Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

March 30, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last lecture

- OpenMP: III
- Libraries

Today

- GPUs and CUDA programming

Announcements and upcoming

- Homework 3 due April 6
- You should have a rough idea about your final project by now - email/talk to me about it and finalize a project description by April 6.

Final project

Summarize your current plan for the final project in a PDF document and send to me and Melody via email.

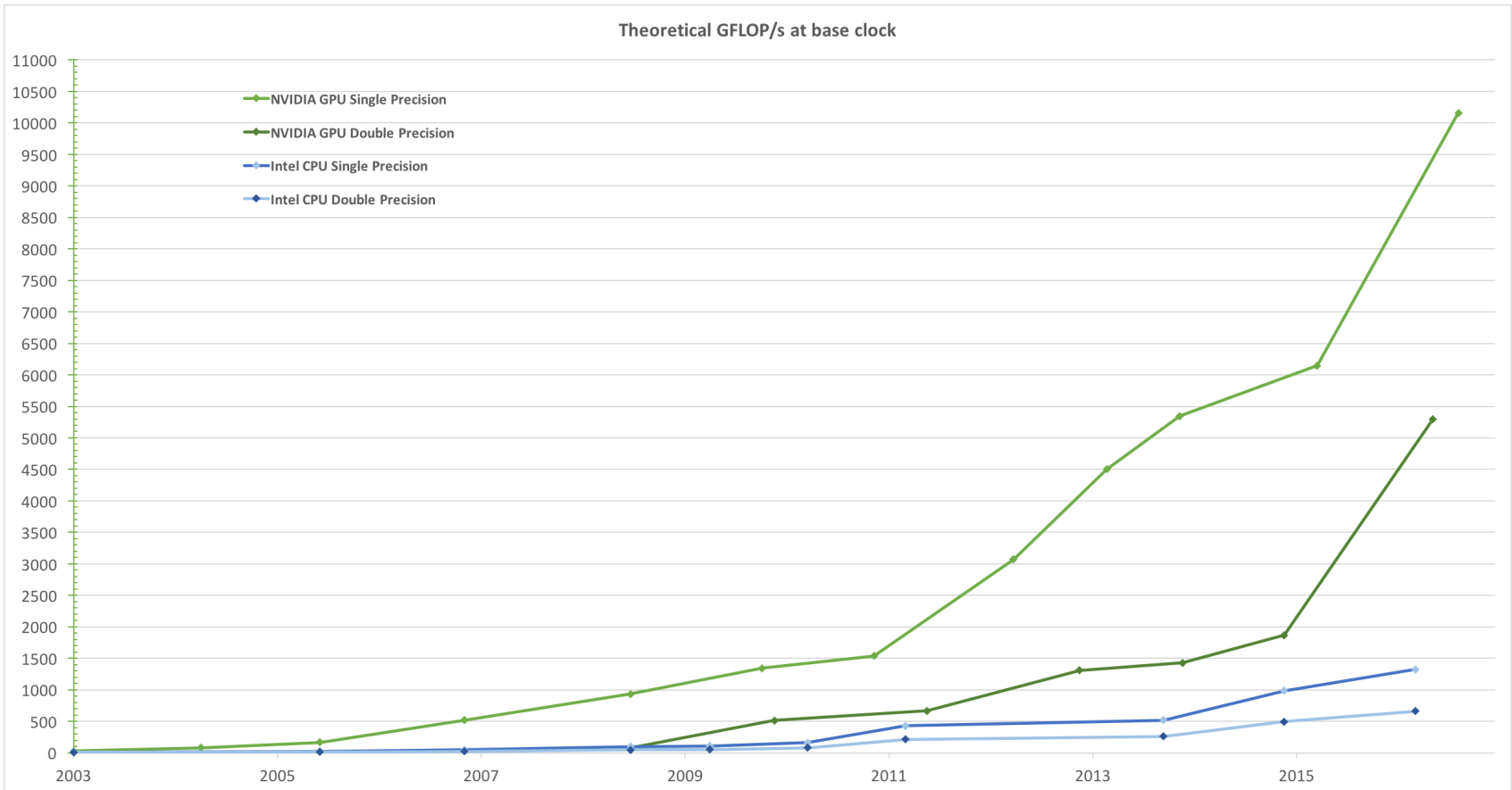
- We assume you have already talked to us about your project ideas when this homework is due and when you have sent the project description via email.
- Detail *what* you are planning to do, and with *whom* you will be cooperating.
- The preferred size of final project teams is two, but if this makes sense in terms of the size of the project, teams of three or doing a project by yourself is fine as well.
- Each team is expected to give a 10 minute presentation about the problem they have worked on and their results and experience during the final week (likely May 4) and hand in a report as well as their code in a repo.
- However, it is important that you call out 4-5 concrete tasks in your project description. We will request frequent updates during the rest of the semester on the progress you are making on these tasks.

Examples of final projects posted on NYU classes

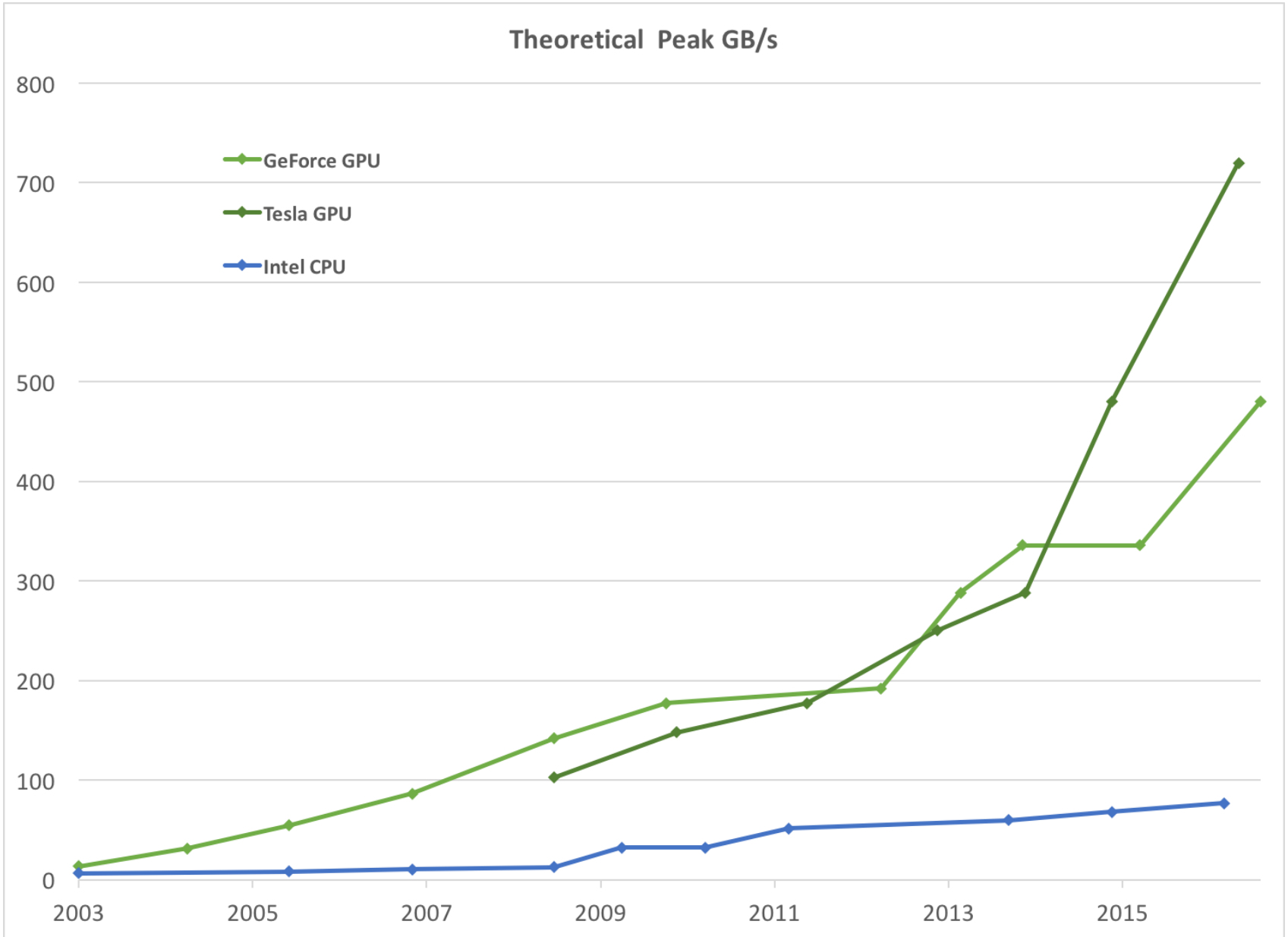
Summary of Previous Classes

- **Memory Hierarchy:** Minimize latency and maximize bandwidth by keeping frequently used data close to CPU in caches.
- **Instruction Level Parallelism:** Pipelining, Superscalar architecture, Out-of-order execution, Branch prediction
- **Vectorization:** Auto-vectorization (compiler hints, loop unrolling), Explicit (OpenMP SIMD, assembly, intrinsics).
- **Shared Memory Parallelism:** OpenMP, threads, fork-join model, synchronization, atomic-operations, NUMA
- **Parallel Scalability:** Strong and weak scaling, Amdahl's Law (speedup $< (s + (1 - s)/p))^{-1}$
- **Libraries:** BLAS, LAPACK, FFTW
- **Tools:** Valgrind, Git, Makefiles

GPUs vs CPUs (FLOP/s)



GPUs vs CPUs (Bandwidth)



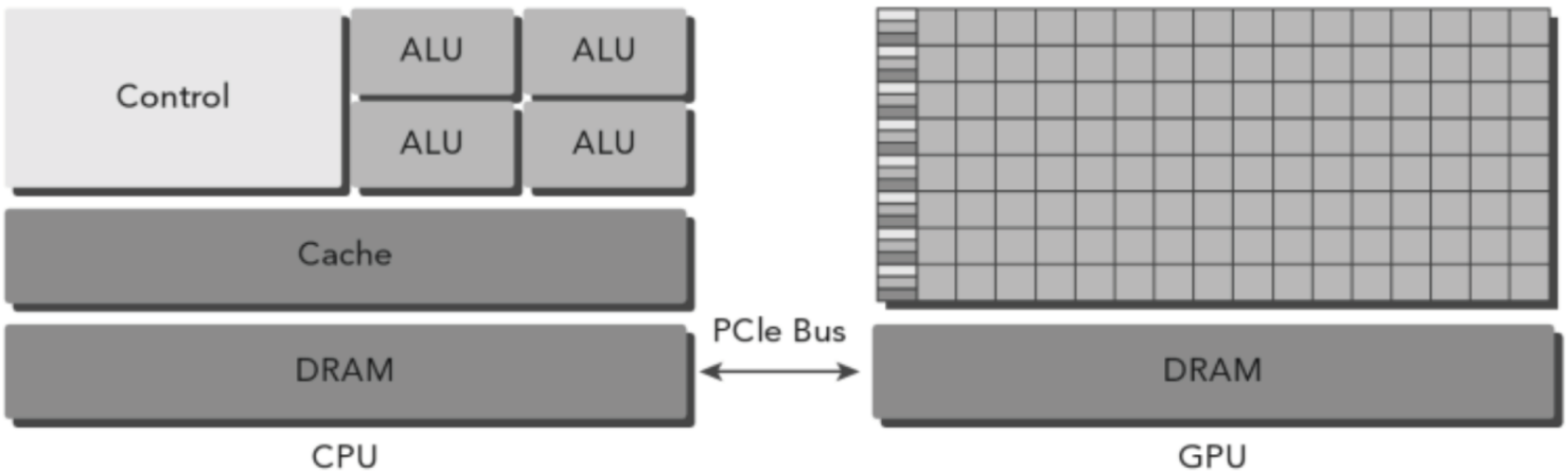
GPUs vs CPUs

CPUs

- Optimized for sequential performance
- Out-or-order execution, branch-prediction

GPUs

- Optimized for floating-point throughput
- No out-of-order, no branch-prediction



Note the difference in number of ALUs

Computing on GPUs

Many different programming models:

NVIDIA

- C for CUDA (Compute Unified Device Architecture)
  - extension of C/C++

Open standards:

- OpenCL: similar to CUDA in programming style, supports other GPUs (AMD) as well as CPUs, other accelerators (Xeon Phi, FPGAs etc).
- OpenACC: similar to OpenMP in programming style, uses pragmas to offload computations to GPU.

Processing flow of CUDA program

CPU (host) connected to one or more GPUs (devices).

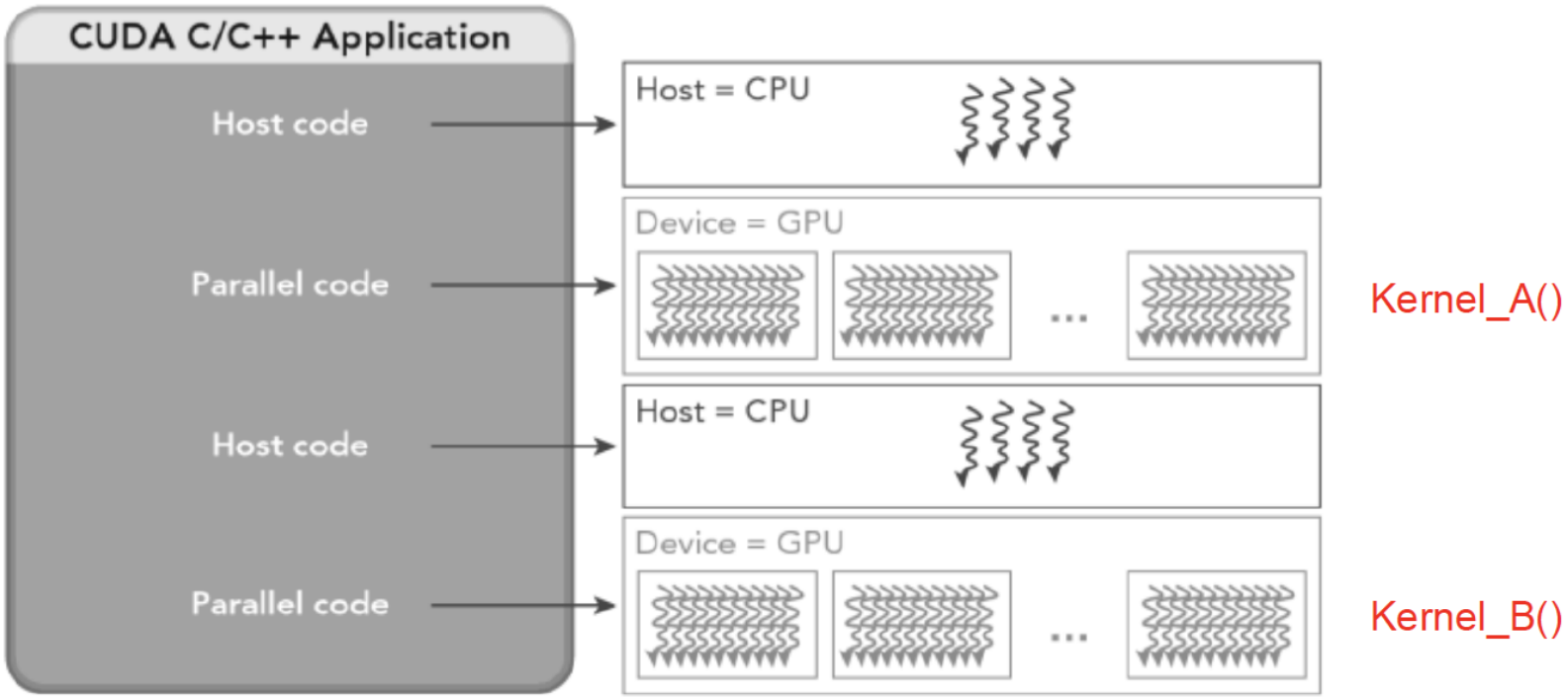
Host executes regular C/C++ code, launches CUDA kernels on device, allocates memory on device, and initiates memory transfers between host and device.

Device executes CUDA kernels.

Divide your code into Host (CPU) and Device (GPU) Code

- Processing flow of a CUDA program:
  - Copy data from CPU memory to GPU memory
  - Invoke kernel to run on the GPU.
  - Copy data back from GPU to CPU memory
  - Release the GPU memory and reset the GPU.
- CUDA code file name extension .cu
- CUDA compiler: nvcc (it compiles .c, .cpp too!) $ nvcc -o a.out a.cu

CUDA program flow



- The kernel function is run concurrently by many threads on the GPU.
- CPU might or might not wait for GPU depending on synchronization.
- Can have more than one kernel functions in your CUDA application.

Thread Hierarchy

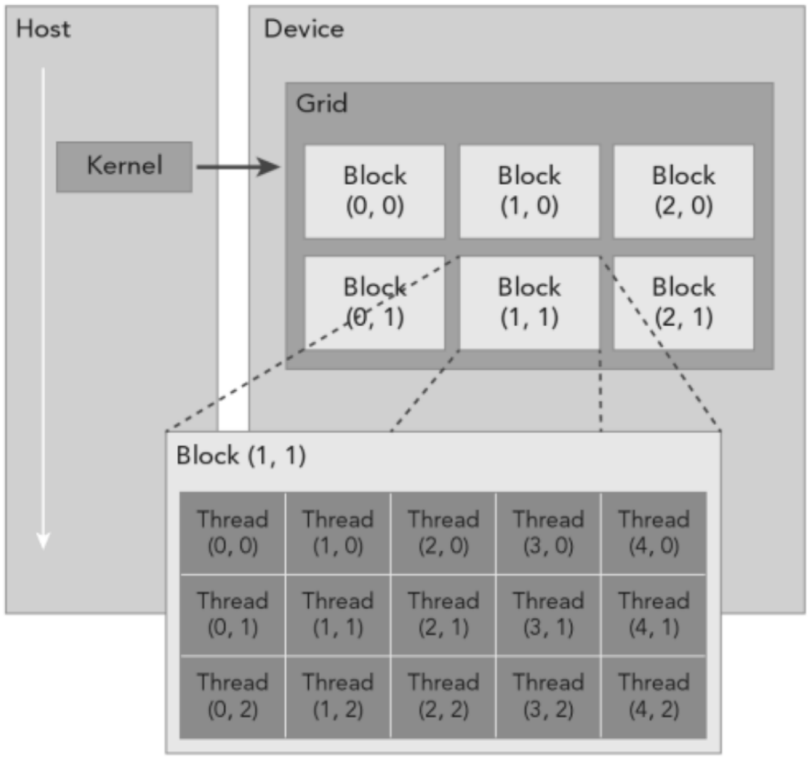
Grid: collection of blocks.

Block: collection of threads. Scheduled on a single streaming multiprocessors (SM).
Threads can share data using shared mem.

blocks are independent, no cross-block synchronisation, execution in undefined order

Warp: group of 32-threads executing together on SM.

Special variables
`gridDim`, `blockDim`, `blockIdx`, `threadIdx`



Calculate global thread index:

```
idx = threadIdx.x + blockDim.x * blockIdx.x
```

Thread Organization - Hardware view

Software view:

grid of blocks,
blocks of threads

Hardware view

- Streaming Multi-Processor (SM) (see right)
- A GTX1080 Ti has 28 SMs.
- Each SM has 128 cores.
- 28*128 = 3584 cores
- A warp = 32 consecutive threads

A Quarter of a Pascal SM

- A thread block can be assigned to only one steaming multiprocessor.
- One multi-processor can have many blocks assigned to it.
- Threads within a block are grouped into warps, each warp has 32 consecutive threads.
- Number of threads in a block should be a multiple of 32 (the warp size).
- If there are fewer than 32 threads, then the rest of the threads are masked out (deactivated)

Supported CUDA level of GPU

Version features and specifications [edit]

Feature support (unlisted features are supported for all compute abilities)	Compute capability (version)										
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x
Integer atomic functions operating on 32-bit words in global memory	No	Yes									
atomicExch() operating on 32-bit floating point values in global memory											
Integer atomic functions operating on 32-bit words in shared memory	No	Yes									
atomicExch() operating on 32-bit floating point values in shared memory											
Integer atomic functions operating on 64-bit words in global memory											
Warp vote functions											
Double-precision floating-point operations	No			Yes							
Atomic functions operating on 64-bit integer values in shared memory	No				Yes						
Floating-point atomic addition operating on 32-bit words in global and shared memory											
_ballot()											
_threadfence_system()											
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()											
Surface functions											
3D grid of thread block											
Warp shuffle functions , Unified Memory	No					Yes					
Funnel shift	No						Yes				
Dynamic parallelism	No							Yes			

Demo: Hello World

GPU code (kernel function):

```
__global__ // tell compiler this runs on GPU
void print_hello() {
  printf("hello from thread %d of block %d\n",
                              threadIdx.x, blockIdx.x);
}
```

Launching the kernel (from CPU)

```
dim3 GridDim( nx, ny, nz ); // can be 1D, 2D or 3D
dim3 BlockDim( mx, my, mz ); // can be 1D, 2D or 3D
print_hello<<<GridDim, BlockDim>>>(); // launch kernel
cudaDeviceSynchronize(); // wait for kernel to finish
```

- One kernel is run per thread

Compiling with NVCC (NVIDIA C Compiler)

```
nvcc hello-world.cu
```

nvcc compiles device functions, host code processed by gcc/g++.

DEMO: gpu01.cu

Synchronizing the Device

You probably have noticed this line in the previous example

```
cudaDeviceSynchronize();
```

- CUDA programming model is asynchronous between CPU and GPU.
- The cudaDeviceSynchronize() force the CPU to wait for the kernel code to finish before moving on.
- (A CPU timer will not count runtime of kernel if CPU does not wait for the kernel.)

How do I get information about the GPU on a node?

To get number of GPU cards on a node:

```
cudaError_t cudaGetDeviceCount(int * dev_count);
```

To get device properties of a device:

```
cudaDeviceProp devProp
cudaGetDeviceProperties(&devProp, dev_number);
printf("device name: %s", devProp.name);
```

Full list of device properties:
https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html

Monitoring GPU activity

```
$ nvidia-smi
Tue Mar 17 15:54:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Quadro P2000        On   | 00000000:65:00.0 Off |                  N/A |
| 44%   23C    P8     4W /  75W |    543MiB /  5025MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1188      G   /usr/lib/xorg/Xorg                           359MiB |
|    0      1260      G   /usr/bin/gnome-shell                         181MiB |
+-----------------------------------------------------------------------------+
```

DEMO: gpu02.cu

DEMO: gpu02.cu

Declaration Syntax:

```
__global__ void name(arg1, arg2, ...) {
function body;
}
```

- __global__ is a function type qualifier.
- Kernel function is invoked by CPU, but run on GPU in many copies (one thread per copy).
- Kernel invoking Syntax:

```
name<<<grid, block>>>(arg1, arg2, ...)
// both grid, block are of type dim3, e.g,
dim3 gridDim(4,1,1); dim3 blockDim(16,1,1);
```

```
__global__ void name1(arg1, arg2, ...){
name2(arg1,arg2); // invoke device function
}
__device__ double name2(arg1, arg2, ...){
function body;
}
__host__ float name3(arg1, arg2, ...){
function body;
}
```

Host and device routines only run on CPU, and GPU respectively.
Global declares kernel function, run on GPU, which can call device functions.

What should be in the Kernel function?

```
for (i = 0; i < 1000; i++)
C[i] = A[i] + B[i];
}
```

What should be in the Kernel function?

```
for (i = 0; i < 1000; i++)
C[i] = A[i] + B[i];
}
```

```
 __global__ kernel (int* A, int* B, int* C) {
id = threadIdx.x + blockIdx.x*blockDim.x;
C[id] = A[id] +B[id];
}
```

In essence, your for loop with for peeled off, but keep the things inside.

The key part is to map your data to threads (array indices).

What should be in the Kernel function?

```
for (i = 0; i < 1000; i++)
C[i] = A[i] + B[i];
}
```

```
 __global__ kernel (int* A, int* B, int* C) {
id = threadIdx.x + blockIdx.x*blockDim.x;
C[id] = A[id] +B[id];
}
```

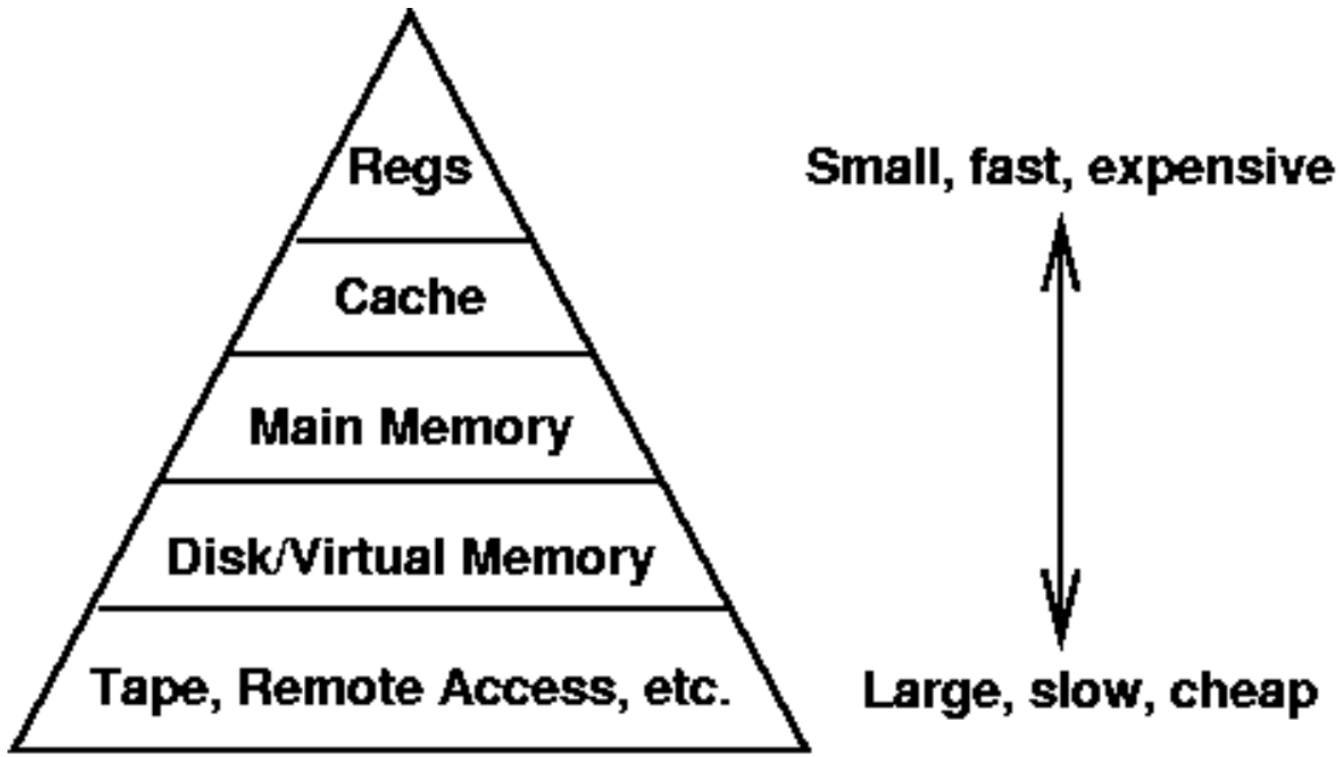
In essence, your for loop with for peeled off, but keep the things inside.

The key part is to map your data to threads (array indices).

Before we can run this kernel, need to look at memory of GPU devices

CUDA Memory Model

- **Similar to thread hierarchy, GPU has a memory hierarchy, and CUDA expose a lot of this hierarchy to you.**



CPU Memory Hierarchy

Memory Hierarchy

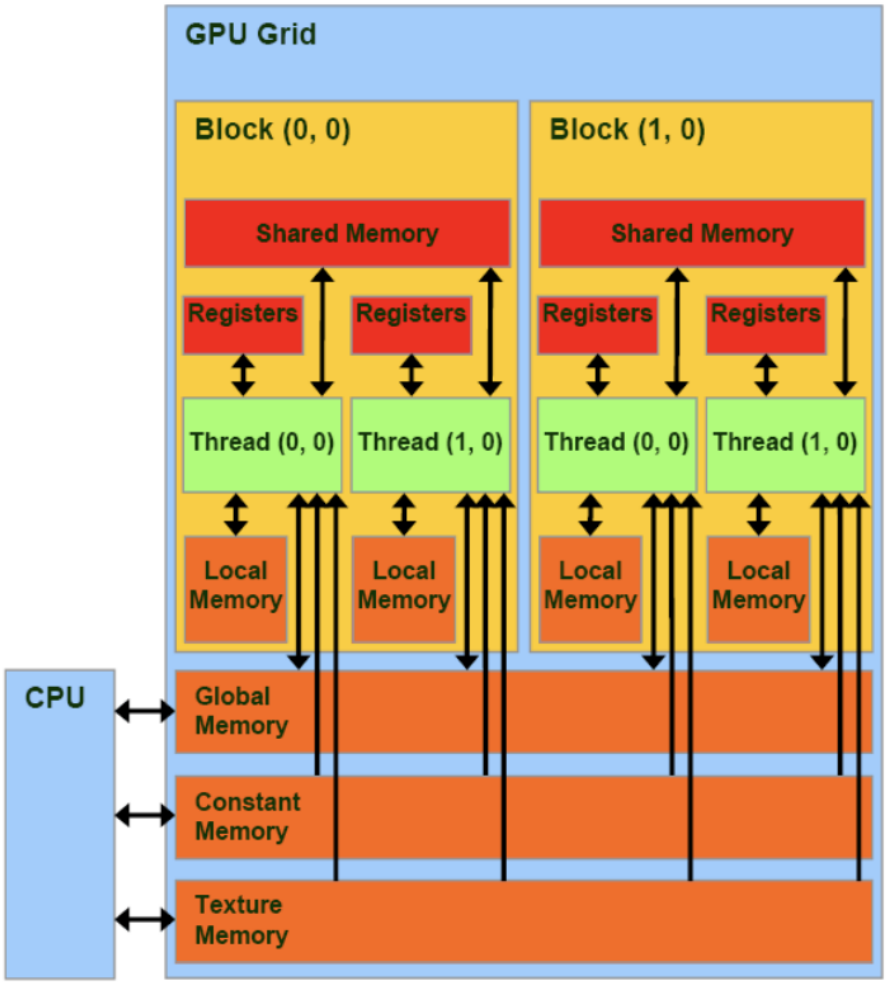
Local memory: Local to each thread (large arrays or register spilling), reside in GPU global memory (slow!).

Shared memory: accessible by all threads in the same thread-block. Eg: `__shared__ A[100];`

Global memory: accessible by all threads, resides in DRAM. Eg: `cudaMalloc(&A,10*sizeof(double))`

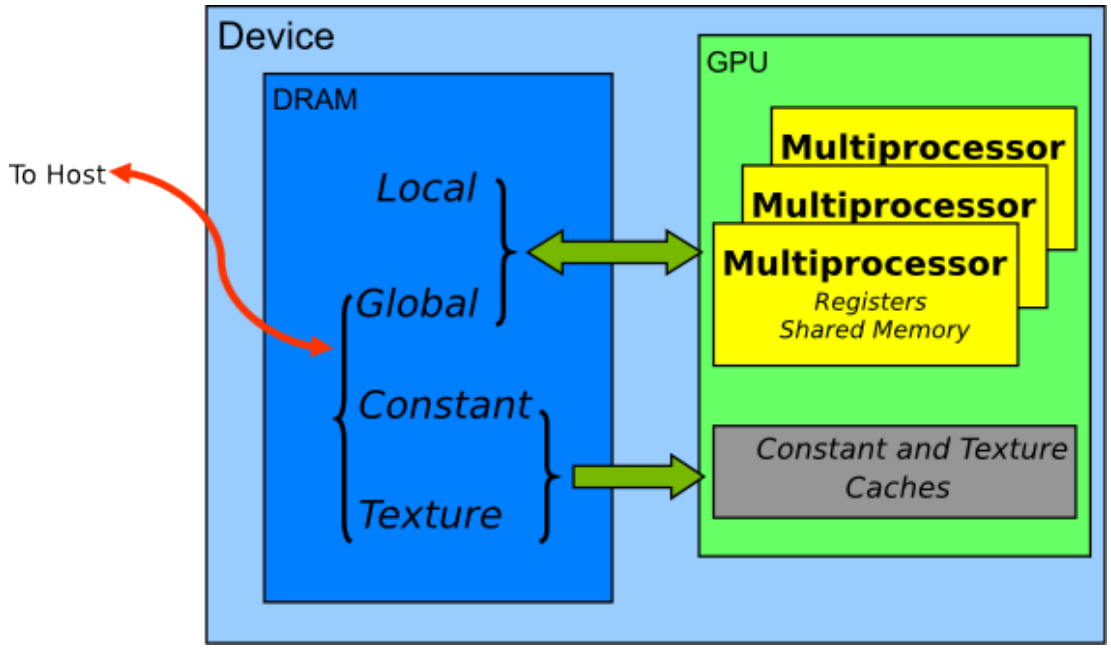
Constant memory: read-only, resides in DRAM and constant cache.

Texture memory: read-only, resides in DRAM and texture cached.



Device Memory Spaces

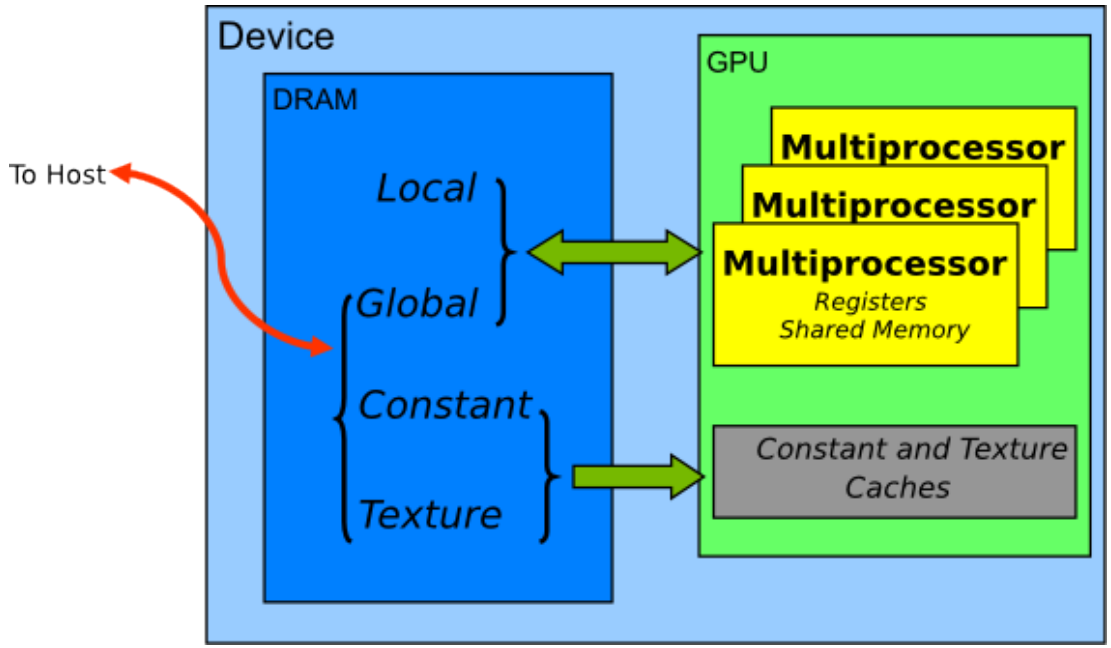
Registers: Thread scope local variables.




```
__global__ void mykernel() {
    int a = 0;
    double x = 1.5;
}
```

Device Memory Spaces

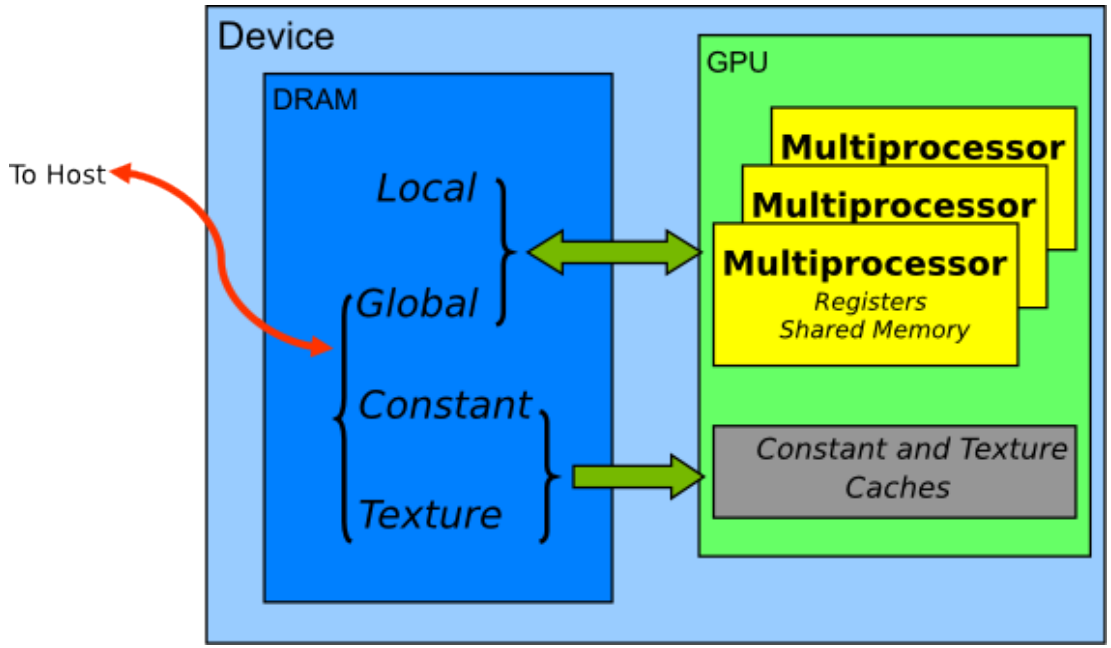
Local memory: Thread scope arrays and spilled registers. Resides in GPU main memory.




```
__global__ void mykernel() {
    int a[100];
    double x[100];
}
```

Device Memory Spaces

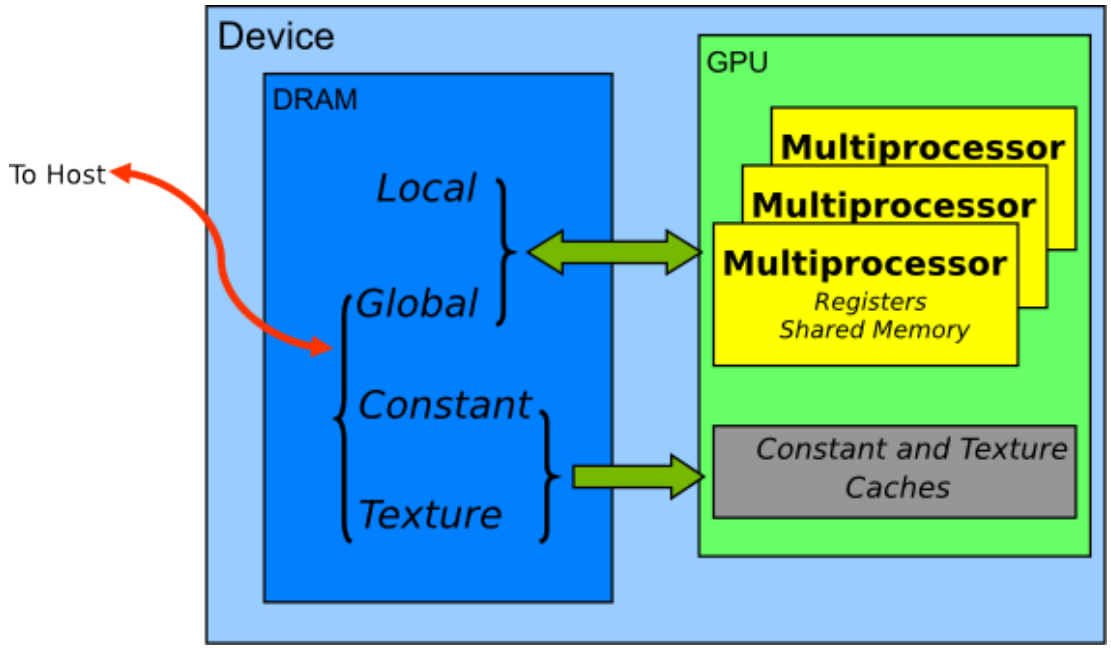
Shared memory: Variables declared using `__shared__` keyword. Accessible by all thread in the same thread-block.




```
__global__ void mykernel() {
    __shared__ double x[1024];
    x[threadIdx.x] = 1.5;
}
```

Device Memory Spaces

Global memory: Dynamically allocated GPU main-memory. Managed (allocate, transfer data, free) from CPU. Cached in L2 cache.
size = size-of-DRAM

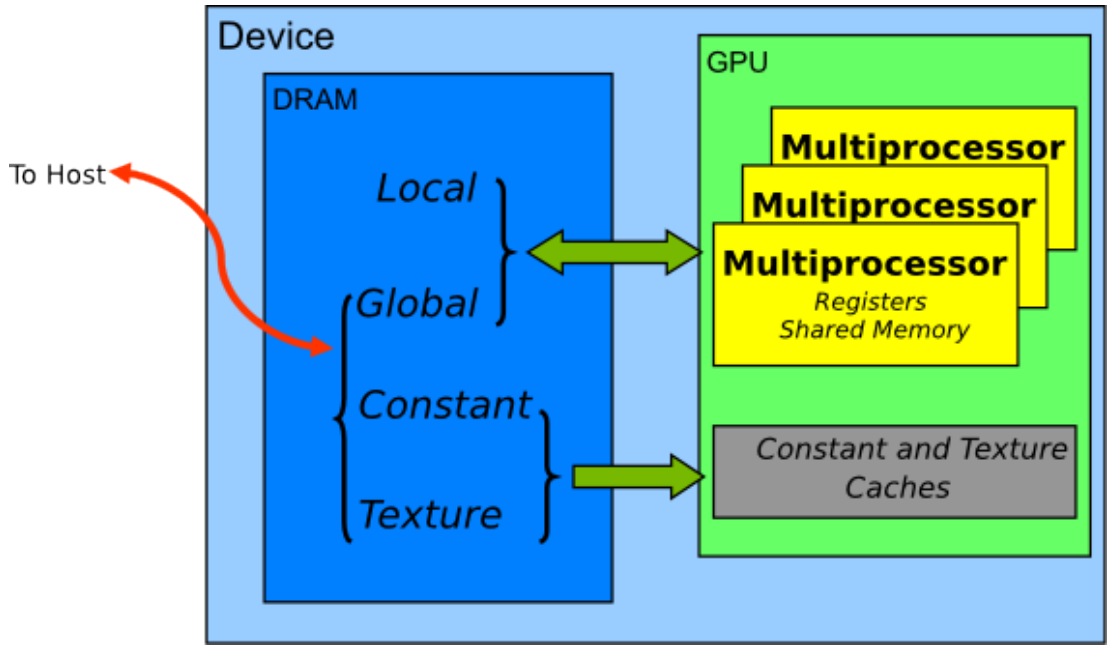



```
int main() {
    double* A; // pointer
    cudaMalloc(&A, 512*sizeof(double)); // allocate
        on GPU
    mykernel<< <15,1028> >>(A); // pass pointer to
        kernel
}
```

Note: It is possible to use malloc in a __device__ function, but slow so try to avoid it

Device Memory Spaces

Constant memory: Global scope, read-only (from GPU) memory declared using `__constant__` keyword.
size = 64KB



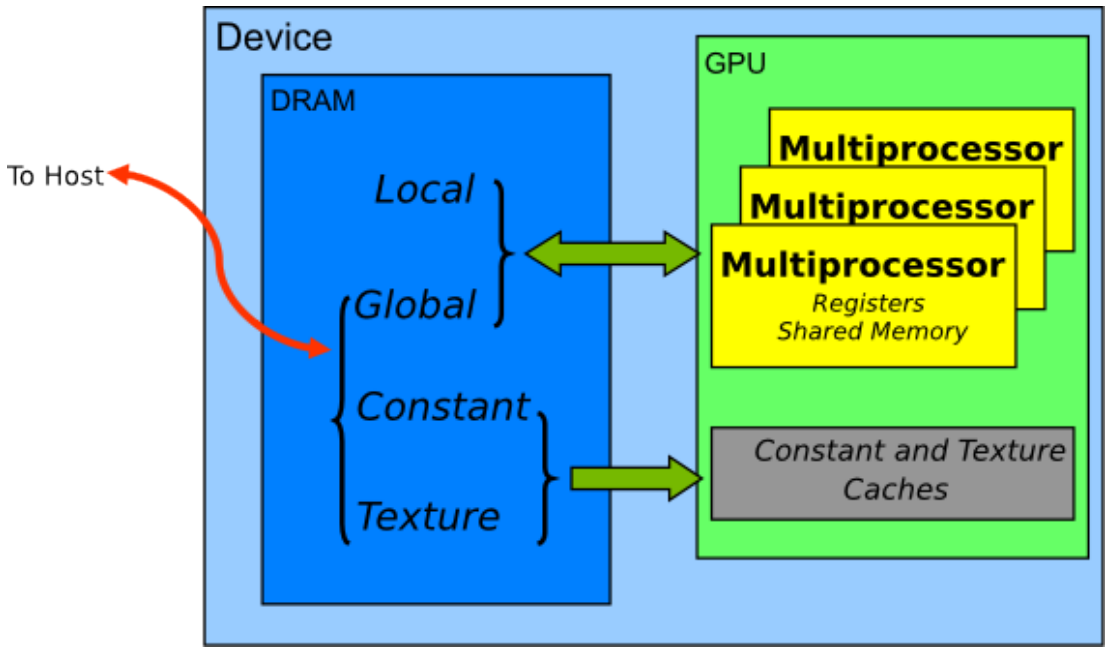

```
__constant__ double A[512];

__global__ void mykernel(){
    double x = A[threadIdx.x];
}
```

Device Memory Spaces

Texture memory: No ordinary memory read.

- Can perform filtering and/or translation of values by treating the texture residing in memory as the sampled/discrete representation of a 1D, 2D, or 3D function.
- Special access functions are implemented in hardware, making them very fast and efficient to use.
- Size and representation limitations make the use of texture memory rather limited in the GPGPU domain.



Resource Limits

Limits for compute capability 7.0 (Volta architecture)

Threads per Warp	32
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)
Total amount of constant memory	64KB
Total amount of shared memory per block	48KB
Max threads per block	1024
Warps per SM	64
Threads per SM	2048
Thread Blocks per SM	32
Shared Memory per SM	96KB

More information:

- compute capabilities (features and limits)
- CUDA Samples: `$CUDA_HOME/samples/1_Utilities/deviceQuery`

DEMO: gpu04.cu

```
__global__ void kernel() {
        double a = 2.73; // register variable
        double c[100]; // local variable
        __shared__ double b; // shared variable
        int tx = threadIdx.x; // register variable

        if(tx == 0) {
                b = 3.1415;
        }
        printf("id = %d, a = %.5f, b = %.5f\n", tx,
            a, b);

}
```

What is the output?

DEMO: gpu04.cu

```
__global__ void kernel() {
        double a = 2.73; // register variable
        double c[100]; // local variable
        __shared__ double b; // shared variable
        int tx = threadIdx.x; // register variable

        if(tx == 0) {
                b = 3.1415;
        }
        printf("id = %d, a = %.5f, b = %.5f\n", tx,
            a, b);

}
```

What is the output? Only tx $= 0$ shows b $= 3.1415$ rest executes statement before b is written.

Thread synchronization

Syncrhonize threads of a kernel

- Threads within a block can be synchronized

```
__syncthreads();
```

- Threads of different blocks CANNOT be synchronized.

  They should NOT be (possible dead lock! because the blocks might need to run on the same SM and blocks waiting for sync are not preempted).

- Different blocks can be scheduled to start at different time by the GPU.

DEMO: gpu04.cu

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
   for (long idx = 0; idx < N; idx++)
      xpy[idx] = x[idx] + y[idx];
}
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
   for (long idx = 0; idx < N; idx++)
      xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long
    N){
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
   for (long idx = 0; idx < N; idx++)
      xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long
    N){
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Launching the kernel (from CPU):

```
vadd_gpu<<<GridDim, BlockDim>>>(xpy, x, y, N);
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
   for (long idx = 0; idx < N; idx++)
      xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long
    N){
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Launching the kernel (from CPU):

```
vadd_gpu<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(xpy, x, y, N);
```

Global Memory Management

CUDA API for Managing Memory:

Memory allocation on device

```
cudaError_t cudaMalloc(void** A, size_t size);
```

Free memory on device

```
cudaError_t cudaFree(void* A);
```

Memory transfer between device and host

```
cudaError_t cudaMemcpy(void* A_d, const void* A,
                       size_t N, cudaMemcpyKind kind);
```

kind = `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`

DEMO: gpu03.cu

Checking Errors

All CUDA API calls return an error code (`cudaError_t`)

- Error in the API call itself
- Error in an earlier asynchronous operation (e.g. kernel)

Return the last error code, and reset it to cudaSuccess:

```
cudaError_t cudaGetLastError(void)
```

Return the last error code, but do NOT reset it:

```
cudaError_t cudaPeekAtLastError(void);
```

Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Example: malloc() might fail

Thread/Warp Divergence

Threads of the same warp work in the SIMT mode

- SIMT: single instruction multiple threads
- Only one instruction can be executed at one time item Warp divergence: when threads in the same warp are executing different instructions. For example,

```
id = threadIdx.x;
if ( id < 16 ) {
printf("I take branch one");
} else {
printf("I take branch two");
}
```

Performance will be degraded because of warp divergence.