C/C++ languages
- These extensions
  - define the GPU tasks (**kernels**)
  - control CPU/GPU interaction
  - handle parallelism (many-thread execution)
  - control execution sequence on the GPU (in case of concurrent kernel execution)
- GPU code (CUDA code) is usually written in `.cu` files, and compiled with the NVIDIA compiler `nvcc`
- Almost every CUDA accelerated code is linked with the CUDA runtime library (`-lcudart`)

CUDA Software Development



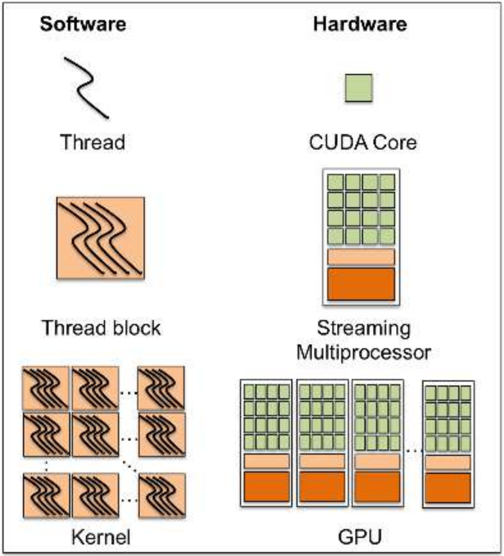
CUDA C/C++ Extensions

- The host always initiates work for the device
  - For Kepler GPUs, device can generate work for itself (Dynamic Parallelism)
- Examples for common extensions are
  - `__global__` defines a GPU kernel that is callable by the CPU
  - `__device__` defines a GPU function that is callable by a GPU kernel or by another device function, but not callable by a CPU (host)
  - `__host__` or no qualifier marks a CPU function

CUDA Kernel Organization

- A CUDA kernel is a grid of thread blocks
  - The grid can be 1D, 2D or 3D
- A thread block is, in turn, a 1D, 2D, or 3D array of threads
- A thread is executed by exactly one stream processor or CUDA core
- A thread block is executed by exactly one Streaming Multiprocessor (SM)
- However, CUDA cores and SM can interleave execution of multiple threads and thread blocks

CUDA Execution Model



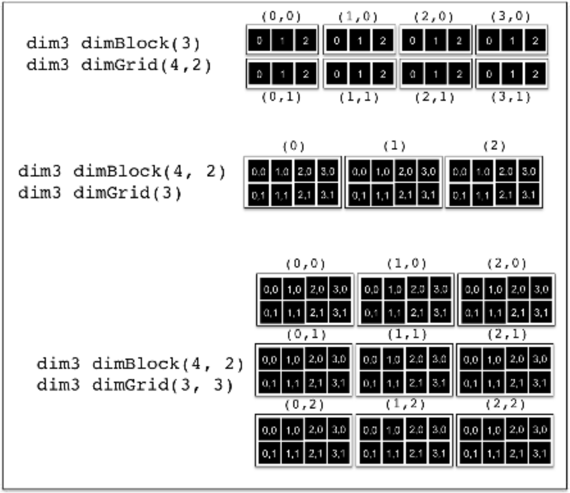
Transparent Scalability

- There is no relation between the number of thread blocks and the number of SMs
- No matter how many thread blocks are launched, the CUDA runtime will automatically schedule them to run on the available SMs
- To guarantee such scalability, CUDA does NOT allow communication/synchronization among thread blocks
- Only threads within the same thread block can communicate/synchronize

Block/Thread Indexing

- CUDA maintains special variables that store:
  - thread index within a thread block `threadIdx(.x, .y, .z)`
  - block index within a kernel grid `blockIdx(.x, .y, .z)`
  - dimension of a thread block `blockDim(.x, .y, .z)`
  - dimension of a kernel grid `gridDim(.x, .y, .z)`
- These variables are used without declaration
- Mainly used in assigning the work per block/thread
- The maximum values of the `(x, y, z)` in both `blockDim` and `gridDim` are GPU-dependent
- At least the `x` component needs to be declared. The default value for the `y` and `z` components is 1

Block/Thread Indexing Example



Basic CPU Code Skeleton

```
int main() {
    // Allocate and initialize memory on CPU

    // Allocate memory on GPU using cudaMalloc()
    // Off-load input data from CPU to GPU using cudaMemcpy()

    // Launch the GPU kernel(s)

    // Copy the results back from GPU to CPU using cudaMemcpy()

    // Free CPU resources
    // Free GPU resources using cudaFree()
}
```

Outline

Example 1: Hello World

- This example shows how to launch a very simple kernel that prints some text on the screen
- `printf()` is natively supported on CUDA
- No data copies are needed between CPU and GPU
- Each thread will print its own ID
- The example also shows how to reorganize threads

Hello World 1

```
__global__ void helloworld_1()
{
  // compute local thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int tz = threadIdx.z;

  // compute local block ID
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int bz = blockIdx.z;

  printf("Hello from thread (%d, %d, %d) in block (%d, %d, %d) \n"
  , tx, ty, tz, bx, by, bz);
}
```

Hello World 2

```
__global__ void helloworld_2()
 {
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // move from 2D to 1D
  int tid = ty * blockDim.x + tx;

  printf("Hello from thread (%d, %d) => %d \n", tx, ty, tid);
}
```

Helloworld: Live Demo

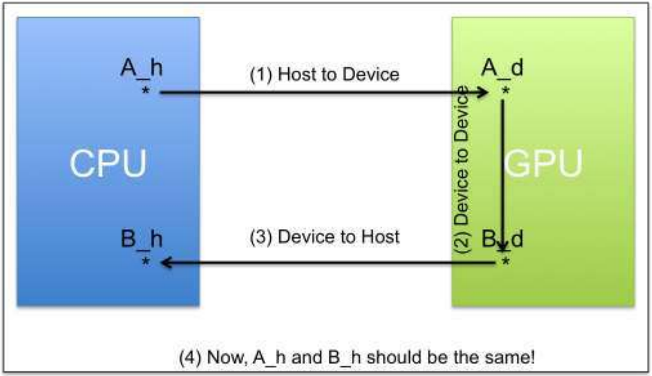
- We are going to
  - have a look at the code structure
  - see how to the compile the code (only one `.cu` file)
  - learn the syntax for kernel launch
    - `kernel-name<<<grid, block>>>(arguments)`
  - see the output for different kernel configuration
  - introduce the concept of per-warp execution
    - Threads are packed into groups of 32 (called warps)
    - Threads in the same warps share the same instruction stream
    - Hardware always rounds up to execute full warps

Example 2: Data Copies

- The example shows how to copy data between CPU and GPU, and inside the GPU memory
- The key function is `cudaMemcpy(dst, src, size, kind)`, where
  - `dst`: pointer to destination memory
  - `dst`: pointer to source memory
  - `size`: size of the data in bytes
  - `kind`: direction of the memory copy, which is
    - `cudaMemcpyHostToHost`: CPU to CPU
    - `cudaMemcpyHostToDevice`: CPU to GPU (off-load input data)
    - `cudaMemcpyDeviceToHost`: GPU to CPU (read results back)
    - `cudaMemcpyDeviceToDevice`: GPU to GPU

Example 2: Data Copies

- No compute tasks (kernels) are launched
- We are going to try the following scenario



Data Copies: Sample Code

```
{
  .
  .
  .
  // allocate gpu memory
  cudaMalloc((void**)&da, length * sizeof(float));
  cudaMalloc((void**)&db, length * sizeof(float));

  // Copying from host to device
  cudaMemcpy(da, ha, length * sizeof(float), cudaMemcpyHostToDevice);

  // Copying inside device memory
  cudaMemcpy(db, da, length * sizeof(float), cudaMemcpyDeviceToDevice);

  // Copying back from device to host
  cudaMemcpy(hb, db, length * sizeof(float), cudaMemcpyDeviceToHost);

  // Compare data pointed to by ha & hb ... should be the same
  .
  .
  .
}
```

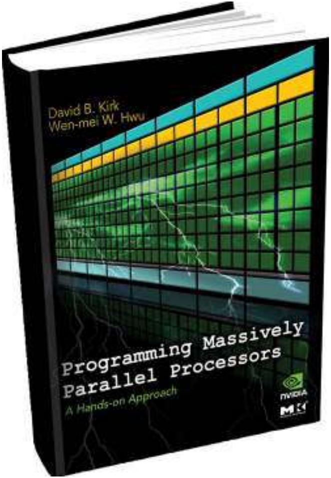
Example 3: Timing GPU Tasks

- Timing is essential for measuring performance
- Timing GPU tasks is done using CUDA events (`cudaEvent_t`)
- A CUDA event can record a certain moment in time
- Timing, therefore, requires at least 2 events

```
float time = 0.0; cudaEvent_t start, stop;
// create events
cudaEventCreate(&start);
cudaEventCreate(&stop);
...
cudaEventRecord(start, 0);
// launch GPU kernel / data copy
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
...
// destroy events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Comprehensive Example: Matrix-Matrix(MM) Multiplication

- Example can be found in this book
- Recommended for more illustration and details

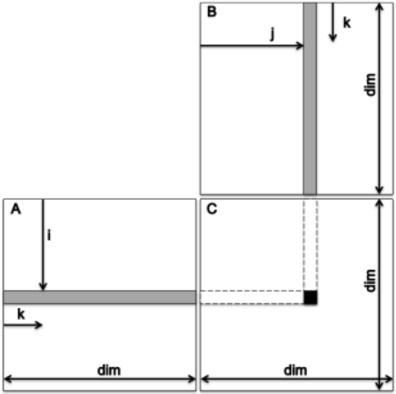


Comprehensive Example: Matrix-Matrix(MM) Multiplication

- $\mathtt{C}_{m \times n} = \mathtt{A}_{m \times p} \times \mathtt{B}_{p \times n}$
- Compute intensive, embarrassingly parallel, and regular computation pattern
    - Best suited for GPUs
- A general standard kernel exists in level-3 BLAS: GEMM
- A common benchmark to measure the sustained peak performance
- We will start with a simple implementation, and add incremental optimization
    - We will study 4 versions in total

MM Multiplication: CPU version

```
void matmul(float *A, float *B, float *C, int dim)
{
  for(int i = 0; i < dim; i++)
    for(int j = 0; j < dim; j++)
    {
      float sum = 0.0;
      for(int k = 0; k < dim; k++)
      {
        float a = A[i * dim + k];
        float b = B[k * dim + j];
        sum += a * b;
      }
      C[i * dim + j] = sum;
    }
}
```



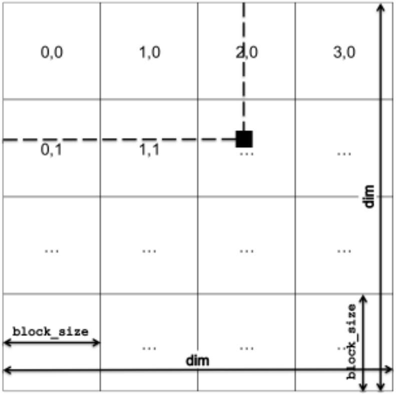
- #flops $= 2n^3$, where $n =$ `dim`
- On GPU, we will assign a thread per output element in `C`
- The two outer loops will, then, disappear in the GPU version

MM Multiplication: GPU version 1

- Key design approach: Each thread will compute exactly one element in the output matrix `C`
- How should we configure the kernel?
  - A Kepler GPU supports up to 1024 threads in one thread block
  - Obviously, a kernel with one thread block will not work if the matrix dimension > 32
  - Alternatively, we will divide the output matrix `C` into square submatrices, each with dimension = `block_size`
  - Each thread block in the GPU kernel will be responsible for a square submatrix (one thread per element)
  - `block_size` can be 32 at maximum, for a Kepler GPU

MM Multiplication: GPU version 1

- Kernel configuration: 2D grid with dimension = `dim` ÷ `block_size`
  - achieves a simple mapping from thread block to submatrix
- Each thread will have to compute its global (x, y) coordinates
  - i.e., with respect to the whole grid, not to the thread block



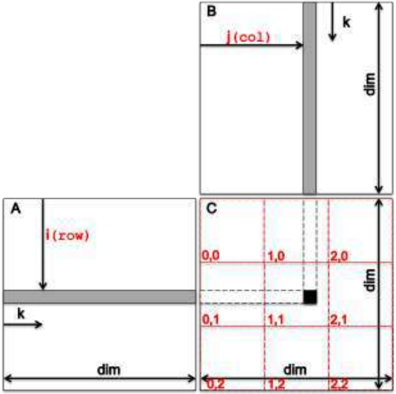
MM Multiplication: GPU version 1

```
__global void matrixmul_1(float* A, float* B, float*
C, int dim, int block_size)
{
   const int bx = blockIdx.x;
   const int by = blockIdx.y;
   const int tx = threadIdx.x;
   const int ty = threadIdx.y;

   // identify (row, col) position of output element
   const int row = by * block_size + ty;
   const int col = bx * block_size + tx;

   // compute the output element
   float sum = 0.0;
   for(int k = 0; k < dim; k++)
      sum += A[row * dim + k] * B[ k * dim + col];

   // store the result
   C[row * dim + col] = sum;
}
```



MM Mult. ver 1: Live Demo

- We will run the kernel for different block sizes (1, 2, 4, 8, 16, and 32)
- You will see that performance increases with increasing the block size
  - We increase parallelism as we increase block size
  - Only the 32×32 configuration makes use of full warps

What is wrong with ver 1?

- The kernel is compute bound, meaning that flops $O(n^3)$ dominate memory accesses $O(n^2)$
- A compute bound kernel should score a peak performance that is close to the GPU theoretical peak performance
  - $\approx$ 3 Tflop/s (single precision) for a K20c GPU
- In the following slides, we will enhance ver 1 step by step

MM Multiplication ver 2: Exploit Data Reuse

- Consider the 4×4 matrix, processed by a 4×4 thread block
- Since each thread works independently, every row in the submatrix will be read 4 times from global memory
- This causes extra unnecessary global memory traffic
- Plus, global memory accesses has the biggest penalty (in terms of latency) throughout the whole memory system
- Threads should cooperate to exploit data reuse from a fast memory rather than from global memory

a(0,0)	a(1,0)	a(2,0)	a(3,0)
a(0,1)	a(1,1)	a(2,1)	a(3,1)
a(0,2)	a(1,2)	a(2,2)	a(3,2)
a(0,3)	a(1,3)	a(2,3)	a(3,3)

MM Multiplication ver 2: Exploit Data Reuse

- A thread should to be able to access an element that has been fetched by another thread
- We need some sort of shared resources which all threads can access

- This is where shared memory (SHMEM) comes into play
- SHMEM is user-controlled cache. It is as fast as L1 cache
- Each thread will read one element of the submatrix and load it into shared memory
- A thread `x` should now be able to read an element that has been fetched by another thread `y`
  - But thread `x` needs to make sure that `y` has already loaded the element into shared memory
  - For that, we use the barrier function `__syncthreads()`

MM Multiplication ver 2: Exploit Data Reuse

```
template<int block_size>
__global__ void matrixmul_2(float* A, float* B, float* C, int dim)
{
   // compute tx, ty, bx, by as before
   __shared__ float as[block_size][block_size];
   __shared__ float bs[block_size][block_size];
   const int row = by * block_size + ty;
   const int col = bx * block_size + tx;
   float sum = 0.0;
   for(int m = 0; m < dim/block_size; m++)
   {
     as[ty][tx] = A[row * dim + (m * block_size + tx)];
     bs[ty][tx] = B[(m * block_size + ty) * dim + col];
     __syncthreads();
     for(int k = 0; k < block_size; k++)
       sum += as[ty][k] * bs[k][tx];
     __syncthreads(); // Why?
   }
   C[row * dim + col] = sum;
}
```

MM Mult. ver 2: Live Demo

- Investigate the differences between ver 2 and ver 1
- We will run the kernel for different block sizes (1, 2, 4, 8, 16, and 32)
- Compare performance against version 1

MM Mult. ver 3: Instruction Mix

- Now we pay attention to useful computation
- Performance is measured in flops/s. Only floating point operations count
- However, integer and branch instructions are necessary to handle array indexing, address calculation, for loop counter updates, ... etc
- Loop unrolling is an optimization that increases the ratio of floating point instructions to integer and branch "unuseful" instructions
- It is usually a compiler technology (# pragma unroll)
- It helps eliminate the branch instruction and the counter update of a loop
- The ideal case is when the loop is fully unrolled
  - This needs the loop interval to be known at compile time
  - Luckily, we can fully unroll the inner most loop in the MM Multiplication example

MM Mult. ver 3: Live Demo

- Investigate the differences between ver 3 and ver 2
- We will run the kernel for different block sizes (1, 2, 4, 8, 16, and 32)
- Compare performance against versions 1 and 2

MM Mult. ver 4: Data Prefetching

- Processors are much faster than memory systems. This is true for GPUs too
- For better performance, processors (cores) should not wait for data
- In order to hide memory latency, we will use data prefetching
  - Prefetch the next data block while the current is being processed

```
Load first tile from global memory into registers
Loop
{
  Deposit tile from registers to shared memory
  __syncthreads()
  Load next tile from global memory into registers
  Compute current tile
  __syncthreads()
}
Compute the last tile
```

MM Mult. ver 4: Live Demo

- Investigate the changes made to ver 3 to apply data prefetching
- Run the kernel for different block sizes (1, 2, 4, 8, 16, and 32)
- Compare performance against versions 1, 2, and 3

Cublas GEMM vs MKL GEMM: Live Demo

- Standard implementations provided by NVIDIA and Intel
- Both comfortably outperform the 4 versions explained earlier
  - We tried only basic optimizations

SGEMM Sample Results

Outline

CUDA Streams

"A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g., inter-kernel communication is undefined)..."

CUDA C Programming Guide

- If there is a dependency between two kernels (or more) in different streams, the developer has to manually handle it
  - Using synchronization
  - Using CUDA events
- Any kernel or data copy operation is submitted to a stream
  - If no stream is defined, the default is stream 0
- In multi-GPU programming, communication and computation are usually submitted into different stream to hide communication penalty

Submitting Commands to Streams

```
cudaStream_t strema_1, stream_2;

cudaStreamCreate(&stream_1);
cudaStreamCreate(&stream_2);
    ⋮
kernel_1<<<grid, block, 0, stream_1>>>(...);
kernel_2<<<grid, block, 0, stream_2>>>(...);
kernel_3<<<grid, block, 0, stream_1>>>(...);
// kernel_3 depends on kernel_1. Both are independent from kernel_2
    ⋮
cudaStreamDestroy(stream_1);
cudaStreamDestroy(stream_2);
```

- Communication can overlap computation through
  - `cudaMemcpyAsync()` for CPU-GPU interactions
  - `cudaMemcpyPeerAsync()` for GPU-GPU interactions (must be on the same node)

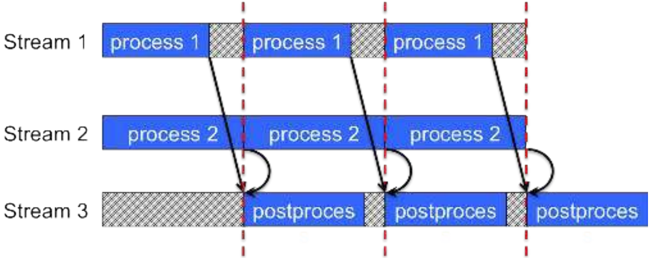
Inter-stream Dependencies

- Approach 1: Using synchronization
  - cudaDeviceSynchronize(): Sync against everything on GPU
  - cudaStreamSynchronize(): Sync against a specific stream
  - cudaEventSynchronize(): Sync against a specific event that has been recorded in a certain stream
  - All these calls block the CPU thread
- Approach 2: Using asynchronous calls
  - Using cudaStreamWaitEvent()

```
kernel_1<<<..., ..., ..., stream_x>>>(...);
cudaEventRecord(event_1, stream_x);
⋮
cudaStreamWaitEvent(stream_y, event_1, 0);
kernel_2<<<..., ..., ..., stream_y>>>(...);
// kernel 2 in stream y depends on kernel 1 in stream x
// CPU is free to do concurrent computation
⋮
```

Toy Example: Concurrency

- Three kernels: `process_1()`, `process_2()`, and `postprocess()`
- `postprocess()` depends on `process_1()` and `process_2()`
- `process_1()` and `process_1()` are mutually independent
- We have multiple inputs and want to pipeline the execution of all kernels in a batch execution



Toy Example: Two Scenarios

- Scenario 1: offload, compute, and then copy back

- Scenario 2: offload with concurrent compute and copy back
  - Hides the communication time from GPU to CPU

Toy Example: Scenario 1

```
process_1<<<..., ..., ..., stream_1>>>(...);
process_2<<<..., ..., ..., stream_2>>>(...);
Loop
{
    // wait for both stream then post-process;
    cudaStreamSynchronize(stream_1);
    cudaStreamSynchronize(stream_2);
    postprocess<<<..., ..., ..., stream_3>>>(...);
    .
    .
    .
    // process next input in the batch;
    process_1<<<..., ..., ..., stream_1>>>(...);
    process_2<<<..., ..., ..., stream_2>>>(...);
}
// finalize last iteration;
cudaStreamSynchronize(stream_1);
cudaStreamSynchronize(stream_2);
postprocess<<<..., ..., ..., stream_3>>>(...);

// Wait for GPU to finish then copy back;
cudaDeviceSynchronize();
cudaMemcpy(..., ..., ..., cudaMemcpyDeviceToHost);
```

Toy Example: Scenario 2

```
process_1<<<..., ..., ..., stream_1>>>(...);
process_2<<<..., ..., ..., stream_2>>>(...);
Loop {
    // wait for both stream then post-process;
    cudaStreamSynchronize(stream_1);
    cudaStreamSynchronize(stream_2);
    postprocess<<<..., ..., ..., stream_3>>>(...);
    // schedule a D2H copy right after post_process;
    cudaEventRecord(processing_done, stream_3);
    cudaStreamWaitEvent(stream_4, processing_done, 0);
    cudaMemcpyAsync(..., ..., ..., cudaMemcpyDeviceToHost, stream_4);

    // process next input in the batch;
    process_1<<<..., ..., ..., stream_1>>>(...);
    process_2<<<..., ..., ..., stream_2>>>(...);
}
cudaStreamSynchronize(stream_1);
cudaStreamSynchronize(stream_2);
postprocess<<<..., ..., ..., stream_3>>>(...);
// schedule a D2H copy right after post_process;
cudaEventRecord(processing_done, stream_3);
cudaStreamWaitEvent(stream_4, processing_done, 0);
cudaMemcpyAsync(..., ..., ..., cudaMemcpyDeviceToHost, stream_4);
```

Profiling the Toy Example: Live Demo

- Profile the toy example using NVIDIA profiler (`nvprof`, `nvvp`)
- Investigate the time lines produced by `nvvp`
- Real timelines generated by `nvvp`
  - Scenario 1

  - Scenario 2

Outline

1. Session Overview
2. Introduction
3. Examples
4. Streams and Concurrency
5. Summary

Session Summary

- You should now be familiar with
  - How GPUs are programmed for general purposes
  - How a CPU can drive a GPU as an accelerator
  - The CUDA execution model
  - Basic optimizations to CUDA kernels
  - Measuring performance of a CUDA kernel
  - Concurrent kernel-execution/data-communication on a single GPU

Thank You