

Programming GPUs with CUDA

Day 2

Slides: <http://syam.sharcnet.ca/CUDA2.pdf>

Sergey Mashchenko
SHARCNET

Summer School on High Performance Computing
University of Toronto, July 13-17, 2015

Outline

- Preliminary remarks
- When to use CUDA?
- C language extensions
- CUDA code optimization
- New NVIDIA GPUs

Preliminary remarks

Preliminary remarks

- The lecture only covers capability 2.x GPUs (like Tesla M2070 in cluster monk).
- If time permits, some new features of the newer Tesla GPUs K20 / K40 / K80 will be covered at the end.

When to use CUDA?

When to use CUDA?

- Writing a CUDA code from scratch
 - Is it really necessary to write an explicitly parallel code?
 - Going parallel means more researcher's time spent on code development, debugging, profiling, maintenance, and less – on research
 - Less HPC resources are available for parallel codes (almost not an issues for MPI codes, still a big obstacle for GPU codes)
 - If your project can be carried out using a *serial farm* (a bunch of independent serial processes), you can have an almost 100% scalability (speedup will be simply proportional to the number of CPU cores), whereas parallel codes have <100% scalability, and scalability drops for larger N_{cpu} , placing a limit of how many CPU cores one can efficiently use.
 - Some algorithms are inherently non-parallizable (because of severe data dependencies). Example: classical formula for Fibonacci series:

$$F_n = F_{n-2} + F_{n-1}; \quad F_0=0, \quad F_1=1$$

When to use CUDA?

- Writing a CUDA code from scratch
 - When to use CUDA, as opposed to MPI, OpenMP, pthreads, ... ?
 - When the bulk of computations can be done in a data parallel fashion, with the number of parallel independent threads $> \sim 1000$ (the more, the better)
 - When the code will be very fine-grained (very short alternating computation and communication episodes)
 - When you can program in C/C++ (Fortran CUDA is not provided by NVIDIA – **but now provided by PGI**)
 - When the expected speedup more than compensates for the scarcity and/or higher cost of GPU resources (should probably be > 20 to make a sense)

When to use CUDA?

- Converting an existing code to CUDA
 - Pretty much the same requirements as for writing CUDA code from scratch
 - Serial and OpenMP codes should be easier to convert to CUDA than MPI and pthreads codes
 - C++/C codes are the easiest; Fortran codes would have to be re-written in C++. (Unless you want to take a chance with non-standard Fortran CUDA extensions, like the PGI implementation)

Question

When to use CUDA?

- Should you commit your code to CUDA?
 - CUDA is still very new, is evolving fast, and it is not an open standard (like OpenCL).
 - The underlying hardware (GPUs) is also quickly evolving.
 - As a result, CUDA codes can become obsolete very quickly, unless they are very actively maintained.
 - More importantly, it is not clear if GPUs will remain mainstream HPC a few years down the road – new HPC technologies (like Intel Phi) might completely replace GPU computing at some point.

When to use CUDA?

- Should you commit your code to CUDA?
 - It is advisable to maintain both CUDA and non-CUDA version of the code:
 - One could use macros to have e.g. both serial and CUDA versions in one code (you compile with different switches to get the version you need)
 - One could rely on CUDA emulators, and maintain a pure CUDA code, but (a) NVIDIA no longer provides an emulator, (b) third party emulators (like PGI's CUDA-x86) can have efficiency and portability issues, and most importantly (c) when/if CUDA/GPU HPC will disappear, CUDA/GPU emulators will follow.
 - Or one can simply co-develop two separate versions of the code, CUDA and non-CUDA

When to use CUDA?

- Positive aspects of going CUDA
 - Get your science done much sooner, or go after much larger problems
 - By doing non-CUDA -> CUDA code conversion, you are forced to re-arrange the code in the way which makes it much easier to adapt it to any future data-parallel friendly HPC technology (like Phi's, CPUs with dozens of cores etc.).

When to use CUDA?

- What speedup with CUDA to aim at?
 - In terms of availability of resources:
 - SHARCNET's GPU cluster, monk, has 54 nodes with two GPUs (M2070) in each – so 108 GPUs in total.
 - SHARCNET's resources for MPI/serial farming codes are ~20,000 CPU cores
 - So formally, you'd want to aim at >200x speedups, which is unrealistic for most codes.
 - GPU availability will likely to improve*, so probably a better comparison is in terms of costs:
 - One orca node costs ~5000\$. One monk node, with 2 GPUs, costs ~10,000\$. Then your CUDA code should aim at running faster on one monk's GPU than on one orca node (24 CPU cores), meaning a speedup $\geq 24x$.

C language extensions

C language extensions

- Kernels (one block)

```
// Kernel definition
__global__ void VecAdd (float* d_A, float* d_B, float* d_C)
{
    int i = threadIdx.x;
    d_C[i] = d_A[i] + d_B[i];
}

int main()
{
    ...

// Kernel invocation with N threads
    VecAdd <<<1, N>>> (d_A, d_B, d_C);

    ...
}
```

Pointers to
device
addresses!



C language extensions

- Kernels (multi-block)

```
// Kernel definition
__global__ void VecAdd (float* d_A, float* d_B, float* d_C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    d_C[i] = d_A[i] + d_B[i];
}

int main()
{
    ...

// M blocks with N threads each:
    VecAdd <<<M, N>>> (d_A, d_B, d_C);
    ...
}
```


C language extensions

- Static global device arrays

```
// Device code:
```

```
__device__ float d_A[10][50];
```

```
// Host code:
```

```
float h_A[10][50];
```

```
// Host to device copying:
```

```
cudaMemcpyToSymbol (d_A, &h_A, sizeof(h_A), 0, cudaMemcpyHostToDevice);
```

```
// Device to host copying:
```

```
cudaMemcpyFromSymbol (&h_A, d_A, sizeof(h_A), 0, cudaMemcpyDeviceToHost);
```

C language extensions

- Dynamic global device arrays

```
// Host code:
```

```
// Host array allocation:
```

```
size_t size = N * sizeof(float);  
float* h_A = (float*) malloc (size);
```

```
// Device array allocation:
```

```
float* d_A;  
cudaMalloc(&d_A, size);
```

```
// Host to device copying:
```

```
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
```

```
// Device to host copying:
```

```
cudaMemcpy (h_A, d_A, size, cudaMemcpyDeviceToHost);
```

C language extensions

- Device functions

```
// Device code  
  
// Function:  
__device__ double my_function (double x)  
{  
    double f;  
    ...  
    return f;  
}  
  
// Kernel:  
__global__ void my_kernel ();  
{  
    double f1, x1;  
    f1 = my_function (x1);  
}
```

C language extensions

- Shared memory
 - Can be much faster than global (device) memory
 - Shared across the threads in a single block
 - The amount is very limited, so it is often a limiting factor for CUDA optimization
 - Typically statically defined

```
// Device code
```

```
__shared__ double A_loc[BLOCK_SIZE];
```

C language extensions

- Execution synchronization on device
 - Can only be done within a single block.
 - As a result, only up to 1024 (usually between 64 and 256) threads can be synchronized.
 - Used for binary reductions; more generally: when a data dependence exists between different parts of a kernel

```
// Device code  
  
__syncthreads();
```

C language extensions

- Execution synchronization on device
 - Handling a data dependence between different parts of a kernel

```
// Kernel code
__shared__ float A[BLOCK_SIZE];

// Initializing all threads in a block:
A[threadIdx.x] = 1;

// This is needed as the previous section is executed by sequential groups of
// 32 (warp size) threads, in an undetermined order
__syncthreads();

// A primitive example of a data dependence:
if (threadIdx.x == 0)
    for (i=0; i<BLOCK_SIZE; i++)
        sum = sum + A[i];
```

C language extensions

- Synchronization between host and device
 - Kernel calls and some CUDA memory operations are executed asynchronously on host
 - But in many cases you need host-device synchronization, e.g.:
 - Before using on the host a result computed in a kernel
 - Before using a host timer, for profiling

```
// Host code
```

```
CudaDeviceSynchronize ();
```

C language extensions

- Synchronization between host and device
 - Example: when copying data from device to host

```
// Device code  
__device__ int d_y;  
  
// Host code  
int h_y;  
  
// Kernel computes the result and stores it in global device variable d_y;  
my_kernel <<<M, N>>> ();  
// Copying the result to host:  
cudaMemcpyFromSymbol (&h_y, d_y, sizeof(h_y), 0, cudaMemcpyDeviceToHost);  
// Forcing host to pause until the copying is done:  
CudaDeviceSynchronize ();  
// Using the result on host:  
printf ("Result: %d\n", h_y);
```


C language extensions

- Synchronization between host and device
 - Example: when profiling the code

```
// Host code
```

```
struct timeval tdr0, tdr1;
```

```
gettimeofday (&tdr0, NULL);
```

```
my_kernel <<<M, N>>> ();
```

```
// Without synchronization, tdr1-tdr0 will not measure time spent inside the kernel
```

```
// (it will be much smaller):
```

```
CudaDeviceSynchronize ();
```

```
gettimeofday (&tdr1, NULL);
```

Question

C language extensions

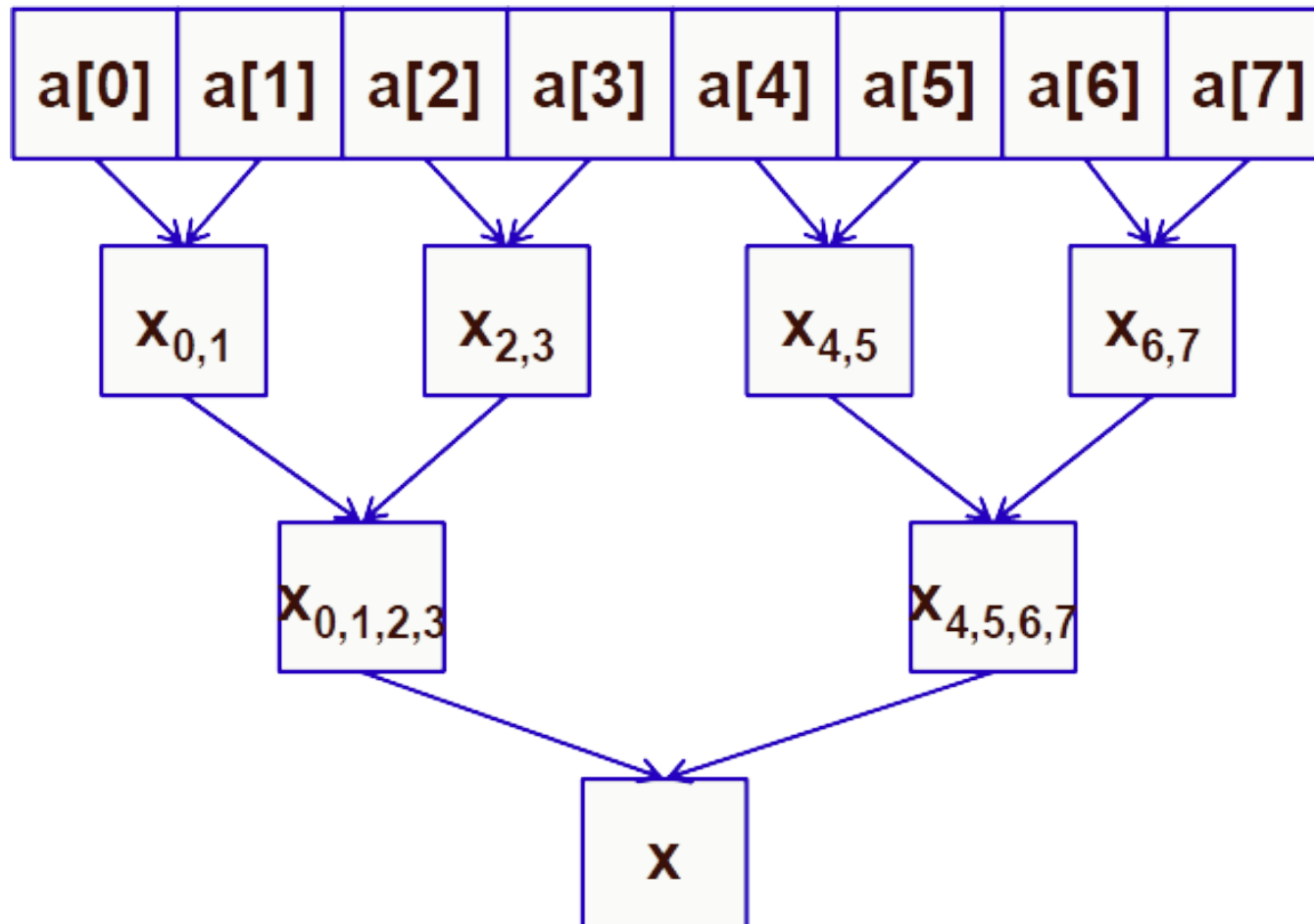
- Reductions in CUDA
 - Reductions: min/max, average, sum, ...
 - Can be a significant bottleneck for the performance, because it breaks pure data parallelism.
 - There is no perfect way to do reductions in CUDA. The two commonly used approaches (each with its own set of constraints) are
 - Binary reductions
 - Atomic reductions

C language extensions

- Binary reductions
 - The most universal type of reductions (e.g., the only way to do double precision reductions)
 - Even when using single precision (which is faster than double precision), binary summation will be more accurate than atomic summation, because it employs more accurate *pairwise summation*.
 - Usually the more efficient way to do reductions

C language extensions

Binary reduction



C language extensions

- Binary reductions
 - But: typically relies on (very limited) shared memory – placing constraints on how many reductions per kernel one can do
 - Relies on thread synchronization, which can only be done within a single block – places constraints on how many threads can participate in a binary reduction (usually 64 ... 256; maximum 1024)
 - For a large number of data elements (>1024), this leads to the need to do multi-level (multi-kernel) binary reductions, with storing the intermediate data in device memory; this can reduce the performance
 - Can be less efficient for small number of data elements (<64)
 - Significantly complicates the code

C language extensions

- Examples: binary summation with the number of elements being a power of two. The result is in sum[0].

```
__shared__ double sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x; // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1)
{
    int halfPoint = (nTotalThreads >> 1); // divide by two

    if (threadIdx.x < halfPoint)
    {
        int thread2 = threadIdx.x + halfPoint; // the second element index
        sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two }
```

C language extensions

- Examples: binary min/max with the number of elements being a power of two.

```
__shared__ double min[BLOCK_SIZE];
...
__syncthreads(); // To make sure all min[] elements were initialized
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
    int halfPoint = (nTotalThreads >> 1); // divide by two
    if (threadIdx.x < halfPoint) {
        int thread2 = threadIdx.x + halfPoint; // the second element index
        double temp = min[thread2];
        if (temp < min[threadIdx.x])
            min[threadIdx.x] = temp;
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```


C language extensions

- Examples: multiple binary reductions.

```
__shared__ double min[BLOCK_SIZE], sum[BLOCK_SIZE];  
...  
__syncthreads(); // To make sure all array elements were initialized  
int nTotalThreads = blockDim.x;  
  
while(nTotalThreads > 1)  
{  
    int halfPoint = (nTotalThreads >> 1); // divide by two  
    if (threadIdx.x < halfPoint) {  
        int thread2 = threadIdx.x + halfPoint;  
        sum[threadIdx.x] += sum[thread2]; // First reduction  
  
        double temp = min[thread2];  
        if (temp < min[threadIdx.x])  
            min[threadIdx.x] = temp; // Second reduction  
    }  
    __syncthreads();  
    nTotalThreads = halfPoint; // Reducing the binary tree size by two }
```

C language extensions

- Examples: two-level binary reduction

```
// Host code  
#define BSIZE 1024 // Always use a power of two; can be 32...1024  
// Total number of elements to process: 1024 < Ntotal < 10242  
  
int Nblocks = (Ntotal+BSIZE-1) / BSIZE;  
  
// Low level (the results should be stored in global device memory):  
x_prerreduce <<<Nblocks, BSIZE >>> ();  
  
// High level (will read the input from global device memory):  
x_reduce <<<1, Nblocks >>> ();
```

C language extensions

- Examples: binary reduction with an arbitrary number of elements (BLOCK_SIZE).

```
__shared__ double sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim_2; // Total number of threads, rounded up to the next power of two

while(nTotalThreads > 1)
{
    int halfPoint = (nTotalThreads >> 1); // divide by two

    if (threadIdx.x < halfPoint)
    {
        int thread2 = threadIdx.x + halfPoint;
        if (thread2 < blockDim.x) // Skipping the fictitious threads blockDim.x ... blockDim_2-1
            sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

C language extensions

- Continued: binary reduction with an arbitrary number of elements.
 - You will have to compute `blockDim_2` (`blockDim.x` rounded up to the next power of two), either on device or on host (and then copy it to device). One could use the following function to compute `blockDim_2`, valid for 32-bit integers:

```
int NearestPowerOf2 (int n)
{
    if (!n) return n; // (0 == 2^0)

    int x = 1;
    while(x < n)
    {
        x <<= 1;
    }
    return x;
}
```

Question

C language extensions

- Atomic reductions
 - Very simple and elegant code
 - Almost no change compared to the serial code
 - A single line code: much better for code development and maintenance
 - No need for multiple intermediate kernels (saves on overheads related to multiple kernel launches)
 - Requires no code changes when dealing with any number of data elements – from 2 to millions
 - Usually more efficient when the number of data elements is small (<64)

C language extensions

- Atomic reductions
 - But: atomic operations are serialized, which usually means worse performance
 - Only single precision accuracy - can become really bad for summation and averaging when the number of elements is large (many thousands) – because it uses sequential summation.
 - A commonly employed good compromise is to use binary reduction at the lower level, and then use atomic reduction at the higher level. (*Only for single precision*).
 - All the above means that to find the right way to carry out a reduction in CUDA, with the right balance between code readability, efficiency, and accuracy, one often has to try different approaches, and choose the most efficient.

C language extensions

- Examples: atomic reductions.

```
// In global device memory:  
__device__ float xsum;  
__device__ int isum, imax;  
  
// In a kernel:  
float x;  
int i;  
__shared__ imin;  
...  
atomicAdd (&xsum, x);  
atomicAdd (&isum, i);  
atomicMax (&imax, i);  
atomicMin (&imin, i);
```

- Some other atomic operations:
 - atomicExch, atomicAnd, atomicOr

C language extensions

- Binary at the lower level, atomic at the higher level

```
__shared__ float sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x; // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1){
    int halfPoint = (nTotalThreads >> 1); // divide by two

    if (threadIdx.x < halfPoint)
    {
        int thread2 = threadIdx.x + halfPoint; // the second element index
        sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
if (threadIdx.x == 0)
    atomicAdd (&xsum, sum[0]); // Atomic reduction
```

C language extensions

- How about the “only the first half of the threads will be active” part? Does it mean that the algorithm is 2x slower than it could be?
 - One can re-write the original binary reduction code in a way where each thread reads two values of the array from device memory (instead of one), but block size is 2x smaller, and each thread is active in the first iteration.
 - Naively, the new approach shouldn't be faster, as we have the same number of active threads processing the same number of array elements in each block.
 - For fairly complicated reasons (efficiency of thread scheduling etc.), the new approach is comparable to the old one for small N (~10,000), but becomes up to 3x faster for very large N.

C language extensions

- Better algorithm (all threads are active)

```

int i = threadIdx.x + blockDim.x * blockIdx.x;
sum[threadIdx.x] = d_A[i]; // Initializing first half of sum[]
sum[threadIdx.x+BLOCK_SIZE_T] = d_A[i+SHIFT]; // Initializing the second half of sum[]
__syncthreads(); // To make sure all sum[] elements were initialized
int nElements = BLOCK_SIZE_E; // Total number of elements to process in this block;

while(nElements > 1){
  int halfPoint = (nElements >> 1); // divide by two

  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint; // the second element index
    sum[threadIdx.x] += sum[thread2]; // Pairwise summation
  }
  __syncthreads();
  nElements = halfPoint; // Reducing the binary tree size by two
}
if (threadIdx.x == 0)
  atomicAdd (&xsum, sum[0]); // Atomic reduction

```

NMAX/2

Number of threads in a block

Number of elements to process = 2*BLOCK_SIZE_T

Hands on exercise #1

- Copy all the exercises to your home directory (ignore error messages):

```
cp -pr ~syam/CUDA_day2/ ~
```
- Use one of the nodes listed in ~/CUDA_day2/nodes.txt

```
ssh monXX  
export CUDA_VISIBLE_DEVICES=Y
```
- Text editors: vim, emacs, nano (for syntax highlighting:

```
cp ~syam/.nanorc ~
```

)
- Help material (PDF files in ~/CUDA_day2).
- Exercises:
 - [CUDA_day2 / Reduction](#): implementing hybrid reduction scheme

C language extensions

- Concurrent execution and streams
 - Concurrency (parallel execution) between GPU and CPU is either a default, or easily enabled behaviour
 - Kernel launches are always asynchronous with regards to the host code; one has to use explicit device-host synchronization any time a kernel needs to be synchronized with the host:
CudaDeviceSynchronize ()
 - The default behaviour of GPU<->CPU memory copy operations is asynchronous for small transfers (<64kB), and synchronous otherwise. But one can enforce any memory copying to be asynchronous by adding *Async* suffix, e.g.:
 - `cudaMemcpyAsync ()`
 - `cudaMemcpyToSymbolAsync ()`
 - For debugging purposes, one can enforce everything to be synchronous by setting the `CUDA_LAUNCH_BLOCKING` environment variable to 1.

C language extensions

- Concurrent execution and streams
 - Concurrency between different device operations (kernels and/or memory copying) is a completely different story
 - On a hardware level, modern GPUs are capable of running multiple kernels and memory transfers both to and from the device concurrently
 - By default, everything on device is done serially (no concurrency)
 - To make use of the device concurrency features, one has to start using multiple **streams** in the CUDA code
 - But even with multiple streams, there are some limitations to concurrency on GPU

C language extensions

- Concurrent execution and streams
 - A stream is a sequence of commands (possibly issued by different host threads) that execute in order
 - If stream ID is omitted, it is assumed to be “0” (default) stream. For non-default streams, the IDs have to be used explicitly.
 - For concurrent memory copying on GPU, one has to both add the Async suffix and specify the stream ID.

```
mykernel <<<Nblocks, Nthreads, 0, ID>>> ();
```

```
cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID);
```

C language extensions

- Concurrent execution and streams
 - Before using, streams have to be created. At the end, they have to be destroyed

```
// Host code
```

```
cudaStream_t ID[2];
```

```
// Creating streams:
```

```
for (int i = 0; i < 2; ++i)  
    cudaStreamCreate (&ID[i]);
```

```
// These two commands will run concurrently on GPU:
```

```
mykernel <<<Nblocks, Nthreads, 0, ID[0]>>> ();  
cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID[1]);
```

```
// Destroying streams:
```

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


C language extensions

- Concurrent execution and streams
 - Limitations:
 - For memory copying operations to run concurrently with any other device operation (kernel or another memory copying operation), the host memory has to be *page-locked* (or *pinned*; allocated with [cudaMallocHost](#) instead of `malloc`; static variables can be made pinned using [cudaHostRegister](#))
 - Up to 16 kernels can run concurrently
 - Concurrency on GPU is not guaranteed (e.g., if kernels use too much local resources, they will not run concurrently)

C language extensions

- Concurrent execution and streams
 - Other stream-related commands
 - `cudaDeviceSynchronize()` : global synchronization (across all the streams and the host);
 - `cudaStreamSynchronize (ID)` : synchronize stream ID with the host;
 - `cudaStreamQuery (ID)` : tests if the stream ID has finished running.

Question

CUDA code optimization

CUDA code optimization

- Converting a code to CUDA can be considered an advanced exercise in code optimization
 - You should start profiling CUDA code from the very beginning, from the first kernel you write
 - You should start the conversion from the most cpu-intensive parts of the code
 - You often have to play with different approaches until you get the best performance in a given part of the code
- We will consider a few common optimization strategies

CUDA code optimization

- Kernels: how many?
 - You have to start a new kernel every time there is a global (across multiple blocks) data dependence
 - Example: two-level binary reduction shown previously
 - Another example: you need a separate kernel to initialize variables used to store an atomic reduction result:

```
// In global device memory:
```

```
__device__ double d_sum;
```

```
// On host:
```

```
// Initializing d_sum to zero:
```

```
init_sum <<<1, 1>>> ();
```

```
// Here d_sum is used to store atomic summation result from multiple blocks
```

```
compute_sum <<<Nblocks, BSIZE>>> ();
```

CUDA code optimization

- Kernels: how many?
 - You can try to split a kernel if it uses too many registers (*register pressure*)
 - It happens e.g. if the kernel has many complex algebraic expressions using lots of parameters
 - Register pressure can be identified when profiling the code with NVIDIA CUDA profilers (e.g., it will manifest itself via low *occupancy number*)
 - It is very non-intuitive: sometimes the register pressure can be decreased by making the kernel *longer* (presumably, because sometimes adding more lines of code gives CUDA compiler more flexibility to re-arrange register usage across the kernel)

CUDA code optimization

- Kernels: how many?
 - You should start a new kernel when there is a device-host dependence
 - Example:

```
// On host:  
  
kernel1 <<<N, M>>> ();  
  
CudaDeviceSynchronize ();  
  
// Host code dependent on kernel1 results:  
library_function1 ();
```


CUDA code optimization

- Kernels: how many?
 - Otherwise, you should try to make kernels as large as possible
 - Because each kernel launch has an overhead, in part because one has to store and then read the intermediate results from a slow (device or host) memory
 - You shouldn't worry that the kernel code won't fit on GPU: modern GPUs have a limit of 512 million instructions per kernel
 - To improve readability, parts of the kernel can be modularized into device functions

CUDA code optimization

- What should be computed on GPU?
 - You start with the obvious targets: cpu-intensive data-parallel parts of the code
 - What should you do with the leftover code (not data-parallel and/or not very cpu intensive)?
 - If not a lot of data needs to be copied from device to host and vice versa for the leftover code, it may be beneficial to leave these parts of the code on host
 - If on the other hand the leftover code needs access to a lot of intermediate results from CUDA kernels, then it may be more efficient to move (almost) everything to the GPU – even purely serial (single-thread) computations. This way, no intermediate (scratch) data will ever need to leave GPU.

CUDA code optimization

- Moving leftover code to GPU

```

// On host:

// First chunk of data-parallel code goes here:
kernel1 <<<N, M>>> ();

// Copying kernel1 results to host:
cudaMemcpy (h_A, d_A, size,
            cudaMemcpyDeviceToHost);
CudaDeviceSynchronize ();

// Non-parallelizable part of the code:
serial_computation (h_A, h_B);

// Copying serial_computation results to device:
cudaMemcpy (d_B, h_B, size,
            cudaMemcpyHostToDevice);
// CudaDeviceSynchronize (); - Why?

// Second chunk of data parallel code which depends on
d_B:
kernel2 <<<N, M>>>

```

```

// On host:

// First chunk of data-parallel code goes here:
kernel1 <<<N, M>>> ();

// Now it is executed on GPU, serially:
serial_computation_kernel <<<1, 1>>> ();

// Second chunk of data parallel code which depends on
d_B:
kernel2 <<<N, M>>>

```

Question

CUDA code optimization

- Optimizing memory copying between GPU and CPU
 - GPU - device memory bandwidth is much (~20x) larger than GPU - host memory bandwidth
 - As a result, minimizing amount of data copied between GPU and CPU should be a high priority
 - One possible solution is described on the previous slide: move to GPU the “leftover” code (even if it poorly performs on GPU) if it helps to cut significantly on GPU-CPU memory copying

CUDA code optimization

- Optimizing memory copying between GPU and CPU
 - Sometimes one can reduce or eliminate the time spent on GPU-CPU data copying if it is done in parallel (asynchronously) with **host** computations:

```
// On host:
```

```
// This memory copying will be asynchronous only in regards to the host code:
```

```
cudaMemcpyAsync (d_a, h_a, size, cudaMemcpyHostToDevice, 0);
```

```
// This host code will be executed in parallel with memory copying
```

```
host_computation ();
```

CUDA code optimization

- Optimizing memory copying between GPU and CPU
 - One can also run memory transfer operation concurrently with another (opposite direction) memory transfer operation, or a kernel. For that, one has to create and use streams.
 - Only works with pinned host memory

```
// This memory copying will be asynchronous in regards to the host and stream ID[1]:  
cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, ID[0]);
```

```
// The kernel doesn't need d_a, and will run concurrently with the previous line:  
kernel1 <<<N, M, 0, ID[1]>>> ();
```

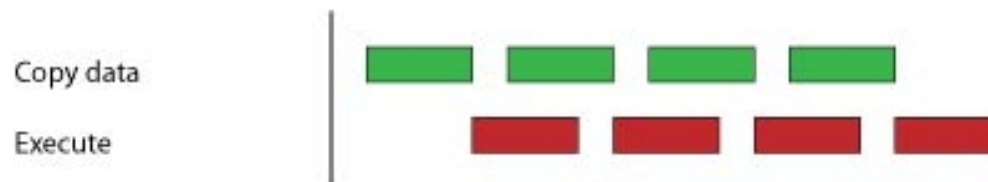
CUDA code optimization

- Optimizing memory copying between GPU and CPU
 - Staged concurrent copy and execute

One stream scenario:



You need two streams for this:



Hands on exercise #2

- Copy all the exercises to your home directory (ignore error messages):

```
cp -pr ~syam/CUDA_day2/ ~
```
- Use one of the nodes listed in ~/CUDA_day2/nodes.txt

```
ssh monXX  
export CUDA_VISIBLE_DEVICES=Y
```
- Text editors: vim, emacs, nano (for syntax highlighting:

```
cp ~syam/.nanorc ~
```

)
- Help material (PDF files in ~/CUDA_day2).
- Exercises:
 - [CUDA_day2 / Staged](#): using streams to stage copying and computing

CUDA code optimization

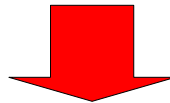
- Optimizing memory copying between GPU and CPU
 - To save on memory copying overheads, one should try to bundle up multiple small transfers into one large one
 - This can be conveniently achieved by creating a single structure, with the individual memory copying arguments becoming elements of the structure

CUDA code optimization

- Optimizing memory copying between GPU and CPU

// Host code:

```
cudaMemcpyToSymbol (d_A, &h_A, sizeof(h_A), 0, cudaMemcpyHostToDevice);  
cudaMemcpyToSymbol (d_B, &h_B, sizeof(h_B), 0, cudaMemcpyHostToDevice);  
cudaMemcpyToSymbol (d_C, &h_C, sizeof(h_C), 0, cudaMemcpyHostToDevice);
```



// Header file:

```
struct my_struct {  
    double A[1000];  
    double B[2000];  
    int C[1000];  
};  
__device__ struct my_struct d_struct;  
struct my_struct h_struct;
```

// Host code:

```
cudaMemcpyToSymbol (d_struct, &h_struct, sizeof(h_struct), 0, cudaMemcpyHostToDevice);
```

CUDA code optimization

- Optimizing memory copying between GPU and CPU
 - If you use dynamic memory allocation on host, you can usually accelerate copying to/from the device by using `cudaMallocHost` instead of `malloc`.
 - This will force the compiler to use page-locked memory for host allocations, which has much higher bandwidth to the device
 - Use this sparingly, as the performance can actually degrade when not enough of system memory is available for paging

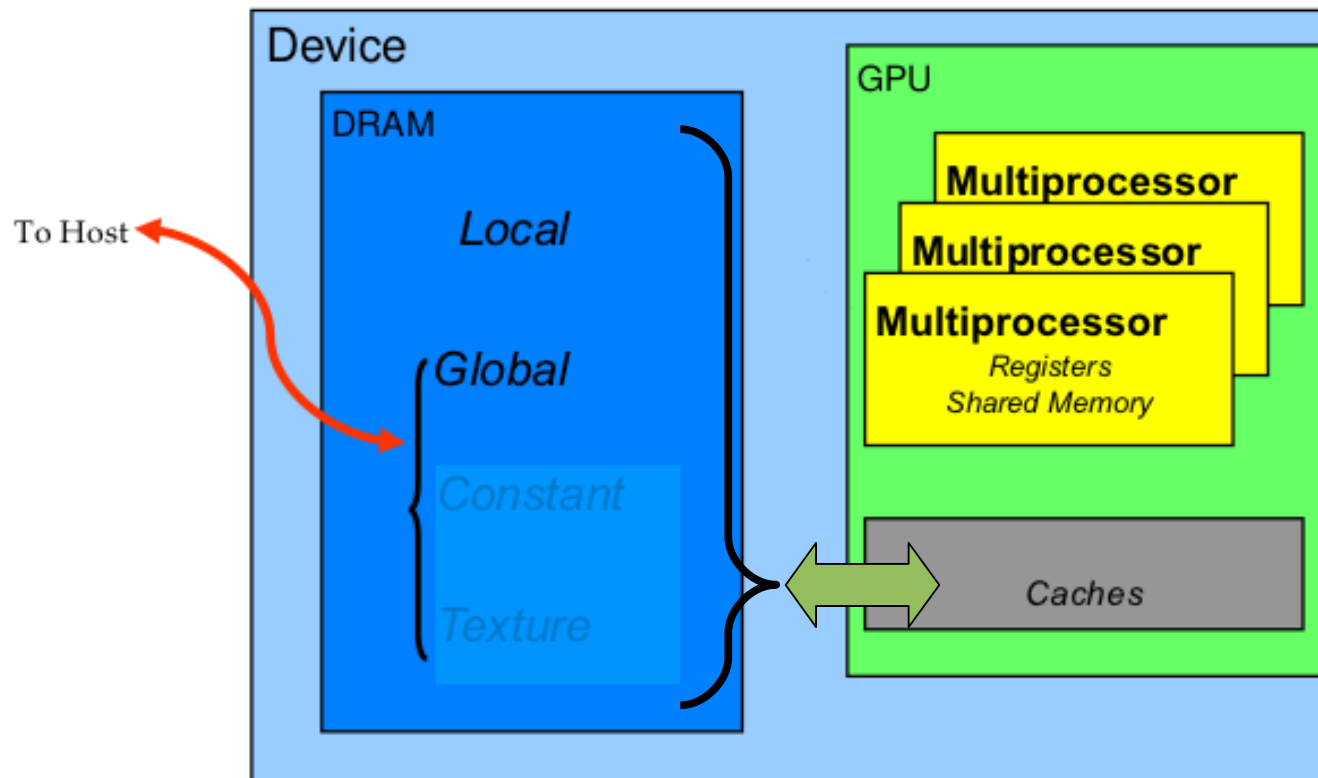
```
// Host code:
```

```
float *h_A;
```

```
cudaMallocHost (&h_A, N*sizeof(float));
```

CUDA code optimization

- Optimizing memory access on GPU
 - Memory spaces on GPU

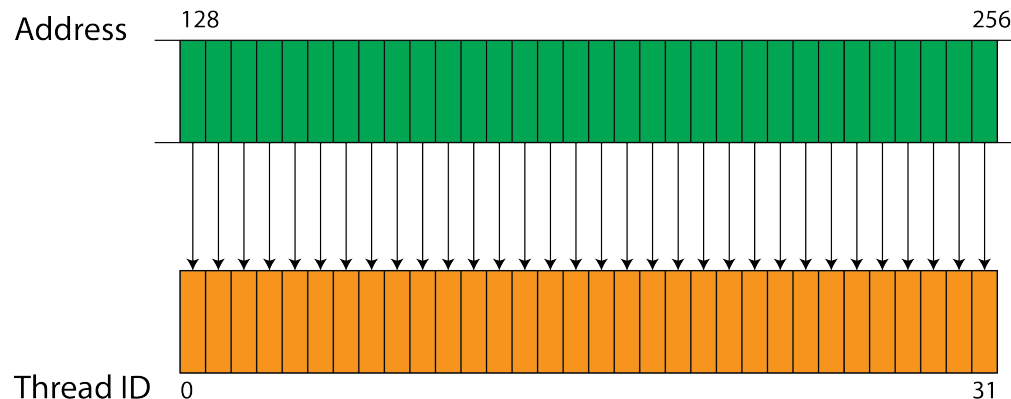


CUDA code optimization

- Optimizing memory access on GPU
 - Registers \leftrightarrow “Local” memory are not under your direct control, making it harder to optimize
 - Global and shared memory, on the other hand, are under direct programmer's control, so they are easier to optimize.
 - Main strategies for optimization:
 - Global memory: coalescence of memory accesses
 - Shared memory: minimizing bank conflicts

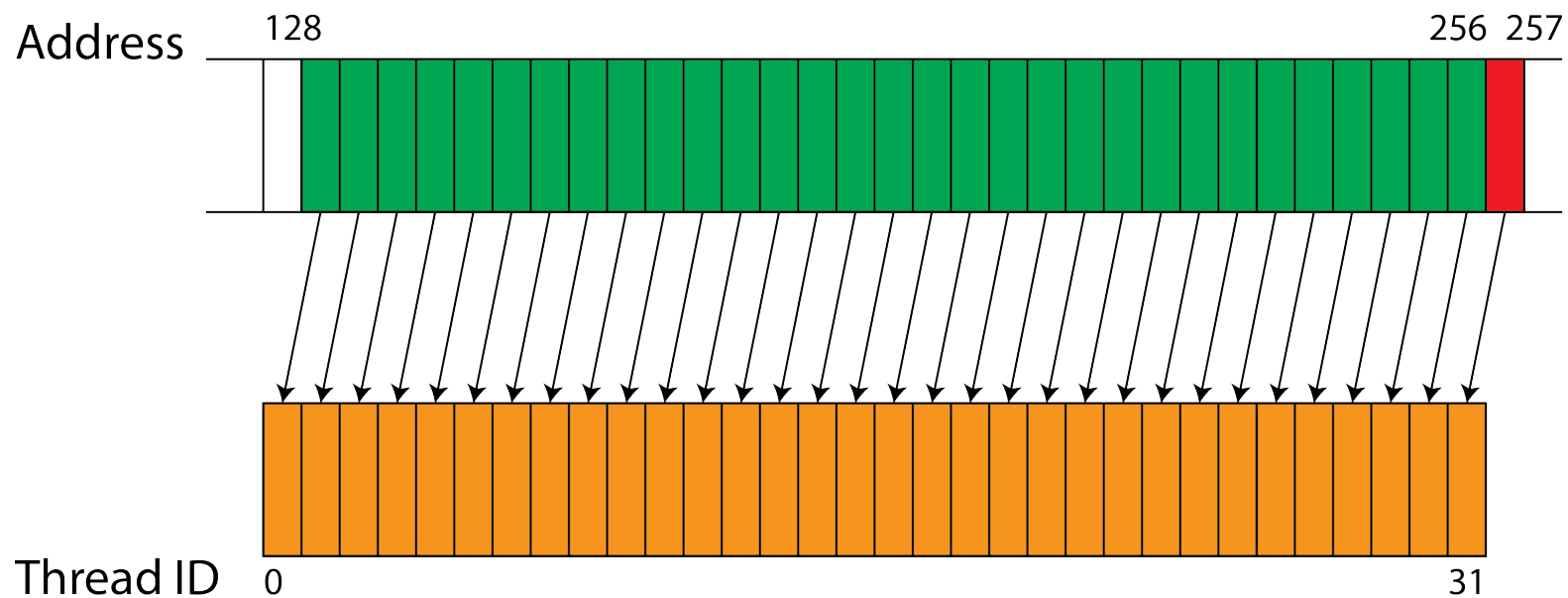
CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - Global memory loads and stores by threads of a warp are coalesced by the device into as few as one transaction when certain access requirements are met
 - By default, all accesses are cached through L1 as 128-byte lines
 - Coalescence is the best when accessing flat arrays (unit stride) consecutively.



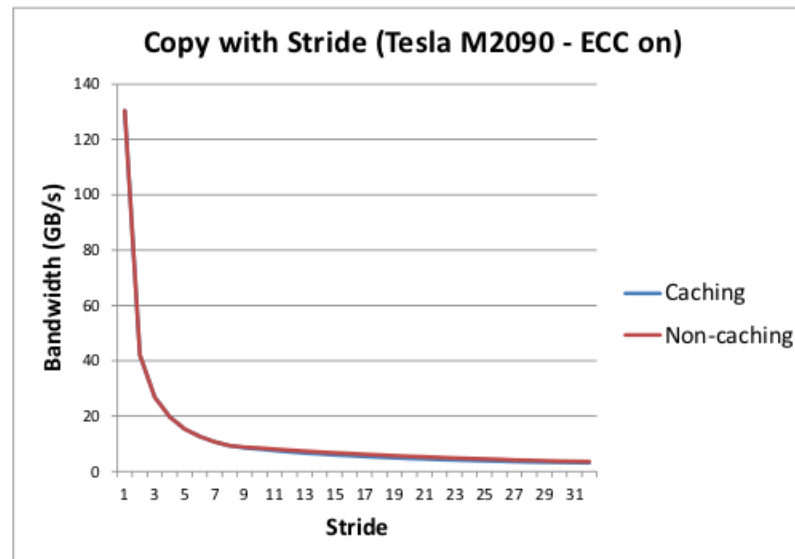
CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - Misaligned access degrades the performance, but not dramatically



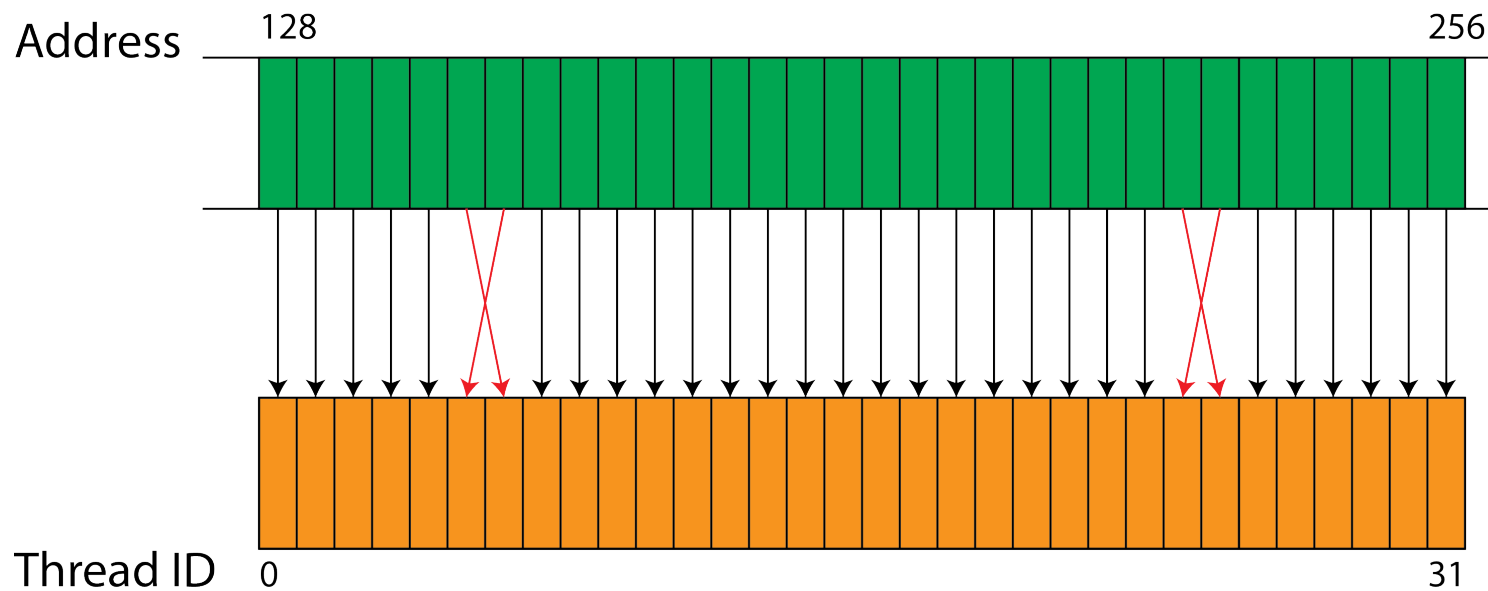
CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - Non-unit stride access (e.g. multi-D arrays), on the other hand, degrades the performance very rapidly, as the stride increases from 2 to 32:



CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - Any kind of non-sequential memory access (not just $\text{stride} > 1$) is bad*



CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - The strategy with multi-D arrays is to either
 - flatten them yourself (the only way if >3 dimensions), or
 - use special CUDA functions `cudaMallocPitch()` and `cudaMalloc3D()` to allocate properly aligned 2D and 3D arrays, respectively, or
 - at the very least, convert row-major arrays to **column-major** ones

CUDA code optimization

- Optimizing memory access on GPU

```
// Flattened using grids of blocks, good for 2-6D; N1 should be a multiple of 32

// On device:
__device__ float d_A[N1*N2*N3*N4];
__global__ void mykernel ()
{
  //           i1           i2           i3           i4
  int i = threadIdx.x+blockDim.x*(blockIdx.x+gridDim.x*(blockIdx.y+gridDim.y*blockIdx.z));
  d_A[i] = ...
}

// On host:
dim3 Nblocks (N4, N3, N2);
dim3 Nthreads (N1, 1, 1);
mykernel <<<Nblocks, Nthreads>>> ();
```

CUDA code optimization

- Optimizing memory access on GPU

```
// Flattened, good for any D; individual dimensions can be arbitrary
// On device:
#define N_TOTAL N1*N2*N3*N4
__device__ float d_A[N_TOTAL];
__global__ void mykernel (){
int i = threadIdx.x + blockDim.x * blockIdx.x;
if (i < N_TOTAL) {
    d_A[i] = ...
    // You compute individual indexes only if they are needed for the computations:
    int i1 = i % N1;  int m = i / N1;
    int i2 = m % N2;  m = m / N2;
    int i3 = m % N3;
    int i4 = m / N3;
}
}
// On host:
int Nblocks = (N_TOTAL + BLOCK_SIZE - 1) / BLOCK_SIZE;
mykernel <<<Nblocks, BLOCK_SIZE>>> ();
```

CUDA code optimization

- Optimizing memory access on GPU

```
// Using cudaMallocPitch, 2D case
```

```
// Host code
```

```
int width = 64, height = 64;
```

```
float* devPtr;
```

```
size_t pitch;
```

```
cudaMallocPitch (&devPtr, &pitch, width * sizeof(float), height);
```

```
MyKernel <<<64, 64>>> (devPtr, pitch);
```

```
// Device code
```

```
__global__ void MyKernel (float* devPtr, size_t pitch)
```

```
{
```

```
int ix = blockIdx.x;
```

```
int iy = threadIdx.x;
```

```
float* row = (float*)((char*)devPtr + ix * pitch);
```

```
float element = row[iy]; // Coalesced access
```

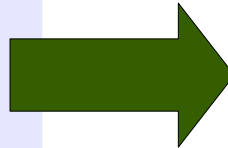
```
}
```

CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - If you have to use non-flattened static multi-D arrays, transpose them to “**column-major**” if they are “**row-major**”:

```
// Row-major (non coalesced)
```

```
float A[N][30];  
...  
A[threadIdx.x][0]=...;  
A[threadIdx.x][1]=...;
```



```
// Column-major (coalesced)
```

```
float A[30][N];  
...  
A[0][threadIdx.x]=...;  
A[1][threadIdx.x]=...;
```

CUDA code optimization

- Optimizing memory access on GPU
 - Global memory: coalescence of memory accesses
 - For the same reason, use **structures of arrays** instead of **arrays of structures** (the latter results in a memory access with a large stride)

```
// Array of structures behaves like row major accesses (non coalesced)  
struct Point { double x; double y; double z; double w; } A[N];  
...  
A[threadIdx.x].x = ...
```



```
// Structure of arrays behaves like column major accesses (coalesced)  
struct PointList { double *x; double *y; double *z; double *w; } A;  
...  
A.x[threadIdx.x] = ...
```


Question

CUDA code optimization

- Optimizing memory access on GPU
 - Using shared memory to optimize access to global memory
 - Shared memory is much faster than global memory; also, access to shared memory doesn't need to be coalesced
 - Shared memory can be viewed as a “user-managed cache for global memory”
 - One can store in shared memory frequently used global data
 - One can use shared memory to make reading data from global memory coalesced

CUDA code optimization

- Optimizing memory access on GPU

// Straightforward and inefficient way

```
__global__ void simpleMultiply(float *a, float* b, float *c, int
N)
{
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
for (int i = 0; i < TILE_DIM; i++) {
    sum += a[row*TILE_DIM+i] * b[i*N+col];
}
c[row*N+col] = sum;
}
```

*// Using shared memory to both store frequently used global
// data and to make the access coalesced – 2.3x faster on K20*

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int
N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                    bTile[TILE_DIM][TILE_DIM];

int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
aTile[threadIdx.y][threadIdx.x] =
    a[row*TILE_DIM+threadIdx.x];
bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
__syncthreads();
for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];
}
c[row*N+col] = sum;
}
```

CUDA code optimization

- Optimizing memory access on GPU
 - Shared memory: minimizing bank conflicts
 - Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks
 - A bank conflict only occurs if two or more threads access the same bank

```
// No bank conflicts for 32-bit data is when the stride is odd (s = 1, 3, ...)
```

```
__shared__ float shared[BLOCK_SIZE];  
float data = shared[BaseIndex + s * threadIdx.x];
```

```
// No bank conflicts for 64-bit data:
```

```
__shared__ double shared[BLOCK_SIZE];  
double data = shared[BaseIndex + threadIdx.x];
```

CUDA code optimization

- Minimizing warp divergence
 - The smallest independent execution unit in CUDA is a warp (a group of 32 consecutive threads in a block)
 - Within a warp, execution is synchronous (that is, warp acts as a 32-way vector processor)
 - Any flow control instruction (if, switch, do, for, while) acting on individual threads within a warp will result in **warp divergence** (with the different execution paths serialized), resulting in poor performance
 - Warp divergence minimization is hence an important CUDA optimization step

CUDA code optimization

- Minimizing warp divergence
 - Ideally, controlling conditions should be identical within a warp:

```
// On device:  
__global__ void MyKernel ()  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int warp_index = i / warpSize; // Remains constant within a warp  
  
    if (d_A[warp_index] == 0) // Identical execution path within a warp (no divergence)  
        do_one_thing (i);  
    else  
        do_another_thing (i);  
}
```

CUDA code optimization

- Minimizing warp divergence
 - As warps can't span thread blocks, conditions which are only a function of block indexes result in non-divergent warps

// On device:

```
__global__ void MyKernel ()  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    if (d_A[blockIdx.x] == 0) // No divergence, since warps can't span thread blocks  
        do_one_thing (i);  
    else  
        do_another_thing (i);  
}
```

CUDA code optimization

- Minimizing warp divergence
 - More generally, making a condition to span at least a few consecutive warps results in acceptably low level of warp divergences (even when the condition is not always aligned with warp boundaries)

```
// On device:  
__global__ void MyKernel ()  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int cond_index = i / N_CONDITION; // Is okay if N_CONDITION >~ 5*warpSize  
  
    if (d_A[cond_index] == 0) // Only a fraction of warps will have divergences  
        do_one_thing (i);  
    else  
        do_another_thing (i);  
}
```


CUDA code optimization

- Minimizing warp divergence
 - A frequently encountered warp divergence is a conditional premature thread termination:

// On device:

```
__global__ void MyKernel (int N_total)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i > N_total) // Needed for the last (incomplete) block
        return; // Premature thread termination
    ...
}
```

// On host:

```
int N_total;
int Nblocks = (N_total + BLOCK_SIZE - 1) / BLOCK_SIZE;
mykernel <<<Nblocks, BLOCK_SIZE>>> ();
```

CUDA code optimization

- Accuracy versus speed
 - Situation with double precision speed in CUDA improved dramatically in the recent years, but it is still slower than single precision
 - The ratio was 1:8 for capability 1.3 (old cluster angel)
 - The newer cluster monk (capability 2.0) has the ratio 1:2
 - But newest NVIDIA GPUs seem to be moving in the “wrong” direction (1:3 for K20; 1:32 for Maxwell)
 - Use double precision only where it is absolutely necessary

CUDA code optimization

- Optimal CUDA parameters
 - **Number of threads per block** (BLOCK_SIZE): total range 1...1024; much better if multiples of 32; better still if multiples of 64.
 - **Number of threads per multiprocessor**: at least 768 for capability 2.x to completely hide read-after-write register latency. That means at least 10,752 threads per kernel for the whole monk GPU.
 - **Number of blocks in a kernel**: at least equal to the number of multiprocessors (≥ 14 for monk), to keep all multiprocessors busy.

New NVIDIA GPUs

New NVIDIA GPU – K20/40/80

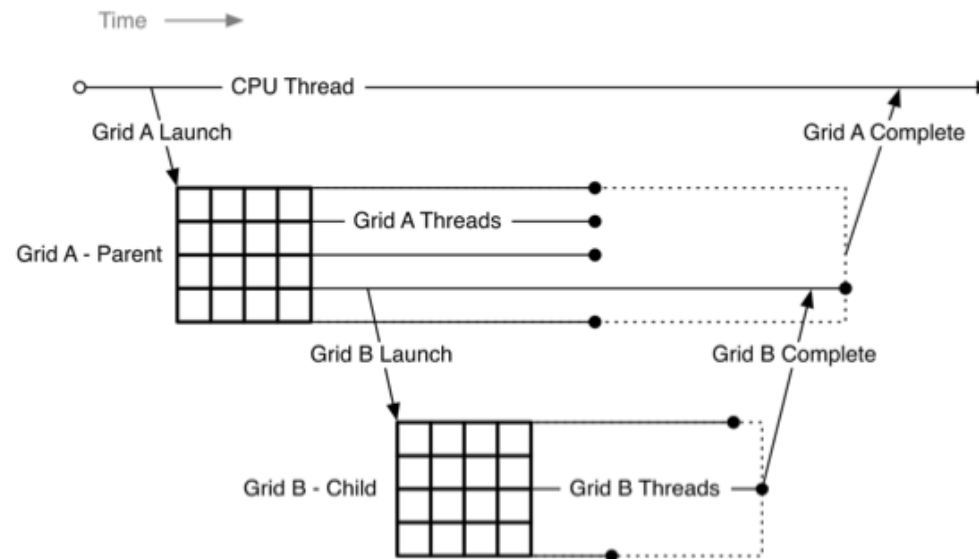
- KXX Tesla GPUs represent both a significant evolutionary and revolutionary changes.
 - Evolutionary:
 - thanks to many more cores (2688 for K20 vs. 448 for C2075), 3.8x more flops for SP, 2x more flops for DP
 - 70% faster core-memory bandwidth (for K20; 250 vs. 150 GB/s)
 - Revolutionary:
 - **CUDA Dynamic Parallelism** (CDP): new hard/software feature allowing for dynamic workload generation on GPU (kernels launched from kernels). Makes GPU much more general purpose computing device.
 - **Hyper-Q**: in previous generations, multiple CPU threads could only access the GPU sequentially (one queue); K20 expands that to 32 parallel queues. This should significantly accelerate mixed MPI/CUDA and OpenMP/CUDA codes, without any code modifications.

New NVIDIA GPU – K20/40/80

- These changes made K20 the first real HPC GPU
 - #2 supercomputer in the world (Titan) has 90% of its 27 Petaflops in K20s. (Consists of almost 19,000 nodes, 16 CPU cores + K20x in each node.)
 - Multiple Tier 1 software packages have shown a significant, ~4x, speedup (“16 CPU cores + K20” versus “16 CPU cores”), thanks to new K20 capabilities.
 - Cosmology code Enzo: with Hyper-Q, speedup increased from 1.4x to 6x (on Titan).
 - N-body tree-code Bonsai: Dynamic Parallelism resulted in ~2x faster code.
- A few K20s and K80s are available in SHARCNET, but only in contributed systems. Calcul Quebec's Guillimin has 100 K20s.

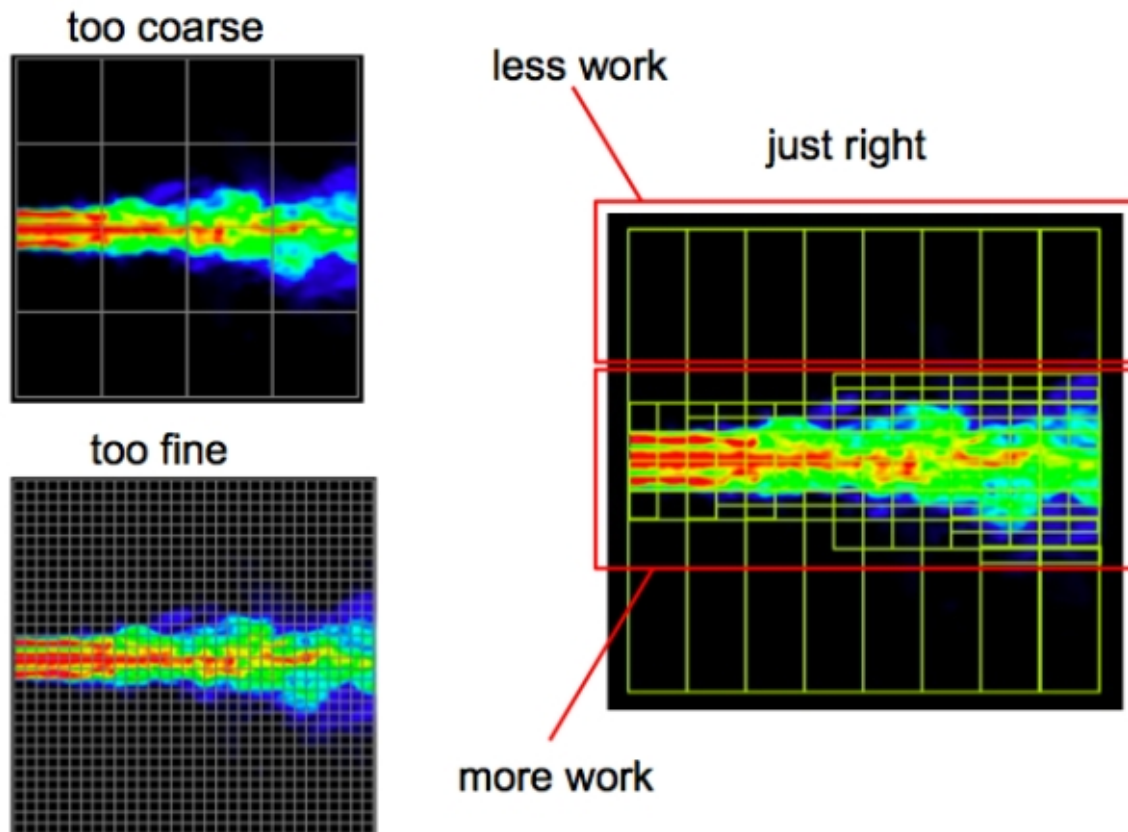
Dynamic Parallelism

- Dynamic parallelism (DP) is available in CUDA 5.0 and later on devices of Compute Capability 3.5 or higher (sm_35).
- Under DP, an application can launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed.



Dynamic Parallelism

- DP is perfect for adaptive grid codes and codes with recursion.



DP: simple example

- DP allows one to move almost everything to GPU.

// On device:

// Second level kernels (multi-threaded):

```
__global__ void kernel1 (){}  
__global__ void kernel2 (){}
```

// Top level kernel (single-threaded):

```
__global__ void main_kernel (){  
    if (threadIdx.x == 0) {
```

// These second level kernels will run sequentially (would need streams for concurrency)

```
        kernel1<<<Nblocks, Nthreads>>>();  
        kernel2<<<Nblocks, Nthreads>>>();
```

```
        ...  
    }  
}
```

// On host:

```
int main() {  
    main_kernel<<<1,1>>>();}
```

Hands on exercise #3

- Copy all the exercises to your home directory (ignore error messages):

```
cp -pr ~syam/CUDA_day2 ~
```
- Use one of the nodes listed in ~/CUDA_day2/nodes.txt

```
ssh monXX  
export CUDA_VISIBLE_DEVICES=Y
```
- Text editors: vim, emacs, nano (for syntax highlighting:

```
cp ~syam/.nanorc ~
```

)
- Help material (PDF files in ~/CUDA_day2).
- Exercises:
 - [CUDA_day2 / Primes](#): converting a serial code for the largest prime number search to CUDA