GPU programming with CUDA

Igor Yakushin
ivy2@uchicago.edu

February 15, 2019

How to get the tutorial and log into midway 2

- Point your browser to
  `https://git.rcc.uchicago.edu/ivy2/cuda`
- If you have RCC account and allocation, login to midway 2 as usual:
  `ssh -Y <username>@midway2.rcc.uchicago.edu`
  - For more details, see:
    `https://rcc.uchicago.edu/docs/connecting/index.html`
- If you do not have RCC allocation, use yubikey:
  `https://git.rcc.uchicago.edu/ivy2/yubikey`
- Once on midway 2, clone the tutorial with the labs:
  `git clone https://git.rcc.uchicago.edu/ivy2/cuda.git`
- The labs are in `cuda/labs`
- The presentation is in `cuda/docs/CUDA.pdf`

Why to use parallel programming?

- Until 2005, CPU frequency kept doubling every year or so and your old sequential program would automatically run faster on the new hardware without you having to do anything.
- This is no longer true: we cannot keep increasing CPU frequency since we cannot handle so much heat per unit area.
- As a result, during recent decade the progress in computing hardware was in increasing degree of parallelism by using wider vectors, more CPU cores, offloading computations to accelerators such as GPUs, TPUs, FPGAs, using faster memory, increasing bandwidth between memory and CPU or CPU and accelerators, using faster interconnect between nodes in a cluster to be utilized, for example, by MPI or other distributed computing framework.
- Your old sequential program will not run any faster on the new hardware unless you redesign it to utilize the available parallelism.
- Offloading part of computation to GPU is one way to speed up your program, potentially by 10^n.

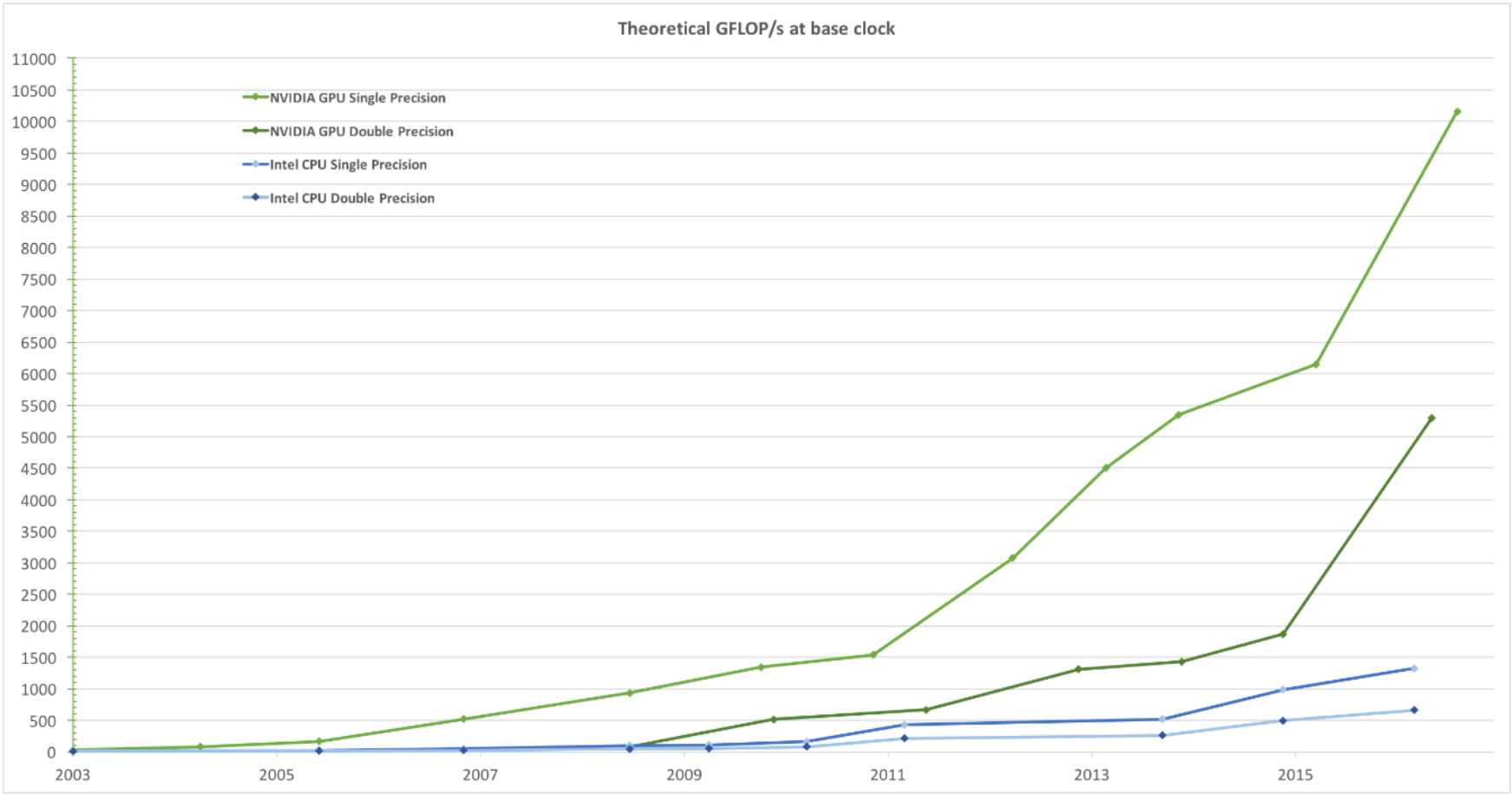


Figure 1 Floating-Point Operations per Second for the CPU and GPU

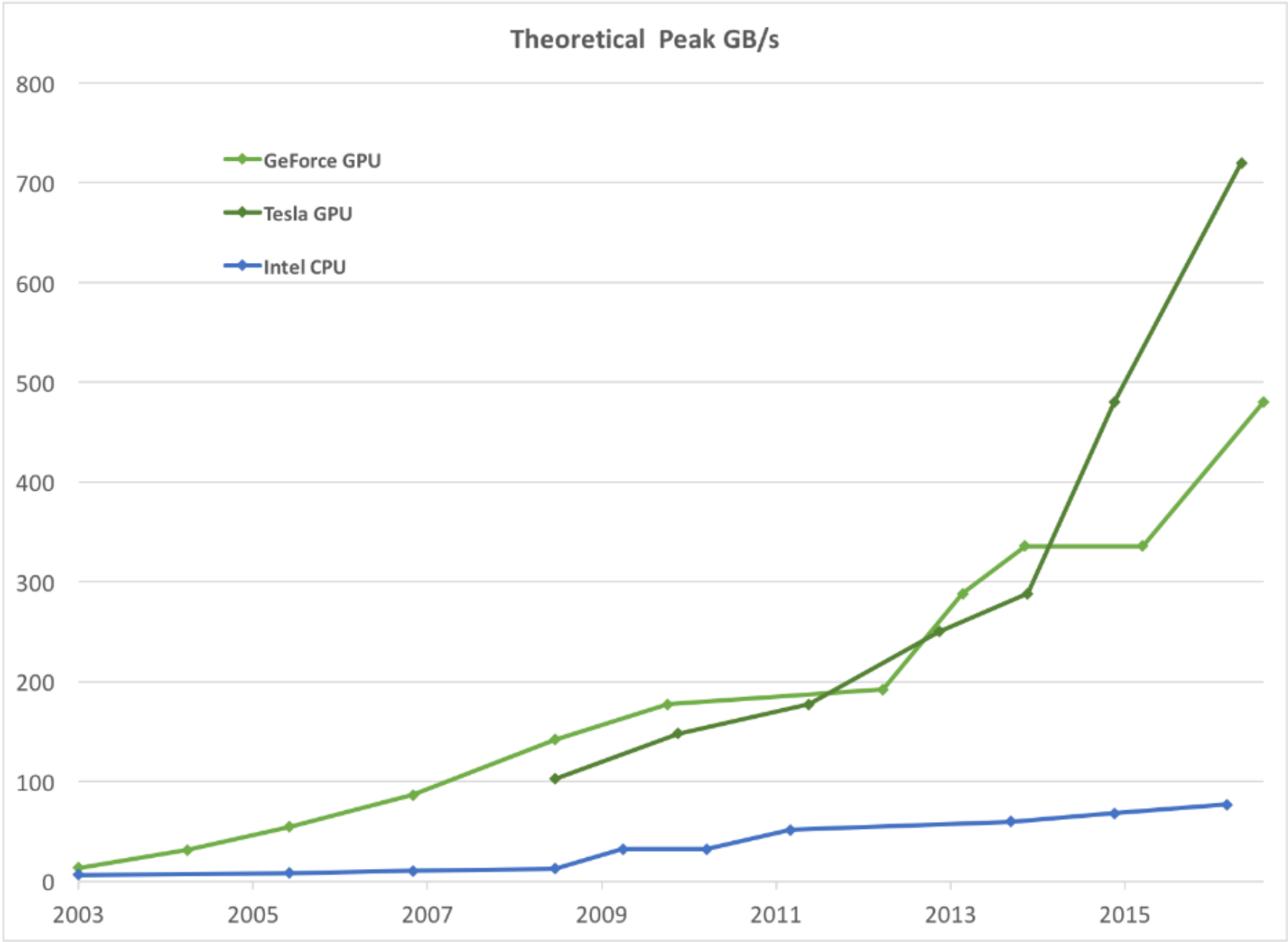


Figure 2 Memory Bandwidth for the CPU and GPU

Scaling: strong

- By how much can one speed up a program using N processors, if the program spends P fraction of its time in the part of the code that can be parallezed?
- Assuming the best case of linear scaling:

$$speedup = \frac{P + (1 - P)}{\frac{P}{N} + (1 - P)} = \frac{1}{\frac{P}{N} + (1 - P)}$$

- The above equation is called Amdahl's law, it describes strong scaling - how faster can one do a fixed amount of work given more processors.
- Obviously, the bigger is P, the more sense it makes to spend efforts parallelizing the corresponding part of the code.
- For example, assuming the best case $N = \infty$ and linear scaling, if $P = 0.99$, $speedup = 100$; however, if $P = 0.5$, $speedup = 2$.
- In reality, linear scaling is too optimistic and is often limited by communication overhead, synchronization, memory bandwidth, etc.

Scaling: weak

- Often we are interested in a different question: given more processors, how much more work can we do during the fixed amount of time? This is called weak scaling and is described by Gustafson's law.

$$morework = \frac{P * N + (1 - P)}{P + (1 - P)} = P * N + (1 - P)$$

- 9 women producing 1 baby in 1 month - strong scaling
- 9 women producing 9 babies instead of 1 baby in 9 months - weak scaling
- Again, linear increase of the amount of work with the number of available processors is usually too optimistic: there will likely be bottlenecks in communication bandwidth, synchronization, etc. that would limit strong scaling. For example, if there are 4 lanes in the road, only 4 cars can travel in parallel.

GPU hardware: thread, warp, block, grid, SM

- GPU consists of several Streaming Multiprocessors (SMs)
- Each SM can run several thousand GPU threads in parallel
- For example, V100 has 80 SMs, K80 has 13 SMs. Each SM in those can run up to 2048 threads. Therefore up to 163840 threads can run in parallel in V100 and up to 26624 threads in K80.
- GPU cores have much simpler logic than CPU core: do not do branch prediction, less caching

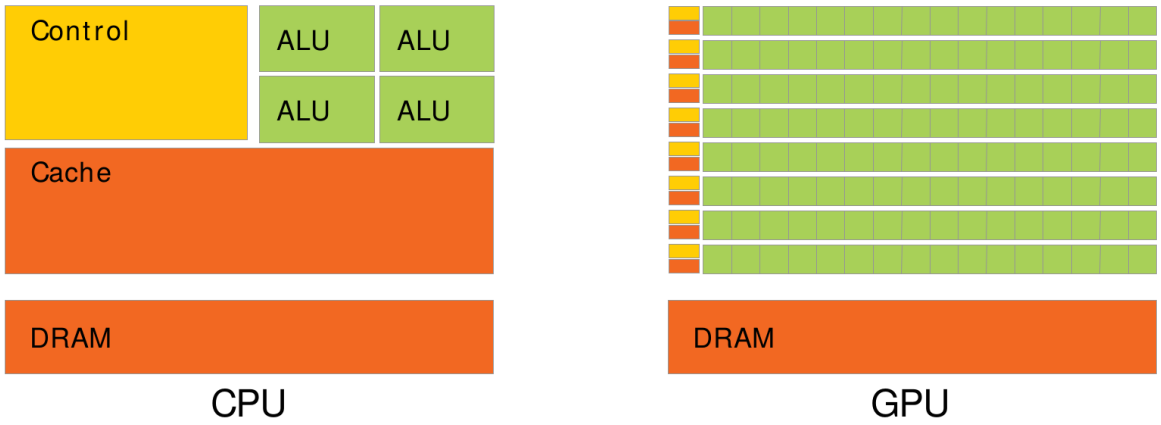
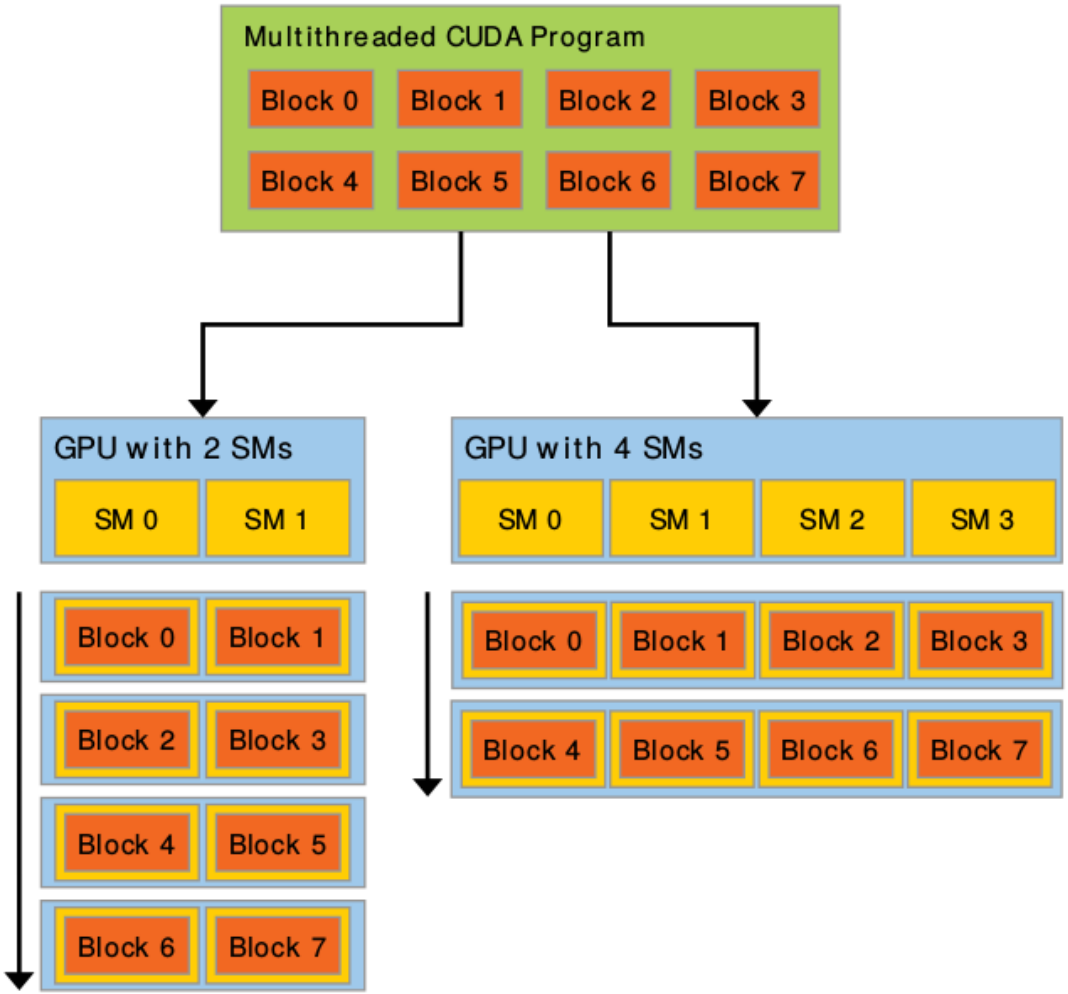


Figure 3 The GPU Devotes More Transistors to Data Processing

- Each GPU core is a few times slower than CPU core. For example, V100's clock frequency is 1.530 GHz, K80's - 0.824 GHz, while typical CPU clock frequencies today are around 2.5 - 3.0 GHz.
- The threads in a job are grouped into a grid of blocks
- Each block can have up to 1024 threads
- Blocks are scheduled to run on SMs in no particular order, in parallel or sequentially, on the same SM or on different SMs
- The whole block runs on one SM
- Inside a block, threads are grouped into warps
- In a warp, there are 32 threads
- Hardware executes each warp as a single entity
- It is best if threads in a warp do not diverge but do the same instruction at each step
- In case of a thread divergence in a warp due to `if` statements or different lengths of loops, the divergent parts are executed sequentially
- This architecture of GPU is called SIMT - Single Instruction Multiple Threads



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 5 Automatic Scalability

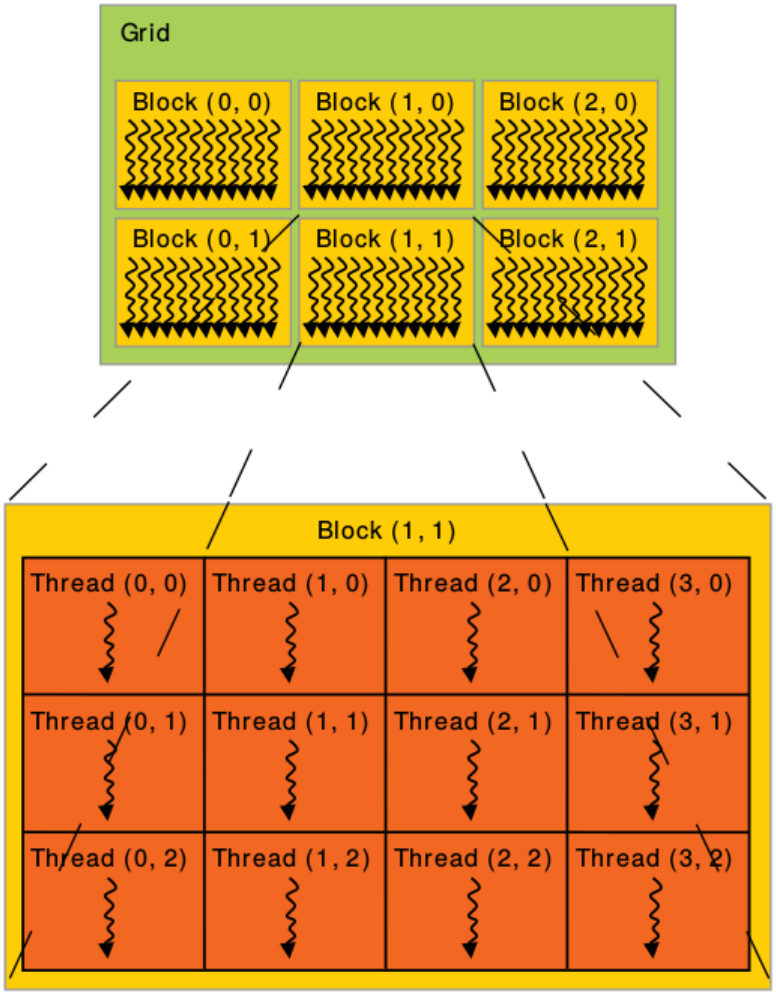
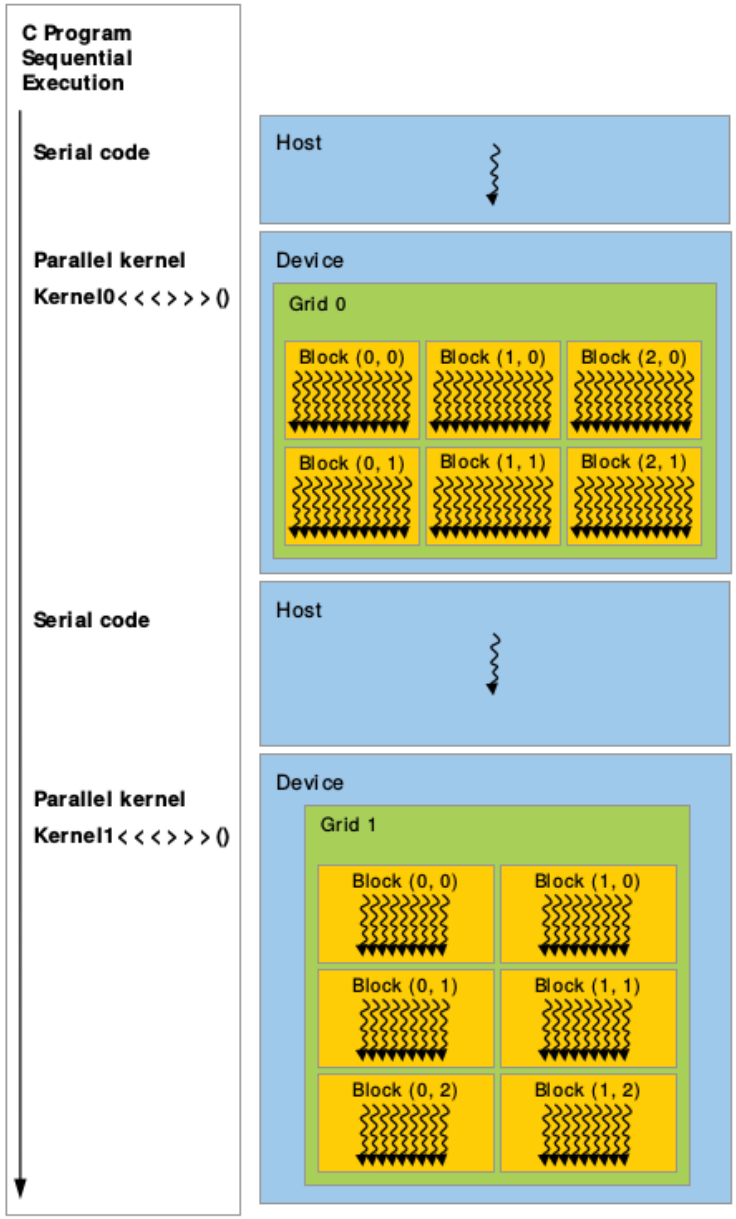


Figure 6 Grid of Thread Blocks



Serial code executes on the host while parallel code executes on the device.

Figure 8 Heterogeneous Programming

GPU hardware: memory

- GPU uses its own memory - global memory
- One might have to copy data between CPU host and GPU device.
- K80 has 12G, V100 - 16G
- The bandwidth between global memory and GPU for K80 is $480 GB/s$, for V100 - $900 GB/s$
- For comparison: the typical bandwidth between CPU and RAM is about $100 GB/s$
- GPU card is typically connected to CPU host with PCI interface with a bandwidth of about $12 GB/s$, for PCIe gen3 - $32 GB/s$
- There are faster interconnects available, for example, NVlink - $300 GB/s$, but so far they are too expensive and not yet widespread and transferring data between CPU and GPU is likely to be the biggest bottleneck for a while
- Therefore to get better performance, one needs to minimize moving data between the host and device or interleave it with the computations to hide the latency

GPU hardware: memory

- Given the number of threads, global memory is slow and one needs to minimize its usage as well
- One way to optimize usage of global memory is coalescing: take into account that data is read from global memory in aligned cache lines and in ideal close threads should read close data
- Per block, there is faster shared memory, limited to $48k$ for K80 and $96k$ for V100. It can be used to cache global memory values manually.
- Local variables in a thread are kept in registers which are faster than shared memory. However, there is limited amount of registers available and some local variables might spill into local memory which physically resides in the same place as global memory and therefore is as slow.
- Constant memory is optimized for read operations although physically it is still the same as global memory.
- Texture and surface memory are used mostly for graphics programming and we shall not discuss it here.

GPU hardware: memory

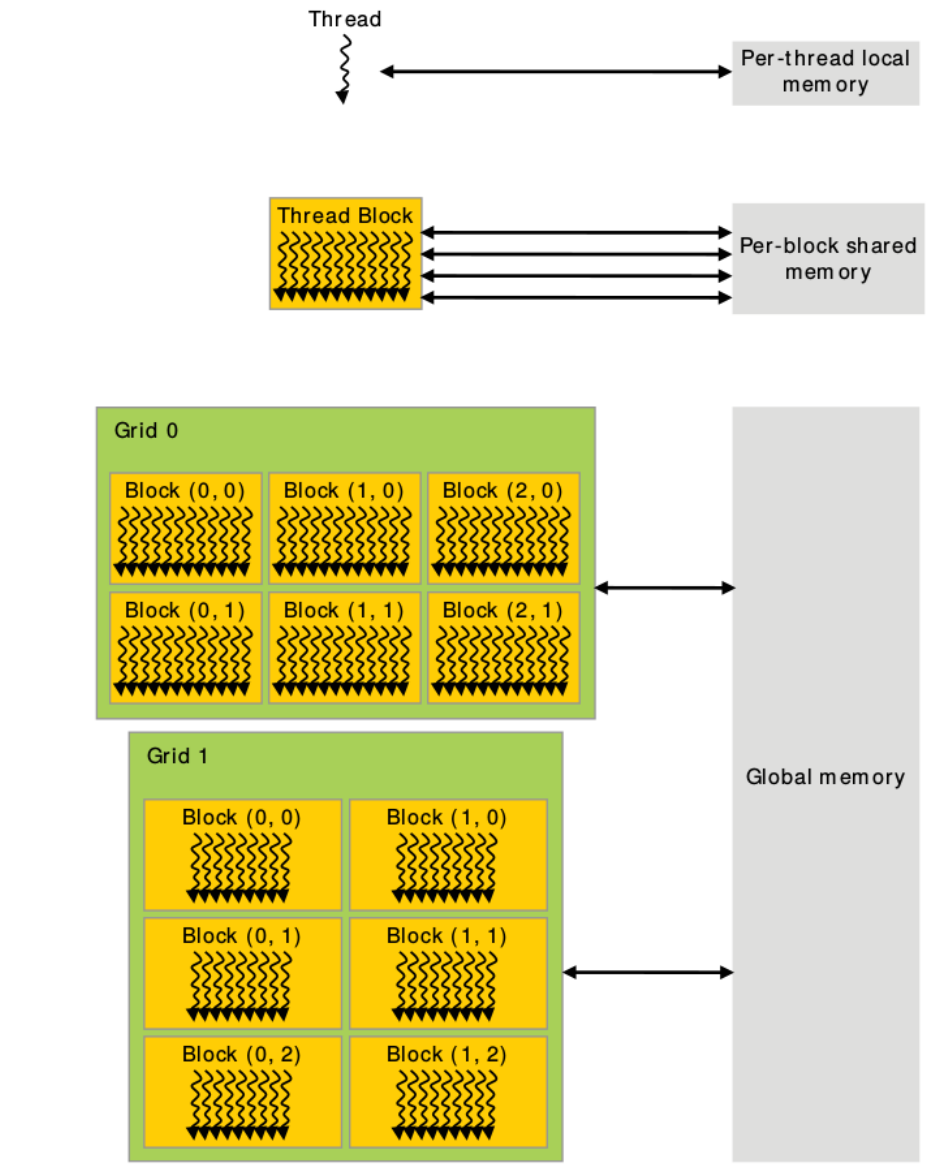


Figure 7 Memory Hierarchy

- The host can only copy data to/from global, constant, texture, surface memory; not shared memory, registers or local memory
- For optimal performance, it is important to align data in memory since reads and writes start at aligned address
- Builtin types are aligned automatically but user defined types like structures might have to be aligned manually

GPU hardware: memory

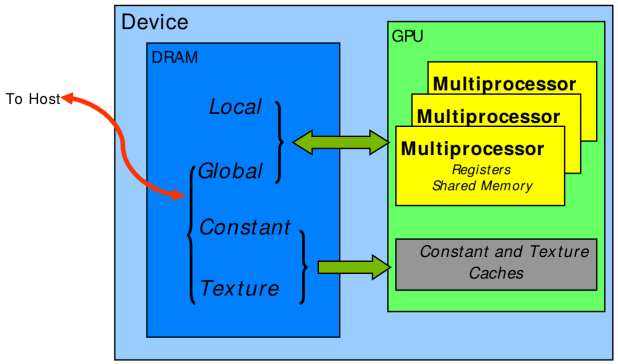


Figure 2 Memory spaces on a CUDA device

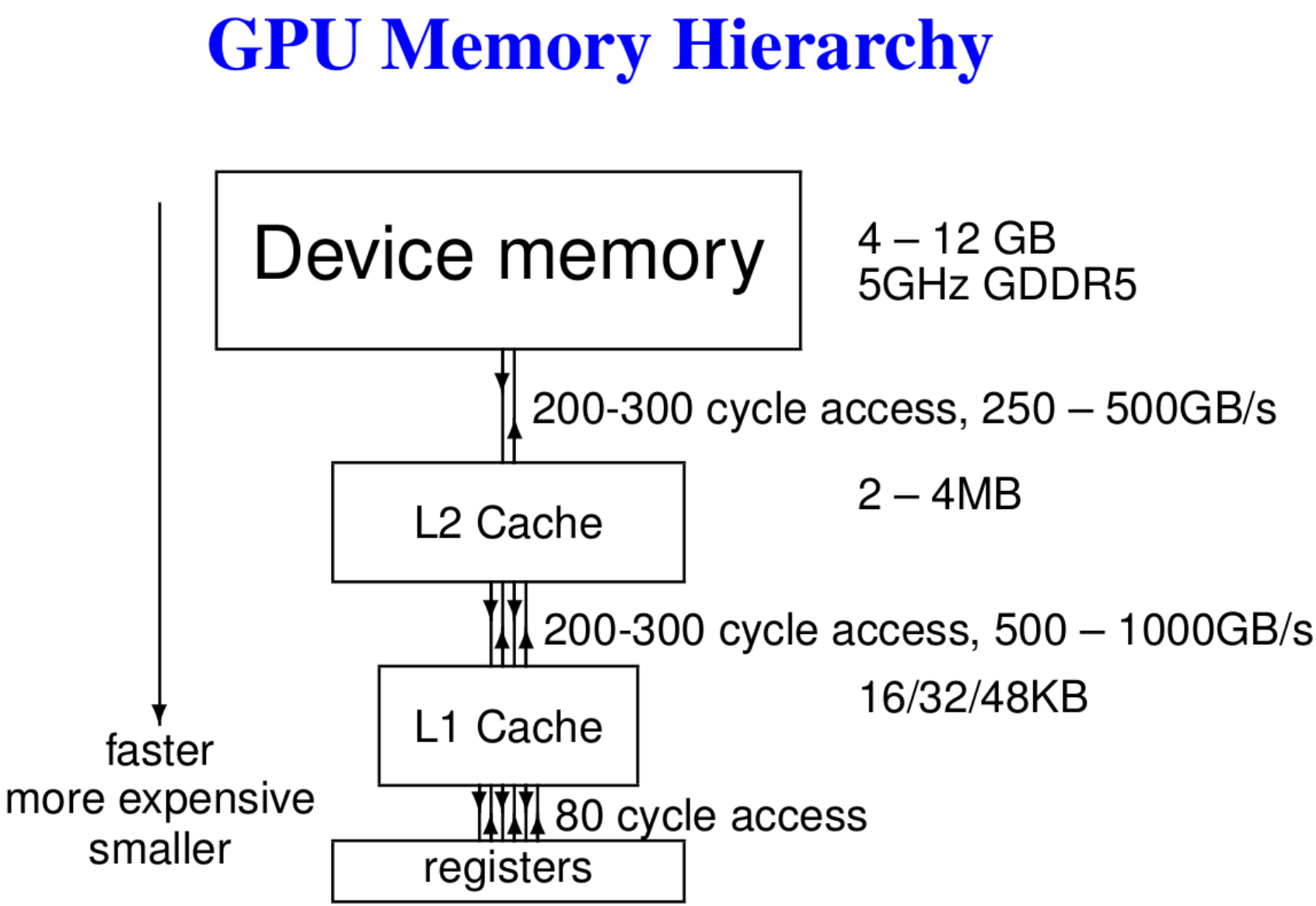
- Unified Memory, available for $CC \geq 3.0$, allows using the same pointer both on the host and device and makes data transfer transparent for the user.

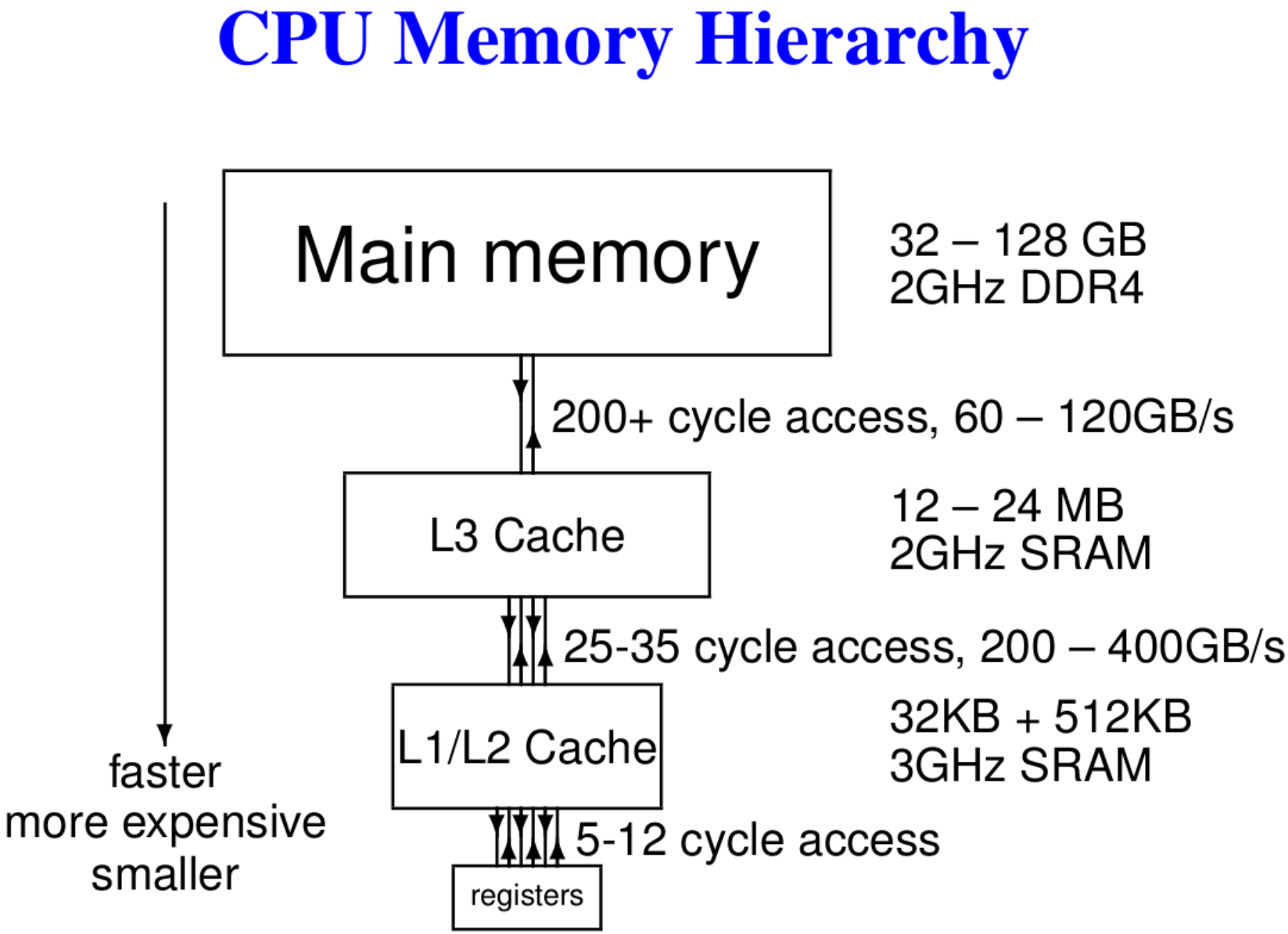
Table 1 Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.


GPU Memory Hierarchy
Device memory
4 – 12 GB
5GHz GDDR5
200-300 cycle access, 250 – 500GB/s
L2 Cache
2 – 4MB
200-300 cycle access, 500 – 1000GB/s
L1 Cache
16/32/48KB
80 cycle access
registers
faster
more expensive
smaller


CPU Memory Hierarchy
Main memory
32 – 128 GB
2GHz DDR4
200+ cycle access, 60 – 120GB/s
L3 Cache
12 – 24 MB
2GHz SRAM
25-35 cycle access, 200 – 400GB/s
L1/L2 Cache
32KB + 512KB
3GHz SRAM
5-12 cycle access
registers
faster
more expensive
smaller

GPU hardware: Compute Capability

- Compute Capability defines a set of features that the program can use
- GPU cards are advertised to support particular compute capability
- K80 has compute capability 3.7, while V100 is compute capability 7.0 and the latest Turing architecture supports 7.5
- When you compile binary that would run on GPU, you specify on the features from what compute capability your code relies
- You can also specify for what particular hardware you build the code and save binaries for different GPU cards in the same executable
- Executable also contains so called PTX code which is basically an assembler code for GPU hardware that supports the specified compute capability.
- PTX allows JIT - just in time compilation: if you try to run your binary on the card for which the code was not compiled it gets recompiled at run time provided that the card is backward compatible with the specified compute capability.

GPU hardware: Generations of GPU cards

- For each compute capability NVIDIA releases many cards for different usage mode: gaming, graphics, laptops, desktops, tablets, data center, etc
- They differ by such hardware resources as memory, number of SMs, number of registers, etc.
- We shall list here only several recent data center quality cards used for heavy computations.

Model	Micro Architecture	Compute Capability	SMs	cores	Memory (GB)	Connection to CPU	Bandwidth to CPU (GB/s)	Bandwidth to memory (GB/s)
K80	Kepler	3.7	2x13	2x2496	2x12	PCIe	12	2x240
P100	Pascal	6.0	56	3584	12, 16	PCIe, NVlink	32, 160	549, 732
V100	Volta	7.0	80	5120	16, 32	PCIe, NVlink	32, 300	900

- Modern consumer level GPU cards found in laptops and desktop typically can also be used for GPU computing.

GPU hardware: Lab 1: querying hardware

- To find out how many GPU cards are there in a node, what kind, driver version, what processes are running on which GPU card, use nvidia-smi:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.72       Driver Version: 410.72       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 00000000:08:00.0 Off |                    0 |
| N/A   27C    P8    27W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           On   | 00000000:09:00.0 Off |                    0 |
| N/A   32C    P8    28W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K80           On   | 00000000:88:00.0 Off |                    0 |
| N/A   28C    P8    25W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K80           On   | 00000000:89:00.0 Off |                    0 |
| N/A   33C    P8    29W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

GPU hardware: Lab 1: querying hardware

- To submit a job in batch to gpu2 node that would run nvidia-smi:

```
cd cuda/labs/1
sbatch nvidia-smi.batch
```

- If we have enough GPUs, we can log into the node and use it interactively:

```
sinteractive -p gpu2 --gres=gpu:1 --reservation=CUDA
```

- Set up the environment `source cuda/labs/env.sh`
- Above you ask for one gpu card in one gpu2 node. `--reservation` only exists for the duration of this workshop. Otherwise, do not use it.
- One can run nvidia-smi continously, every few seconds, as top, using -l flag.
- This can be very useful to diagnose problems:
  - is the program using too much GPU memory?
  - is the load on GPU too low?
  - is there hardware problem with GPU?

GPU hardware: Lab 1: querying hardware

- Note: one can restrict a program to see only some of the available GPUs by setting `CUDA_VISIBLE_DEVICES` to the list of visible GPU cards, separated by comma
- When you submit a job to SLURM scheduler, it sets this variable for you, so that each user on a node sees only as many GPU cards as was requested
- Otherwise, the program sees all GPU cards and can grab them whether their are needed or not, possibly blocking other users

GPU hardware: Lab 1: querying hardware

- One can query the detailed parameters of the device with deviceQuery program that comes with CUDA.

```
/software/cuda-10.0-el7-x86_64/samples/1_Utilities/deviceQuery/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          10.0 / 10.0
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
...
```

GPU hardware: Lab 1: querying hardware

```
...
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                         2147483647 bytes
  Texture alignment:                            512 bytes
  Concurrent copy and kernel execution:         Yes with 2 copy engine(s)
  Run time limit on kernels:                    No
  Integrated GPU sharing Host Memory:           No
  Support host page-locked memory mapping:      Yes
  Alignment requirement for Surfaces:           Yes
  Device has ECC support:                       Enabled
  Device supports Unified Addressing (UVA):     Yes
  Device supports Compute Preemption:           No
  Supports Cooperative Kernel Launch:           No
  Supports MultiDevice Co-op Kernel Launch:     No
  Device PCI Domain ID / Bus ID / location ID:   0 / 9 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.0, CUDA Runtime Version = 10.0, NumDevs = 1
Result = PASS
```

CUDA programming

- The native way to program GPU is to use CUDA - Compute Unified Device Architecture - from NVIDIA.
- The latest version of CUDA is 10.0
- NVIDIA first was producing GPU cards as pure graphics accelerators before it was realized that GPU card can be used to speed up numerical computations, especially those that involve a lot of linear algebra
- CUDA was introduced to support using GPU cards for numerical computations in 2007.
- CUDA provides an extension of C/C++ syntax and libraries that allow to write functions, called kernels, that get executed in parallel by multiple threads on GPU.
- CUDA allows to allocated/deallocate memory on GPU and copy data between CPU's RAM and GPU's global memory.
- One can also program CUDA in Fortran, Python, Java.
- Let us consider the simplest example of adding two vectors.

CUDA programming: Lab 2: thread, block, grid, kernel

- In this lab we are adding two vectors, `A` and `B`, consisting of 50000 floats. The result is put into `C`.
- Obviously different elements of the vectors can be added independently from each other in parallel.
- First, in the main program we allocate memory for `A`, `B`, `C` on the host as usual, with `malloc`, and populate `A` and `B` with random numbers.
- Next we need to allocate memory on GPU card with cudaMalloc:

```
float *d_A = NULL, *d_B = NULL,  *d_C = NULL;
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);
```

CUDA programming: Lab 2: thread, block, grid, kernel

- Copy data from the host to the device:

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

- Next we need to decide how to divide our work into blocks of threads.
  - Each block runs entirely on one of SMs.
  - Different blocks might run in parallel or sequentially and get launched in any order on the same or different SMs.
  - Each block can have no more than 1024 threads.
  - Since the warp size is 32, block size should be divisible by 32.
- We shall consider tradeoffs for different choices later but for now let us just use 256 threads per block and compute how many blocks we need in order to add 50000 elements.

```
int numElements = 50000;
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
```

CUDA programming: Lab 2: thread, block, grid, kernel

- The actual work is done in a kernel - function executed by each thread in the job:

```
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
        C[i] = A[i] + B[i];
}
```

- Notice `__global__` in front of the GPU function
- From inside the kernel one can use the following variables:
  - `blockDim.x` - block size;
  - `blockIdx.x` - block id in the grid;
  - `threadIdx.x` - thread id in the block;
- For each thread, the kernel computes the index `i` of the vector variables that it handles.
- The corresponding elements of `A` and `B` are added and stored in `C`.
- There might be more threads than the number of elements to add, therefore `if` is needed.

CUDA programming: Lab 2: thread, block, grid, kernel

- To call the kernel from the host program, one must specify the number of blocks and threads per block:

```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

- The same kernel can be called with different grid and block dimensions.
- After the kernel completes, we copy the result back to the host and free device memory:

```
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

- Finally we free memory for host variables.

CUDA programming: Lab 2: thread, block, grid, kernel

- Until compute capability 5.0, the device code should be in the same file as the host code that calls the kernel.
- Since we are using K80 with compute cability 3.7, this is the case for us.
- The files that contain device code typically have cu extension.
- To compile the code, we need to use nvcc compiler that would compile device code and call system compiler to build the host code and then link them together:

  ```
  nvcc -o vectorAdd vectorAdd.cu
  ```

- By default, nvcc would compile the code for the current hardware. One can specify compute capability and exact SM architecture, one can even specify several. JIT compilation is supported as well to be able to run on new hardware for which the program was not originally compiled.

CUDA programming: Lab 2: thread, block, grid, kernel

- Now when we understood the overall structure of `vectorAdd.cu`, let us look at the original file from `samples/0_Simple/vectorAdd`
- The main thing that I skipped for simplicity is error handling.

```
cudaError_t err = cudaSuccess;
...
err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
...
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
...
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
err = cudaGetLastError();
...
err = cudaFree(d_A);
```

- The error checking should be done after each CUDA call. Otherwise, you might not notice that the program is working incorrectly and producing wrong results.

CUDA programming: shared memory, barrier

- First let us implement straightforward matrix multiplication (matrixMul0.cu) and then reimplement it using shared memory (matrixMul1.cu) and compare performance between two GPU and CPU sequential implementations.
- We define Matrix structure as follows:

```
typedef struct
{
  int width;
  int height;
  float * elements;
} Matrix;
```

- We allocate memory for matrices A, B, C on the host and device, generate random elements for A and B on the host and copy them to the device.

CUDA programming: shared memory, barrier

- To measure the timing for different kernels, we use cuda events:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

- In general, cuda events can be used to synchronize various threads.

CUDA programming: shared memory, barrier

- Notice that grids and blocks in CUDA can be 1-, 2-, 3-dimensional.
- To multiply 2D matrices, it is natural to use 2D grids and blocks.
- When calling a kernel, instead of 1D integers for grid and block sizes, one can use 3D tuples of type dim3, in which by default, if not specified, all three components are set to 1:

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width/dimBlock.x, A.height/dimBlock.y);
...
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

CUDA programming: shared memory, barrier

- In the first GPU implementation of matrix multiplication, we let each thread to compute a separate entry of C matrix:

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
              B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

- Notice that blockIdx, blockDim, threadIdx are of type dim3

CUDA programming: shared memory, barrier

- We compare the performance of CUDA program with the performance of a single threaded straightforward matrix multiplication on CPU:

```
void sequential_cpu(Matrix A, Matrix B, Matrix C)
{
  for(int i = 0; i < C.height; ++i)
    {
      for(int j = 0; j < C.width; ++j)
        {
          C.elements[i*C.width + j] = 0;
          for(int ac = 0; ac < A.width; ++ac)
            {
              for(int br = 0; br < B.height; ++br)
                {
                  C.elements[i*C.width + j] +=
                    A.elements[i*A.width + ac] *
                    B.elements[j + br*B.width];
                }
            }
        }
    }
}
```

- To multiply 160x240 and 240x320 matrices took 0.4 ms with GPU implementation on K80 and 14320 ms on a single Broadwell CPU core of gpu2 partition. Naive GPU implementation on a 4 year old K80 is 35800 times faster!

CUDA programming: shared memory, barrier

- Notice: the first time the kernel runs on GPU in your program, it is slower than in subsequent times (0.6 ms vs 0.4 ms for this problem) due to some warming up overhead.
- To take this into account, the kernel was called 5 times and the results are averaged over the last 4 measurements.
- In this GPU implementation each element of A is read B.width times and each element of B is read A.height times from global memory.
- Since global memory is slow, in ideal we want to read things from there only once and cache it in shared memory which is much faster.
- In ideal it would also be best to read data from global memory in a coalesced way to take advantage of automatic caching in L1, L2 caches:
  - When you ask to read data at address A of size, say 4 bytes, instead hardware would read a cacheline of something like 32 of such elements and store them in fast but small cache.
  - If the next address you read is available from cache, it is taken from there and global memory is not read

CUDA programming: shared memory, barrier

- In the second implementation of matrix multiplication we shall divide each matrix into submatrices of 16x16 size
- Each thread block caches submatrix of A and submatrix of B in shared memory before performing a straightforward matrix multiplication of such submatrices.
- As a result, each element of A is accessed `B.width/16` times and each element of B is accessed `A.height/16` times reducing the load on the global memory by a factor of 256.

CUDA programming: shared memory, barrier

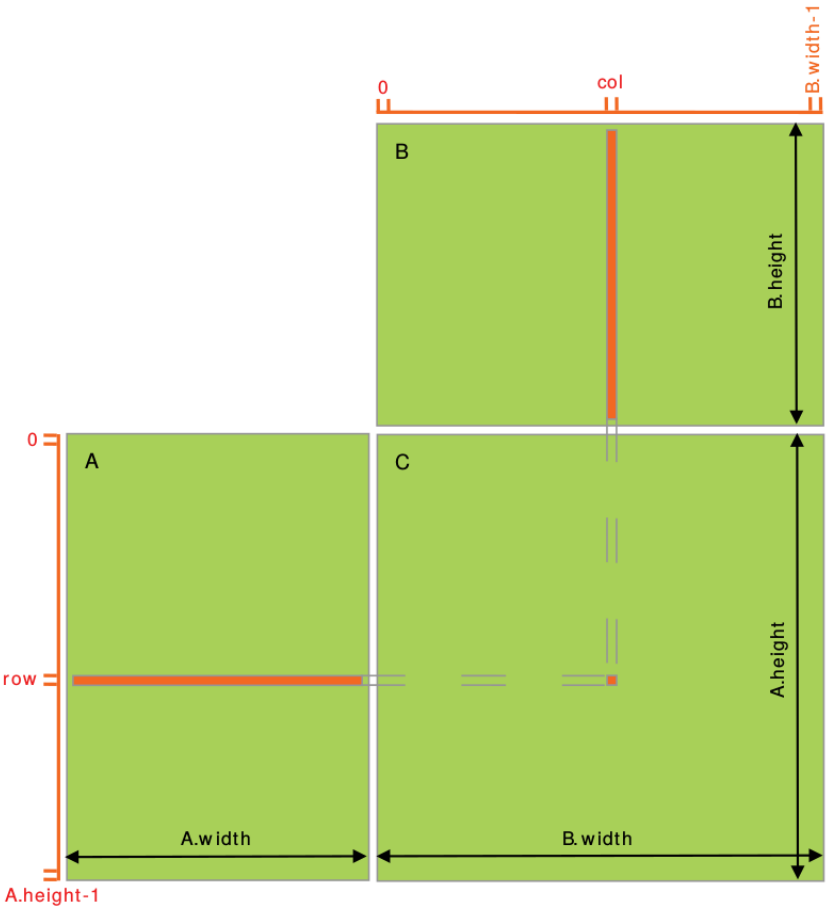


Figure 9 Matrix Multiplication without Shared Memory

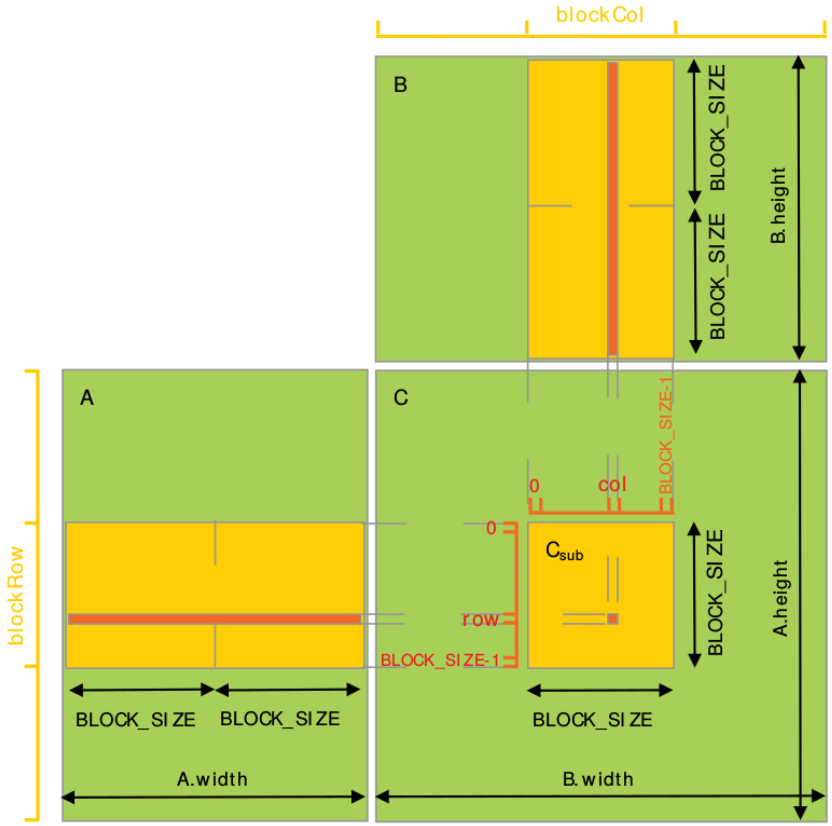


Figure 10 Matrix Multiplication with Shared Memory

CUDA programming: shared memory, barrier

- To deal with submatrices, we need to modify the definition of the Matrix structure:

```
typedef struct
{
  int width;
  int height;
  int stride;
  float * elements;
} Matrix;
```

- Here width refers to the width of the submatrix while stride refers to the width of the big matrix.
- Same structure can be used for the big matrix:
  - in that case width = stride.
- The big matrix is subdivided into submatrices because you can store only so much in shared memory. For K80, it is up to 48kb.

CUDA programming: shared memory, barrier

- We define several convenience device functions:
  - to extract a submatrix in place:

```
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
  Matrix Asub;
  Asub.width = BLOCK_SIZE;
  Asub.height = BLOCK_SIZE;
  Asub.stride = A.stride;
  Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row +
                              BLOCK_SIZE * col];
  return Asub;
}
```

  - to get/set matrix element:

```
__device__ float GetElement(const Matrix A, int row, int col)
{
  return A.elements[row * A.stride + col];
}

__device__ void SetElement(Matrix A, int row, int col, float value)
{
  A.elements[row * A.stride + col] = value;
}
```

- Notice the above device functions are called from the GPU kernel that does the main job

CUDA programming: shared memory, barrier

- Until compute capability 3.5, CUDA 5.0, only host could call a function that runs on GPU
- Now GPU functions can call GPU functions
- This feature is called dynamic parallelism and it allows implementing, for example, recursive algorithms on GPU
- In the main kernel, each block computes a submatrix Csub by looping over submatrices Asub and Bsub
- Each submatrix is copied from the global memory into the shared memory
- All threads in a block wait on a barrier `__syncthreads()` to make sure that all the necessary entries are loaded into shared memory before using them.
- Another barrier is inserted at the end of the loop iteration to make sure that all the threads in a block finished using shared memory before overwriting it with new submatrices

CUDA programming: shared memory, barrier

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;

  int row = threadIdx.y;
  int col = threadIdx.x;

  // Loop over all the sub-matrices of A and B that are required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for(int m = 0; m < (A.width/BLOCK_SIZE); ++m)
    {
      // Get sub-matrix Asub of A
      Matrix Asub = GetSubMatrix(A, blockRow, m);

      // Get sub-matrix Bsub of B
      Matrix Bsub = GetSubMatrix(B, m, blockCol);
```

CUDA programming: shared memory, barrier

```
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation

    __syncthreads();
    // Multiply Asub and Bsub together
    for(int e = 0; e < BLOCK_SIZE; ++e)
      {
        Cvalue += As[row][e] * Bs[e][col];
      }

    // Synchronize to make sure that the preceding
    // computation is done before load two new sub-matrices of A and B in the next
    // iteration
    __syncthreads();
  }
  // Write Csub to device memory
  // Each thread writes one element
  SetElement(Csub, row, col, Cvalue);
}
```

CUDA programming: shared memory, barrier

- By using shared memory we further speed up the program by another factor of 2.5: it now takes 0.16 ms for the same matrices.
- Trying to multiply big matrices on a single CPU core takes forever.
- Of course, on CPU one can also parallelize the program and take advantage of vectorization
- There are 28 CPU cores on midway2 nodes
- Vectorization might also give another factor of 16 or so
- Still: to parallelize on CPU also requires some efforts (probably using OpenMP) and the expected performance gain would probably be 10-30 times worse than on K80.
- Of course, this was just an excercise. To do linear algebra in your application, you should use libraries:
  - on CPU: MKL, OpenBlas, Atlas, Plasma, Eigen, SuperLU...
  - on GPU: cuBLAS, cuSPARSE, Magma...

CUDA programming: Lab 4: intrinsics, streams, tools

- In this lab we shall learn about CUDA math library, streams, nsight tools
- Let us create on CPU a big array, initialize it with random numbers, copy it to GPU, apply some math functions to it and copy it back to the host.
- Here is the kernel

```
__global__ void map(float *d_a, int n, int m)
{
  int id = blockDim.x * blockIdx.x  + threadIdx.x;

  if(id < n)
    for(int i = 0; i < m; ++i)
      {
        d_a[id] = sinf(d_a[id]) + cosf(d_a[id]);
        d_a[id] = expf(d_a[id]);
        d_a[id] = rsqrtf(d_a[id]) - d_a[id];
      }
}
```

- In CUDA there are standard math functions - can be executed both on the host and device - and intrinsic functions - can only run on device, are faster but less precise.
- The code above uses standard functions.

CUDA programming: Lab 4: intrinsics, streams, tools

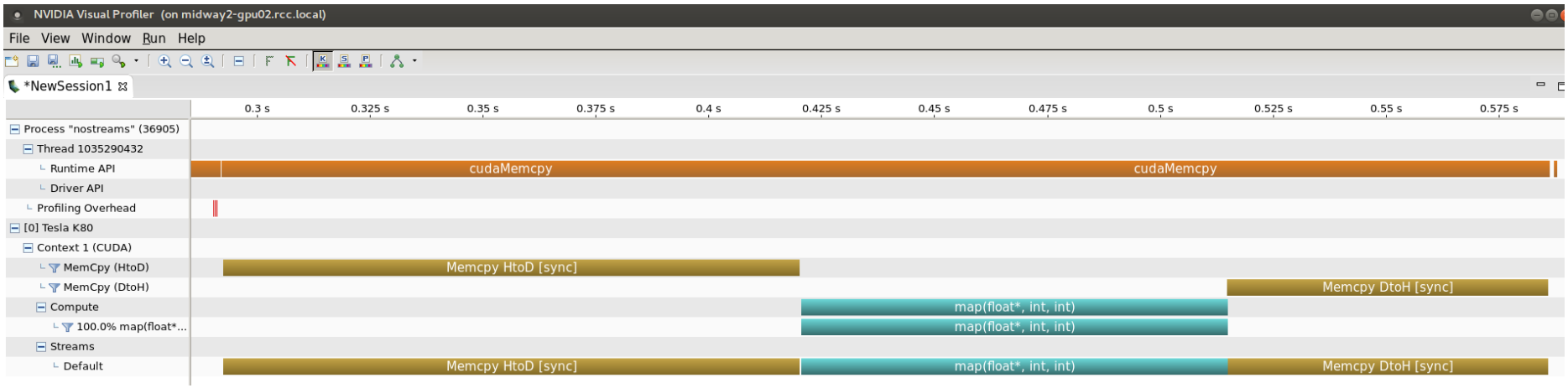
- Examples of standard math functions: sqrtf, sqrt, rsqrtf, rsqrt, sinf, sin, etc. Those are single and double precision functions.
- Examples of the corresponding intrisic functions: `__fsqrt_rn`, `__frsqrt_rn`, `__sinf`, etc. The precision is less than float.
- If one can afford less accurate functions, one can either replace the functions in the code or use compile option `--use_fast_math` to replace all the standard functions by the corresponding intrinsic functions.
- Let us measure the performance of the same kernel with standard and intrinsic math functions: 131 vs 27 ms! This kernel runs almost 5 times faster with intrinsic functions than with standard ones.
- Let us look how the results are different:

```
a[0] = 1.74003, a[1] = 1.45576, a[2] = -3.50318, a[3] = -3.0294, a[4] = -3.22013 ...
vs
a[0] = 1.74017, a[1] = 1.45575, a[2] = -3.50318, a[3] = -3.02938, a[4] = -3.22022 ...
```

- Not a big difference in numerical results.
- Consult the tables in the "CUDA C Programming Guide" for precision guarantees and the complete list of the available functions.

CUDA programming: Lab 4: intrinsics, streams, tools

- Let us run the program through nvvp profiler
- We see that first the program spends significant time transfering data from host to device, then comparable amount of time is spent on the computations and transfering the results back to the host.



- Meanwhile, the computations are embarrasingly parallel and independent for each element of the array.
- Would not it be better first to transfer a part of data, start computations on GPU on the first part of the data, transfer in parallel with the computations second part, etc. to hide the latency

CUDA programming: Lab 4: intrinsics, streams, tools

- Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the concurrentKernels device property, which is equal to 1 for devices that support it.
- The maximum number of kernel launches that a device can execute concurrently depends on its compute capability.
- Some devices can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution. Applications may query this capability by checking the asyncEngineCount device property, which is greater than zero for devices that support it. If host memory is involved in the copy, it must be page-locked
- Some devices of compute capability 2.x and higher can overlap copies to and from the device. Applications may query this capability by checking the asyncEngineCount device property, which is equal to 2 for devices that support it. In order to be overlapped, any host memory involved in the transfers must be page-locked.

CUDA programming: Lab 4: intrinsics, streams, tools

- For K80, according to deviceQuery from Lab 1:
  `Concurrent copy and kernel execution: Yes with 2 copy engine(s)`
- CUDA construction that can help to interleave computations and data transfer is called 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.
- A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host to/from device memory copies.
- To use streams for data transfer between host and device, one needs to allocated memory on the host not with malloc but as follows

```
float *h_a = NULL;
 cudaMallocHost(&h_a, size);
```

to ensure that the memory allocated on the host is page-locked (cannot be swapped)

CUDA programming: Lab 4: intrinsics, streams, tools

- Next we need to create streams:

```
cudaStream_t stream[nstreams];
for(int i = 0; i < nstreams; ++i)
  cudaStreamCreate(&stream[i]);
```

- Send data between host and device in batches and run a kernel on each batch to interleave computations and data transfer:

```
for(int i = 0; i < nstreams; ++i)
{
  cudaMemcpyAsync(d_a + i*batch, h_a + i*batch, batch_size, cudaMemcpyHostToDevice, stream[i]);
  map<<<grid_size, block_size, 0, stream[i]>>>(d_a + i*batch, batch, iterations)
  cudaMemcpyAsync(h_a + i*batch, d_a + i*batch, batch_size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- Notice: we provide stream[i] argument to data transfer operations and to a kernel call.
- Also note: instead of cudaMemcpy we use asynchronous version cudaMemcpyAsync that, contrary to cudaMemcpy immediately returns without waiting for data transfer to finish.

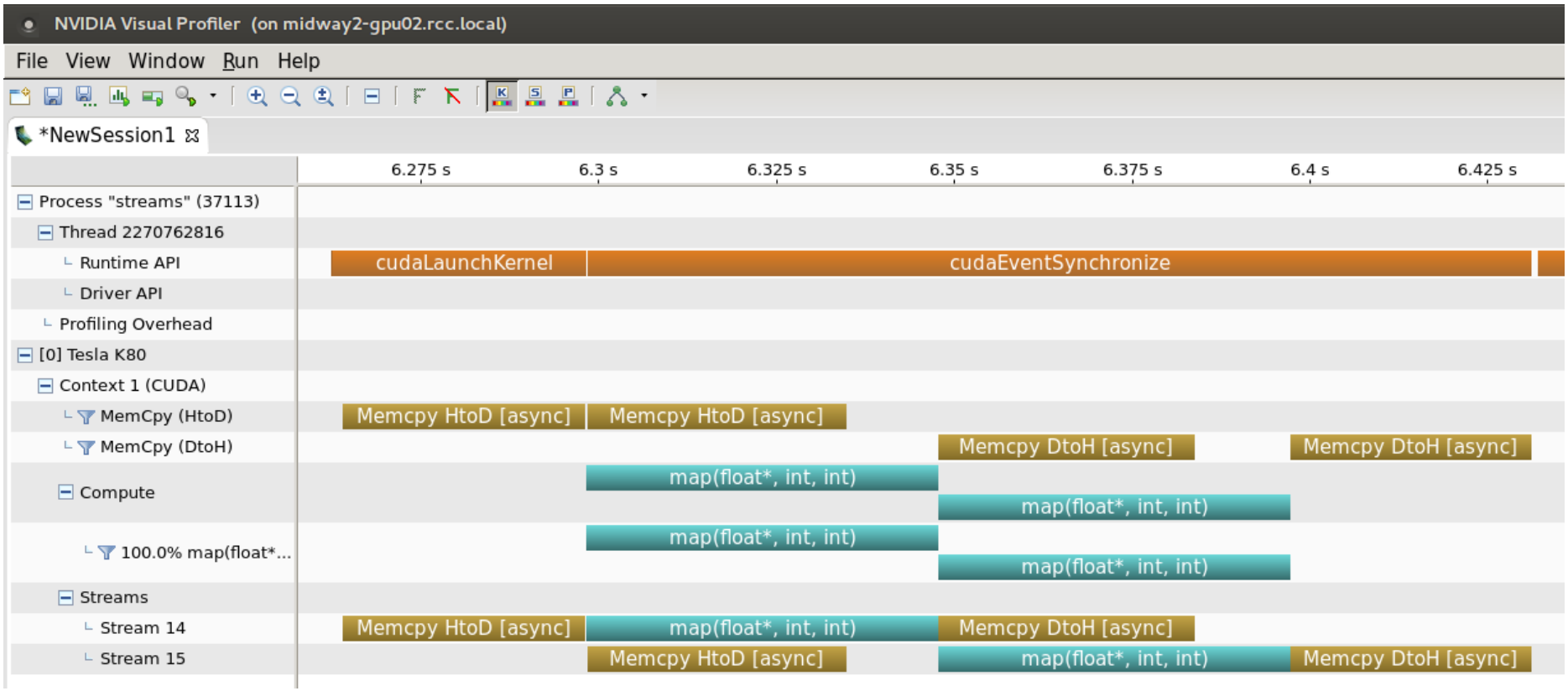
- To free resources, we need to destroy streams and release page-locked memory (not with free but with cudaFreeHost):

```
for(int i = 0; i < nstreams; ++i)
  cudaStreamDestroy(streams[i])
cudaFreeHost(h_a);
```

- In case the device is still doing work in the stream when cudaStreamDestroy() is called, the function will return immediately and the resources associated with the stream will be released automatically once the device has completed all work in the stream.
- Kernel launches and host to/from device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream. They are therefore executed in order.

CUDA programming: Lab 4: intrinsics, streams, tools

- Let us now run nvvp and see if we succeeded in interleaving data transfer and computations:



- As you can see, kernel and data transfer are indeed interleaved but only one kernel runs at a time. This is probably the limitation of K80.
- Let us see how it affected timing: 223 ms vs 138 ms.
- Newer GPU cards might benefit more from streams.
- There are various ways to explicitly synchronize streams with each other:
  - cudaDeviceSynchronize() waits until all preceding commands in all streams of all host threads have completed.
  - cudaStreamSynchronize() takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.
  - cudaStreamWaitEvent() takes a stream and an event as parameters and makes all the commands added to the given stream after the call to cudaStreamWaitEvent() delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to cudaStreamWaitEvent() wait on the event.

- More ways to explicitly synchronize streams with each other.
  - cudaStreamQuery() provides applications with a way to know if all preceding commands in a stream have completed
- One can insert a callback at any point into a stream via cudaStreamAddCallback(). A callback is a function that is executed on the host once all commands issued to the stream before the callback have completed. Callbacks in stream 0 are executed once all preceding tasks and commands issued in all streams before the callback have completed.
- The relative priorities of streams can be specified at creation using cudaStreamCreateWithPriority().

CUDA programming: Lab 4: intrinsics, streams, tools

- Instead of specifying what tasks run in what streams and what are dependencies between the tasks, one can use graphs
- Graphs present a new model for work submission in CUDA. A graph is a series of operations, such as kernel launches, connected by dependencies, which is defined separately from its execution.
- This allows a graph to be defined once and then launched repeatedly.
- Separating out the definition of a graph from its execution enables a number of optimizations: first, CPU launch costs are reduced compared to streams, because much of the setup is done in advance; second, presenting the whole workflow to CUDA enables optimizations which might not be possible with the piecewise work submission mechanism of streams.
- Graphs can be created with Graph API or by capturing an old fashioned streams based program.

- As was mentioned before, to interleave data transfer and computations, the corresponding part of the host memory should be pinned or page-locked.
- That means that it cannot be swapped to disk by OS.
- Other uses of pinned memory:
  - On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to or from device memory
  - On some systems bandwidth between host memory and device memory is higher if host memory is allocated as page-locked
- However, page-locked host memory is a scarce resource, so allocations in page-locked memory will start failing long before allocations in pageable memory. In addition, by reducing the amount of physical memory available to the operating system for paging, consuming too much page-locked memory reduces overall system performance.

CUDA programming: Lab 4: intrinsics, streams, tools

- We have already seen nvvp profiler.
- CUDA comes with a suite of tools for debugging and profiling:
  - There IDE with debugger, profiler, editor called nsight
  - For nsight to show the source code, you need to compile it with -g -G option to nvcc
  - Instead of using nsight GUI, you can run cli-based cuda-gdb which has similar interface as gdb. It allows to look into each thread separately.
  - cuda-memcheck can be used to diagnose memory problems
  - If you cannot run profiler on the node interactively, you can run it as a batch job and then look at the results with GUI.
  - You can also run nsight remotely
- nsight is supposed to be deprecated soon and replaced by nsight compute and nsight systems. So far I can run those on my laptop (Ubuntu 16.04) but not on midway (Scientific Linux 7): they seem to require newer version of glibc.

CUDA programming: Lab 4: intrinsics, streams, tools

- We have already seen how to use events to time program execution.
- CUDA lets the application asynchronously record events at any point in the program and query when these events are completed.
- An event has completed when all tasks - or optionally, all commands in a given stream - preceding the event have completed.
- Events in stream zero are completed after all preceding tasks and commands in all streams are completed.

```
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start);
map<<<grid_size, block_size>>>(d_a, N, iterations);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```

CUDA programming: Lab 5: occupancy, libraries

- In this lab we investigate how to select block size for optimal performance.
- First we randomly generate couple arrays directly on GPU using curand library:

```
#include <iostream>
#include <curand.h>
using namespace std;

struct random_d_array
{
  float *data;
  int n;

  random_d_array(int n) :n{n}
  {
    cudaMalloc((void**)&data, n*sizeof(float));
    curandGenerator_t gen;
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
    curandGenerateUniform(gen, data, n);
  }

  ~random_d_array()
  {
    cudaFree(&data);
  }
};
```

- At compilation the code needs to be linked with the library:

```
nvcc -lcurand -o occupancy occupancy.cu
```

- Also notice `#include <curand.h>` in the code

- The program loops until you press 'q' and at each iteration asks you for the block size, computes the corresponding grid size, occupancy, launches the kernel that multiplies two random vectors and measures average run time:

block size	grid size	run time (ms)	occupancy (%)
32	32768	0.21	25
64	16384	0.13	50
96	10923	0.12	75
128	8192	0.11	100
256	4096	0.11	100
512	2048	0.11	100
1024	1024	0.12	100

- As we can see, the smaller the occupancy, the worse is the running time.

CUDA programming: Lab 5: occupancy, libraries

- Occupancy is the ratio of the number of active warps per SM to the maximum number of possible active warps.
- We obviously prefer to keep the whole GPU happily busy all the time.
- Here is how occupancy is computed in this program:

```
cudaGetDevice(&device);
cudaGetDeviceProperties(&prop, device);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, MyKernel, blockSize, 0);
activeWarps = numBlocks * blockSize/prop.warpSize;
maxWarps = prop.maxThreadsPerMultiProcessor/prop.warpSize;
cout << "Occupancy: " << (double)activeWarps/maxWarps * 100 << "%" << endl;
```

- What limits the occupancy?
- The main factor that determine occupancy is register availability.
- Register storage enables threads to keep local variables nearby for low-latency access. However, the set of registers (known as the register file) is a limited commodity that all threads resident on a multiprocessor must share.

CUDA programming: Lab 5: occupancy, libraries

- Registers are allocated to an entire block all at once.
- So, if each thread block uses many registers, the number of thread blocks that can be resident on a multiprocessor is reduced, thereby lowering the occupancy of the multiprocessor.
- The number of registers available, the maximum number of simultaneous threads resident on each multiprocessor, and the register allocation granularity vary over different compute capabilities.
- To compute occupancy for the given kernel and hardware, one can either use CUDA functions, like in this example, or use an Excel spreadsheet called "Occupancy Calculator" that comes with CUDA:
  `/software/cuda-10.0-el7-x86_64/tools/CUDA_Occupancy_Calculator.xls`
- The `--ptxas options=v` option of `nvcc` details the number of registers used per thread for each kernel.

CUDA programming: Lab 5: occupancy, libraries

- Higher occupancy does not always equate to higher performance - there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.
- Here are some recommendations how to choose the block size:
  - Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
  - A minimum of 64 threads per block should be used
  - Between 128 and 256 threads per block is a better choice and a good initial range for experimentation with different block sizes.
  - Experiment
- In the second example we use

```
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize,
                                   (void*)MyKernel, 0, arrayCount);
```

  to automatically select optimal block size for the given kernel and array size, which is found to be 1024. Running time is 0.018 ms. For non-optimal block size 32, running time is 0.025 ms.

CUDA programming: Lab 6: coalescing

- One of the most important performance consideration in programming for CUDA-capable GPU architectures, according to the documentation, is the coalescing of global 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.
- The access requirements for coalescing depend on the compute capability of the device
- For devices of compute capability 2.x, for example, the concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads of the warp.
- By default, all accesses are cached through L1, which as 128-byte lines. For scattered access patterns, to reduce overfetch, it can sometimes be useful to cache only in L2, which caches shorter 32-byte segments.

CUDA programming: Lab 6: coalescing

- For devices of compute capability 3.x, accesses to global memory are cached only in L2; L1 is reserved for local memory accesses.
- In the lab, we are trying to copy one array into another using coalesced, misaligned and strided access:

```
__global__ void copy1(float *a, float *b, int n)
{
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  if(id < n)
    a[id] = b[id];
}

__global__ void copy2(float *a, float *b, int n, int offset)
{
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  if(id < n)
    a[id] = b[(id + offset) % n];
}

__global__ void copy3(float *a, float *b, int n, int stride)
{
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  if(id < n)
    a[id] = b[(id * stride) % n];
}
```

CUDA programming: Lab 6: coalescing

- Probably because K80 has compute capability 3.7, I do not see that much difference in coalesced, misaligned or strided access to global memory in the lab:
  - Running time for coalleced access is 130 ms
  - Running time for misaligned and strided access is 155 ms

CUDA programming: Lab 7: atomics

- Suppose we want each thread to increment a counter:

```
__device__ int counter = 0;
__global__ void increment()
{
  counter++;
}
```

  where counter above is a global device variable initialize to 0.
- We naively expect that at the end the counter will be equal to the number of threads.
- We block size is 1024 and grid size is 1024. So the total number of threads is 1048576.
- However, when we run the program, each time we get a different number: 54, 56, 53, ...
- The problem is race condition.

CUDA programming: Lab 7: atomics

- Incrementing a counter is not an atomic operation but consists of several atomic operations:
  - first the current value of the counter needs to be read from global memory into the register of the thread
  - then the thread needs to increment it in register
  - finally it writes it back into global memory
- For a sequential program this is not a problem.
- However, for a multithreaded program it is since the order of those operations is not defined. For example, it can be as follows:
  - Thread 1 reads counter $= 3$ into register;
  - Thread 2 reads counter $= 3$ into register;
  - Thread 1 increments the register to 4;
  - Thread 1 writes 4 into the global variable counter which is now 4;
  - Thread 2 increments its register to 4;
  - Thread 2 writes its register to the global variable counter which is now 4.
  - However, we obviously want it to be 5.

CUDA programming: Lab 7: atomics

- To handle such situation, CUDA provides atomic operations. For example, we can rewrite the previous kernel:

```
__global__ void increment()
{
  atomicAdd(&counter, 1);
}
```

- Now the counter behaves as expected since once one thread starts changing the counter, others would have to wait until the operation is finished.
- There are the following atomic functions available: atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS, atomicAnd, atomicOr, atomicXor.
- Unfortunately most of the atomic operations are provided for integers only.
- One can use atomicCAS to create atomic operations for other data types.

CUDA programming: Lab 7: atomics

- atomicCAS stands for atomic Compare And Swap

```
int atomicCAS(int* address, int compare, int val);

unsigned int atomicCAS(unsigned int* address, unsigned int compare, unsigned int val);

unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

- It reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes `(old == compare ? val : old)` , and stores the result back to memory at the same address.
- These three operations are performed in one atomic transaction. The function returns old.

CUDA programming: Lab 7: atomics

- Here is how myAtomicAdd for doubles can be implemented using atomicCAS

```
__device__ double myAtomicAdd(double * address, double val)
{
  unsigned long long int * address_as_ull =
    (unsigned long long int*)address;
  unsigned long long int old = *address_as_ull, assumed;
  do {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed,
                    __double_as_longlong(val + __longlong_as_double(assumed)));
  } while (assumed != old);

  return __longlong_as_double(old);
}
```

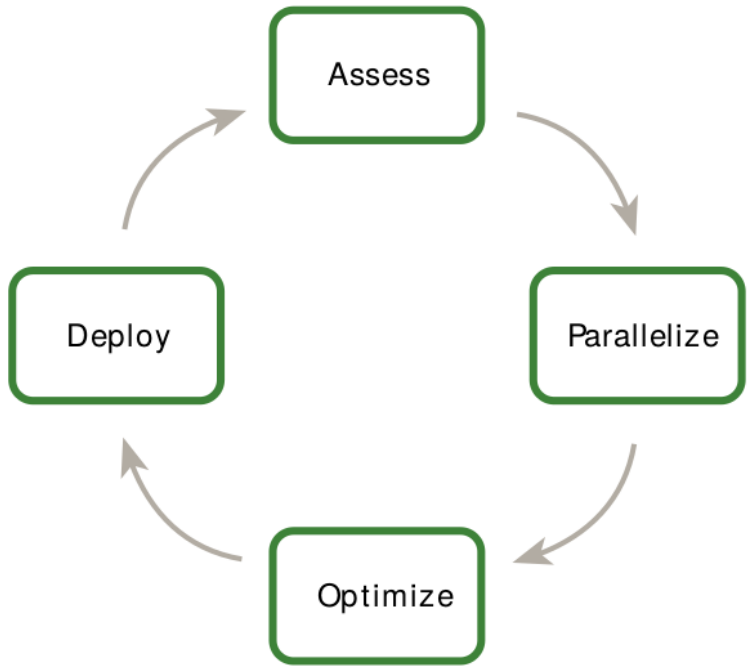
- Notice: atomics introduce some communication overhead since a variable is locked for other threads until the thread that aquired the lock finishes with it.

APOD

- "CUDA Best Practices" suggests APOD design cycle approach to cudarize your program.
- APOD - Assess, Parallelize, Optimize, Deploy
- Assess - find the parts of the application where most of the time is spent, possibly using profiler. Use Amdahl's and Gustafson's laws to understand expected performance benefits from parallelizing that part of the code.
- Parallelize - parallelize the identified bottleneck. Perhaps use the existing library such as cuBLAS, cuFFT, cuSPARSE, Thrust, etc. Or write it from scratch in CUDA.
- Optimize - optimize the parallel version using your understanding of the underlying hardware
- Deploy - put it in production ASAP. This would allow to identify potential problems and benefit from the achieved performance gain faster

APOD

- APOD is a cyclical process: initial speedups can be achieved, tested, and deployed with only minimal initial investment of time, at which point the cycle can begin again by identifying further optimization opportunities, seeing additional speedups, and then deploying the even faster versions of the application into production.



Conclusion

- In this tutorials we covered most of the CUDA constructions that you need to start accelerating your program.
- However, before diving into such a low level programming and reinventing a wheel, consider using existing accelerated libraries:
  - CUDA math library that implements all the standard math functions
  - cuBLAS - Basic Linear Algebra Subprograms
  - NVBLAS - multigpu version of BLAS
  - cuFFT - Fast Fourier Transform
  - nvGRAPH - graph processing routines
  - cuRAND - random numbers generators
  - cuSPARSE - linear algebra for sparse matrices
  - NPP - image, video processing
  - cuSOLVER - higher level linear algebra routines on top of cuBlas and cuSparse
  - Magma - Matrix Algebra on GPU and Multicore Architectures
  - AmgX - Multi-Grid Accelerated Linear Solvers for Industrial Applications
  - ArrayFire - signal and image processing

Conclusion

- Continued:
  - GUNROCK - graph processing
  - GPP - computational geometry
  - CUB - provides a collection of building blocks, includes sort, scan, reduction
  - cuDNN - library for Deep Neural Networks
- There are also higher level frameworks that make it easier to do GPU programming:
  - OpenACC - high level pragma based framework for GPU acceleration, similar to OpenMP
  - Thrust - C++ framework similar to STL
  - Kokkos - another high-level C++ template library

References

CUDA C Programming Guide:
https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

CUDA best practices:
https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

Udacity "Introduction to Parallel Programming":
https://classroom.udacity.com/courses/cs344

Oxford course in CUDA programming:
https://people.maths.ox.ac.uk/gilesm/cuda/

Book: Programming massively parallel devices,
by David B. Kirk, Wen-mei W. Hwu

Thrust:
https://docs.nvidia.com/cuda/thrust/index.html

Book: Parallel Programming with OpenACC,
edited by Rob Farber

Appendix: Tutorials on demand

- Currently the following tutorials are available at any time whenever there is an audience:
  1. Introduction to Linux and RCC
  2. Introduction to Python
  3. Distributed MPI programming in C/C++/Python
  4. Multithreaded programming in OpenMP
  5. GPU programming with CUDA
  6. Introduction to Hadoop
  7. Big Data and Machine Learning with Spark and BigDL
  8. Deep Learning with Keras and TensorFlow
  9. Deep Learning with Theano
  10. Deep Learning with PyTorch
  11. Sequence Models with Recurrent Neural Networks
  12. Reinforcement Learning
  13. Singularity Container
- More are coming: Autoencoders, Generative Adversarial Networks, Object detection with YOLO, C++, ...
- Requests for other topics?