AMS 250: An Introduction to High Performance Computing

Manycore Computing

Shawfeng Dong

shaw@ucsc.edu

(831) 459-2725

Astronomy & Astrophysics

University of California, Santa Cruz

Outline

- Why GPU / Accelerator-based / **Manycore Computing**?
- GPU Architecture
- Introduction to CUDA C Programming
- GPU-Accelerated Libraries
- OpenACC
- Intel MIC / Xeon Phi

Why Manycore Computing?

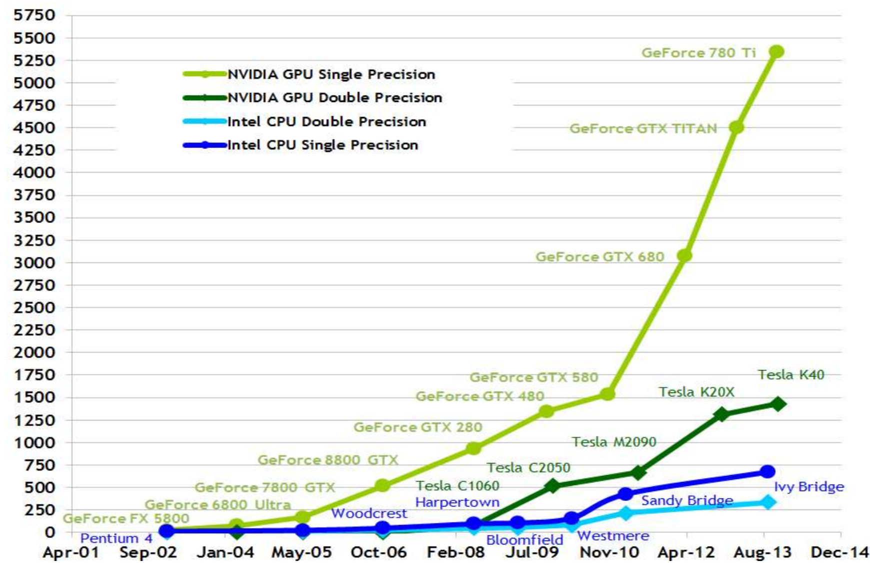
"I think they're right on the money, but the huge performance differential (currently 3 GPUs ~= 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers."

-John Stone, UIUC, circa 2007

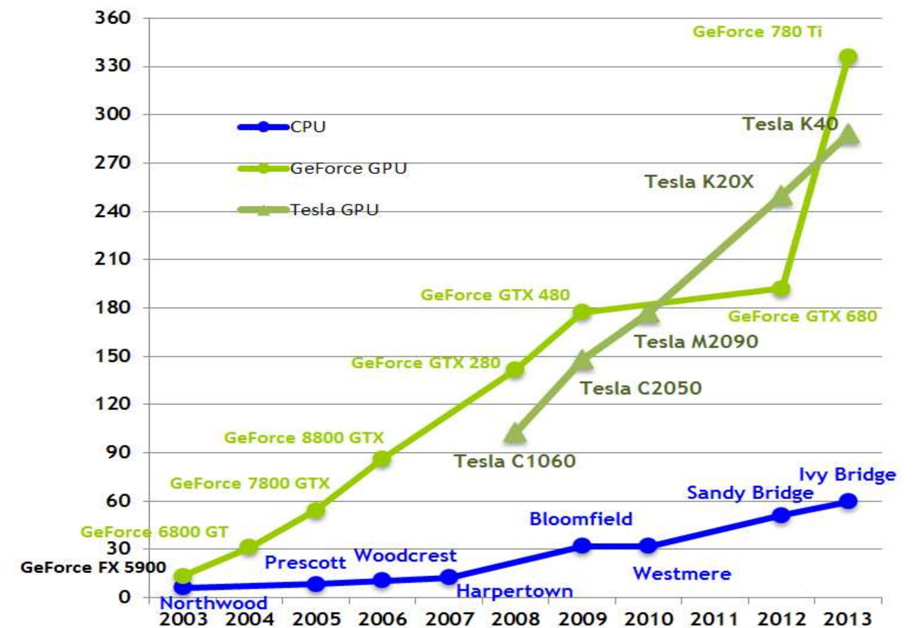
Is a top-of-the-line GPU 100 times faster than a top-of-the-line CPU?

Why Manycore Computing?

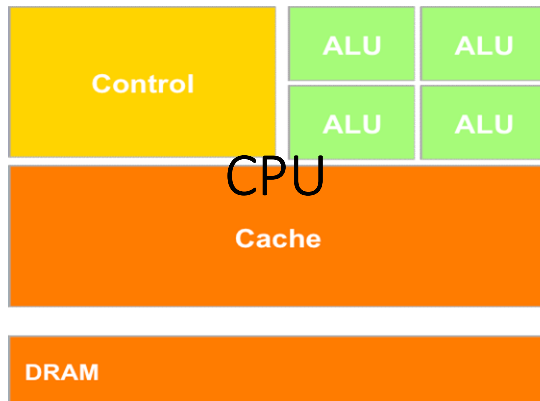
FLOPS

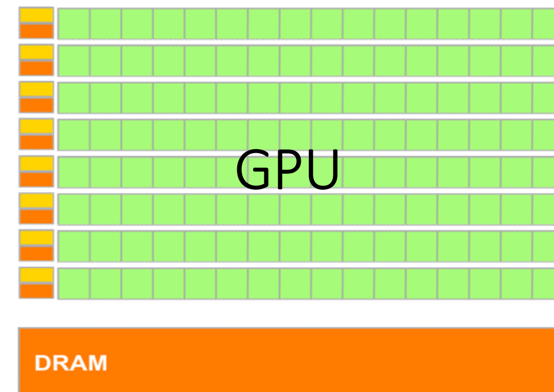


Memory Bandwidth



Multicore CPU vs. Manycore GPU



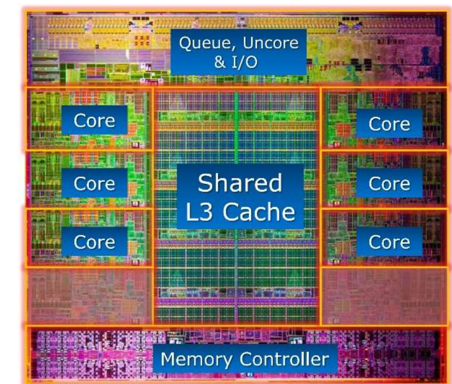


- Optimized for low latency access to cached data sets
- Control logic for out-of-order and speculative execution

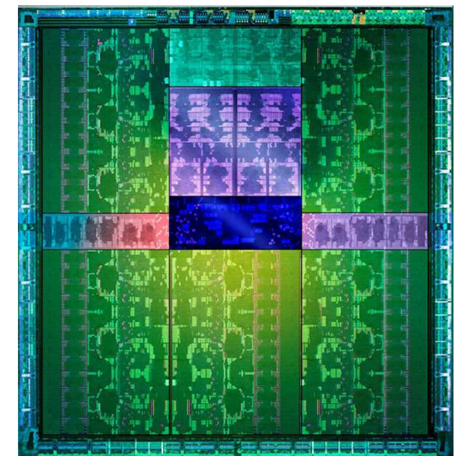
- Optimized for data-parallel, high throughput computing
- Architecture tolerant of memory latency
- More Transistors dedicated to computation

Multicore versus Manycore – Real Chips

	Intel Xeon E5-2650	Nvidia Tesla K20	
Architecture	Sandy Bridge	Kepler	
Processing Units	8 cores (with 256-bit AVX)	13 SM / 2496 cores / 832 DP units	
Clock Speed	2.0 GHz	706 MHz	
Single Precision	256 GLOPS	3.52 TFLOPS	
Double Precision	128 GFLOPS	1.17 TFLOPS	~9x
Memory Bandwidth	51.2 GB/s	208 GB/s	~4x
Memory Size	32 GB	5 GB	
Registers	~100 per core	65536 x 32-bit per SM	
L1 Cache	64 KB per core	64 KB per SM	
L2 Cache	256 KB per core	768 KB shared	
L3 Cache	20 MB shared	N/A	
Process	32nm	28nm	
Transistor Count	2.3B	7.1B	
Thermal Design Power	95W	225W	



Intel Sandy Bridge (32nm)

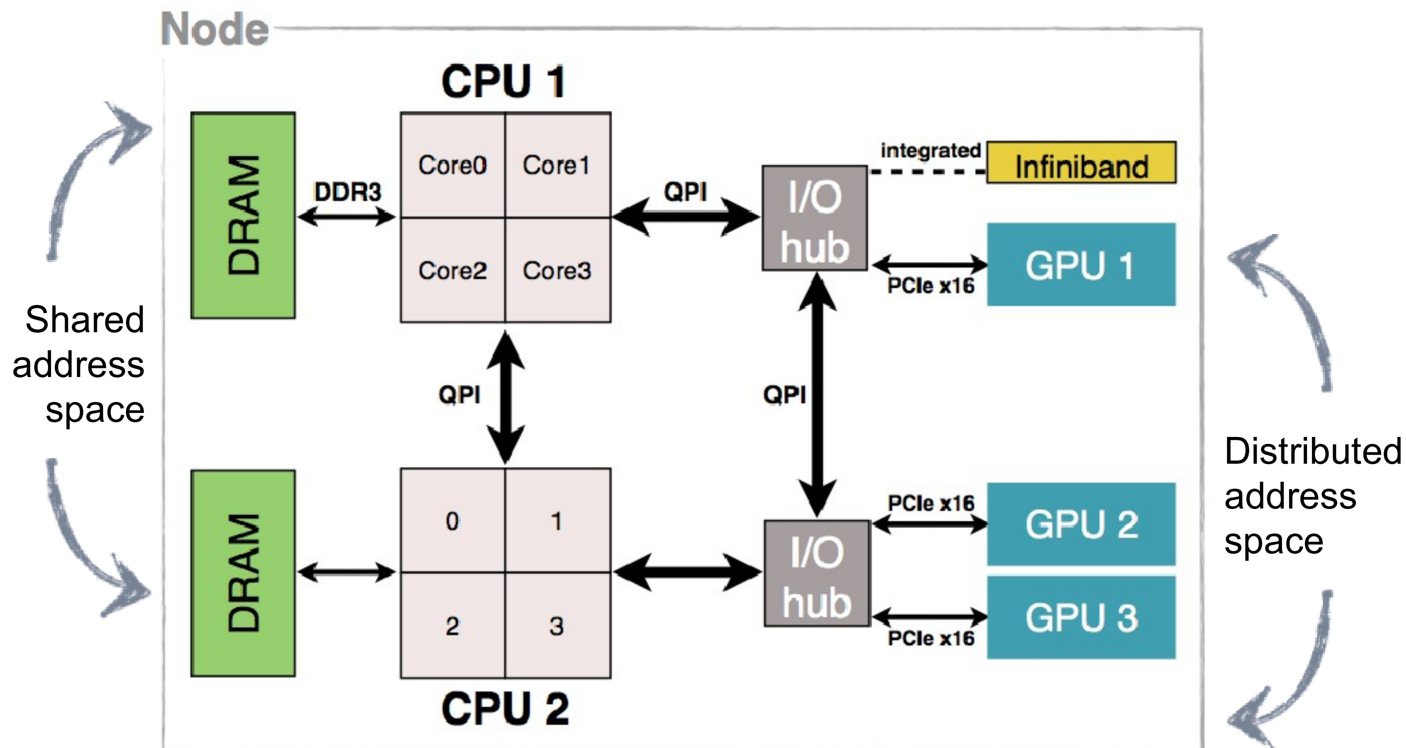


Nvidia GK110 (28nm)

Connecting Manycore GPUs to Multicore CPUs

Typically, GPU devices are accessed over PCIe

- Bandwidth of PCIe 2.0 x16 = 8 GB/s (18GB/s duplex)



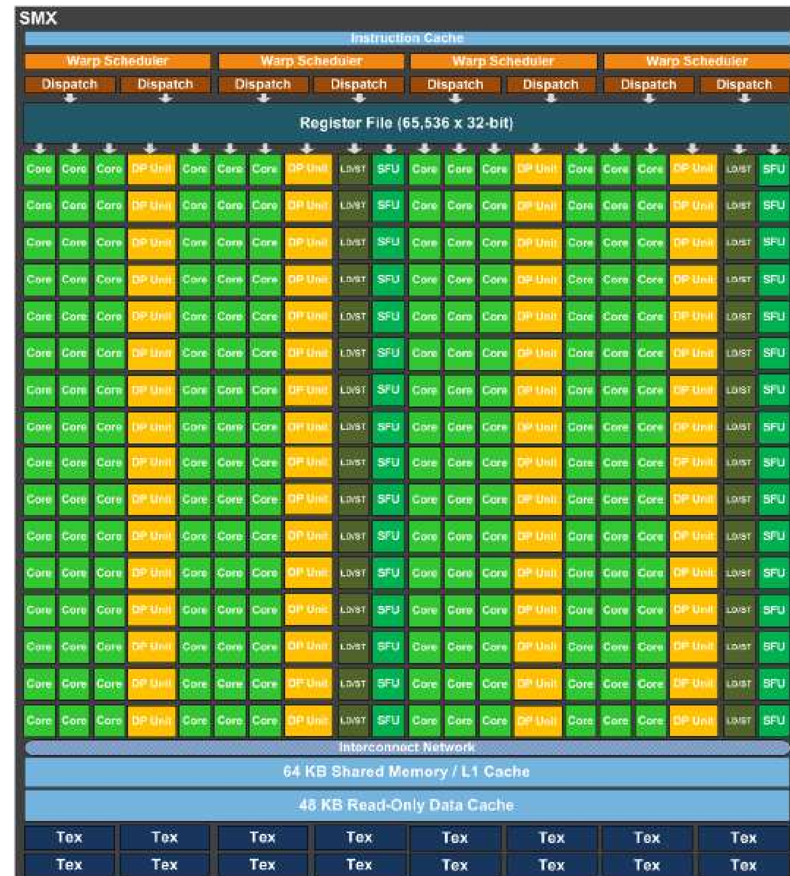
Nvidia Tesla K20 GPU

- Kepler microarchitecture
- 2496 CUDA cores / 832 DP units / 13 SMs
- Core speed: 706 MHz
- Double precision performance: 1.17 TFLOPS
  = 0.706 (GHz) x 832 (DP units) x 2 (FMA) ~ 9x CPU
- Single precision performance: 3.52 TFLOPS
  = 0.706 (GHz) x 2496 (CUDA cores) x 2 (FMA)
- Memory: 5.2GHz, 320-bit wide, 5GB GDDR5
  5.2 (GHz) x 320 / 8 = 208 GB/s ~ 4x CPU
- PCI express *2.0* x16
  500 (MB/s) x 8/10 x 16 = 8 GB/s (16 GB/s duplex)

http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last

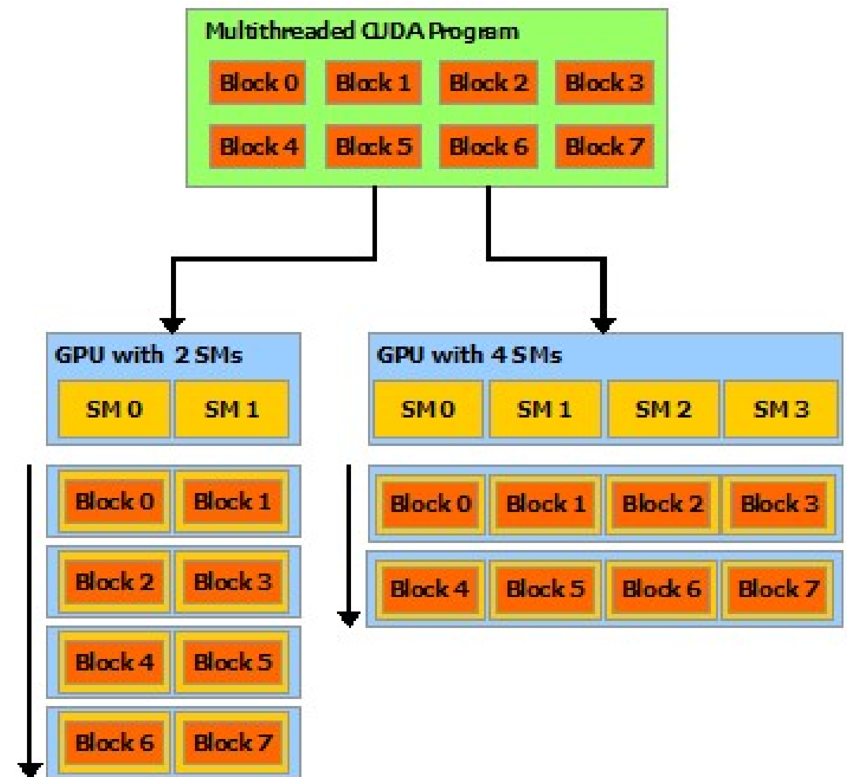
Streaming Multiprocessors (SM)

- GPU
  - A number of SMs (Steaming Multiprocessors)
  - Memory
- Each SM has its own
  - Control unit
  - Pipeline for execution
    192 cores, 64 DP units per SM on K20
  - Registers
  - Shared memory/L1 cache
  - Read-only data cache
  - Texture unit
- SM is the unit of transparent scalability



Transparent Scalability

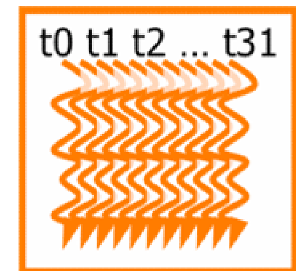
- Threads are assigned to SMs in block granularity
  - Coarse-grained data parallelism
  - Task parallelism
- Threads of a thread block execute concurrently on *one* SM
- Multiple thread blocks can execute concurrently on the same SM
  - Up to 2048 threads per SM
  - Up to 16 blocks per SM (Kepler GK110)
- Each block can execute in any order relative to other blocks



https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-kepler-gpu-accelerators/

Single Instruction Multiple Threads

- SIMT (**S**ingle **I**nstruction **M**ultiple **T**hreads) is the execution model on GPUs
- SIMT is a special case of SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata)
- Fine-grained data parallelism and thread parallelism within a thread block
- GPU threads are lightweight and fast switching
- GPU needs 1000s of threads for full efficiency
  - Max threads per SM = 2048 (on Kepler GK110)
  - Could be 8 blocks if 256 threads per block (typical)
  - Could be 16 blocks if 128 threads per block
- Each block is executed as 32-thread ***warps*** (32-way SIMD)
  - An implementation decision, not part of CUDA programming model
  - Warps are scheduling units in SM
  - If threads of a warp diverge via conditional branch, the warp *serially* executes each branch path taken



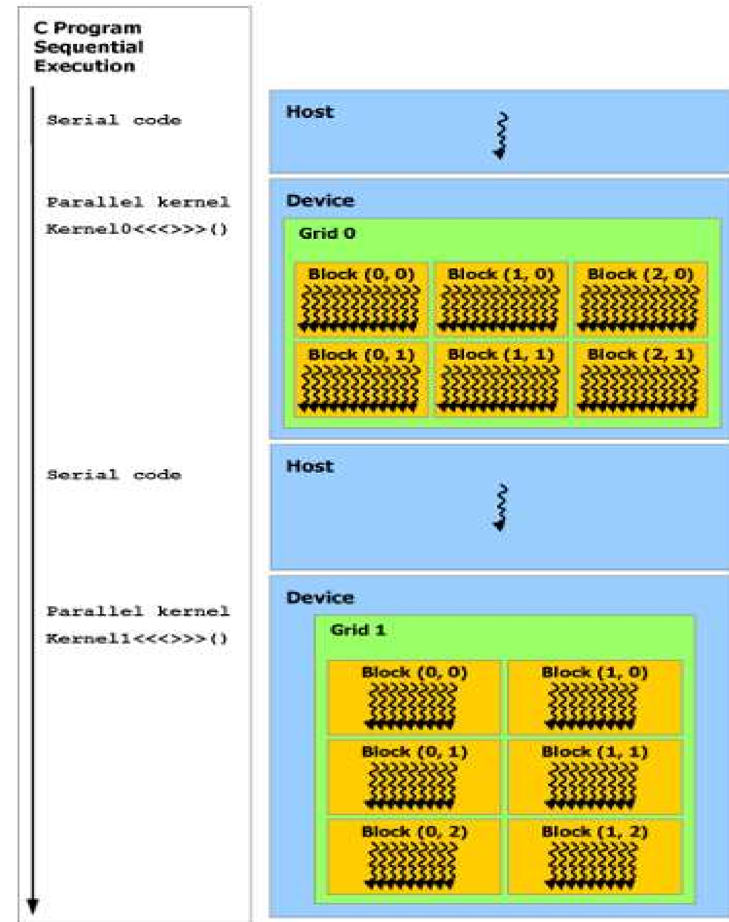
CUDA Programming Model

- CUDA = **Compute Unified Device Architecture**
- General purpose programming model for GPU
  - GPU as a dedicated super-threaded, massively data parallel co-processor
  - Optimized for computation (graphics-free API)
  - Data sharing with OpenGL buffer objects
  - Explicit GPU memory management
- CUDA C consists of a minimal set of extension to the C language and a runtime library
  - Declspecs
  - Built-in types
  - Built-in variables
  - Runtime functions
  - Kernel launches

Heterogeneous Programming

Integrated host+device C/C++ program

- Serial or modestly parallel parts in host C code
- Highly parallel parts in device kernel code



A Sample CUDA program implementing BLAS level-1 function *DAXPY*

A * X + Y

where

X & Y are vectors

A is a scalar

https://pleiades.ucsc.edu/hyades/GPU_QuickStart_Guide

```
#define N 20480
// declare the kernel
__global__ void daxpy(double a, double *x, double *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(double);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

CUDA C Extensions

CUDA Runtime Functions

```
#define N 20480
// declare the kernel
__global__ void daxpy(double a, double *x, double *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(double);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

Kernels

- A kernel run N times in parallel by N different *CUDA threads*
- Defined using the __global__ (2 underscores before *and* after **global**) declaration specifier (must return *void*)
- Launched using the <<<…>>> *execution configuration syntax*
- Kernel Launches are ***asynchronous*** with respect to the host
  - They return immediately!

Example:

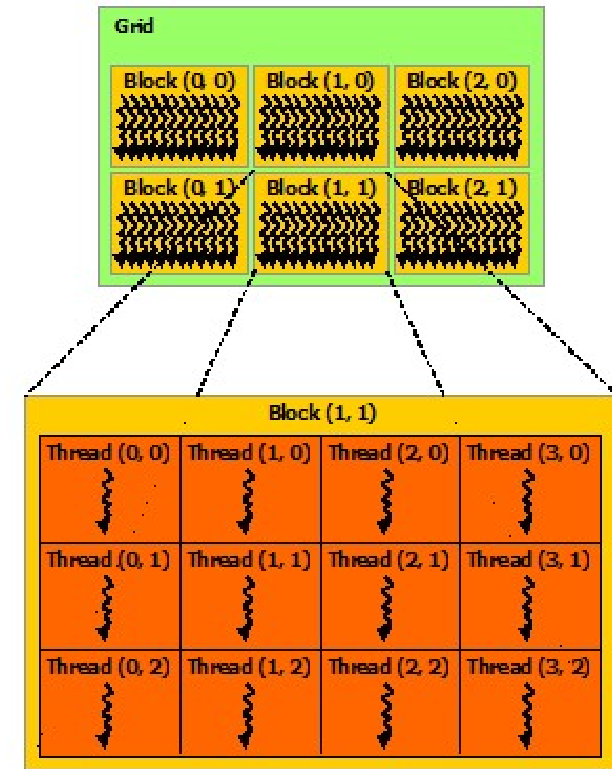
```
__global__ void daxpy(double a, double *x, double *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        y[i] += a*x[i];
    }
}

    daxpy<<<N/256,256>>>(a, dx, dy);
```

Thread Hierarchy

A **Grid** is a collection of **Threads**. Threads in a Grid execute a *Kernel Function* and are divided into Thread **Blocks**.

- Threads within a block (running on the same SM) cooperate via shared memory, atomic operations and barrier synchronization
- Threads in different blocks cannot cooperate (maybe running on different SMs)
- Each grid is executed on one device



Thread Block Size

- Between 256 and 512 threads per block is usually a good choice
  - Overly small blocks will limit the # of concurrent threads due to limitation on maximum # of concurrent blocks/SM
  - Overly large blocks can hinder performance, e.g., by increasing cost of any synchronization/barrier among all the threads in a block
- All CUDA-capable GPUs to date prefer # of threads per block to be a multiple of 32 if possible
  - 32 threads is the ***warp*** size. Non-multiples of 32 waste some resources and cycles
  - A multiple of 32 threads wide (x-dimension) facilitates coalesced memory access to adjacent memory addresses

What CUDA Supports

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across thread in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams

Built-in Variables

The following built-in variables specify the grid and block dimensions and block and threads indices:

- gridDim is of type dim3 and contains the *dimensions* of the **grid**
- blockIdx is of type uint3 and contains the **block** *index* within the **grid**
- blockDim is of type dim3 and contains the *dimensions* of the **block**
- threadIdx is of type uint3 and contains the **thread** *index* within the **block**
- warpSize is of type int and contains the warp size in threads

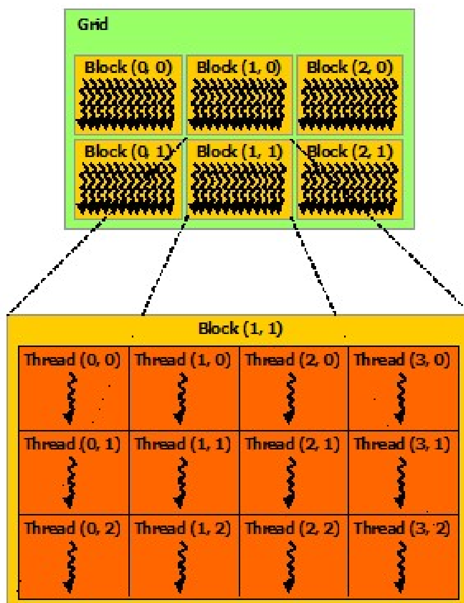
where

uint3: 3-component vector of uint. It is a structure, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z, respectively.

dim3: integer vector type based on uint3. When defining a variable of type dim3, any component left unspecified is initialized to 1.

Thread IDs

Each thread uses IDs to decide what data to work on

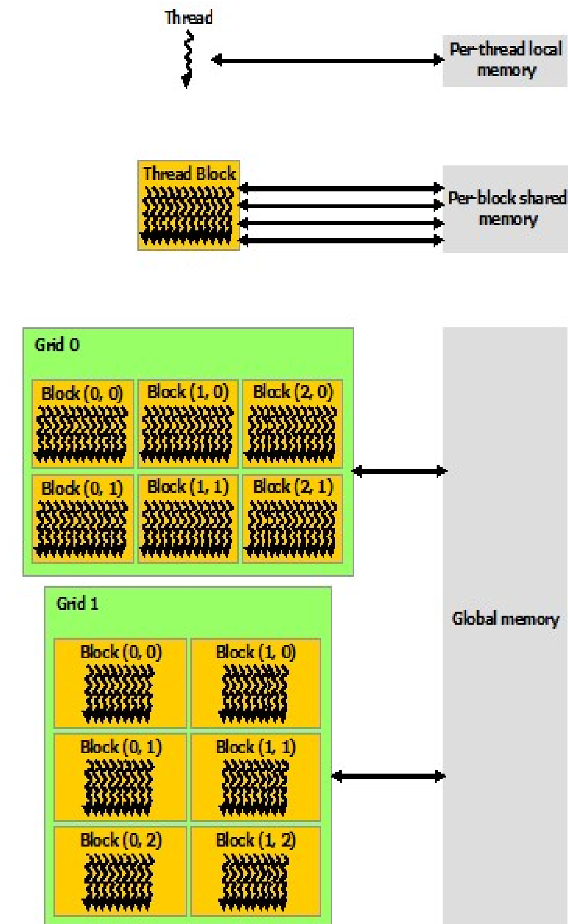



```
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    . . .
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N/16, N/16);
    MatAdd<<<numBlocks,threadsPerBlock>>>(A, B, C);
    . . .
}
```

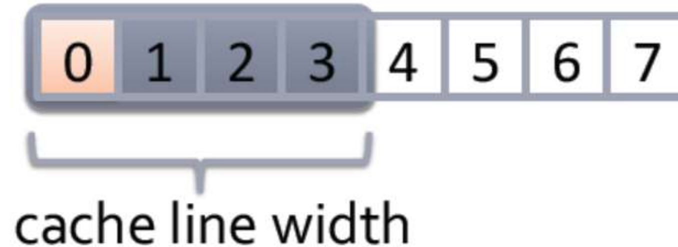
CUDA Memory Hierarchy

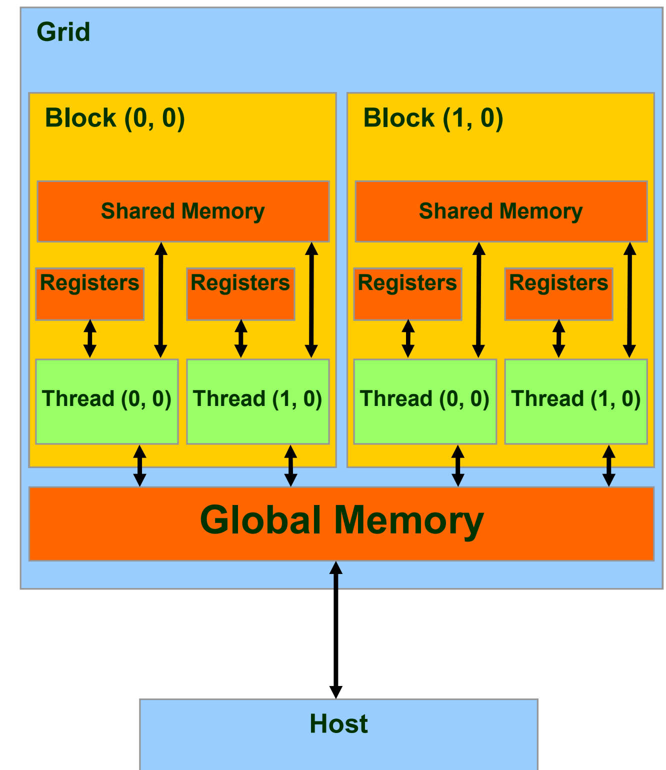
- Each thread has private local memory
- Each thread block has shared memory visible to all threads of the block
- All threads have access to the global memory
- All threads have access to the read-only constant and texture memory



Global Memory

- Main means of data transfer between host and device
- Contents visible to all threads
- *Long* latency access
- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line").





- To use bandwidth effectively, when threads loads, they should:
  - present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

CUDA Device Memory Allocation

cudaMalloc : Allocates object in the device **Global Memory**

```
cudaError_t cudaMalloc(void ** devPtr,
                       size_t size)
```

cudaFree : Frees object from device **Global Memory**

```
cudaError_t cudaFree(void * devPtr)
```

Example:

```
double *x, *y, a, *dx, *dy;
size_t size = N*sizeof(double)
// allocate device memory
cudaMalloc((void **) &dx, size);
cudaMalloc((void **) &dy, size);
// deallocate device memory
cudaFree(dx);
cudaFree(dy);
```

CUDA Device Memory Allocation (cont'd)

cudaMallocPitch : Allocates 2D arrays on the device

```
cudaError_t cudaMallocPitch(void **  devPtr,
                            size_t * pitch,
                            size_t   width,
                            size_t   height)
```

cudaMalloc3D : Allocates 3D arrays on the device

```
cudaError_t cudaMalloc3D(struct cudaPitchedPtr * pitchedDevPtr,
                         struct cudaExtent       extent)
```

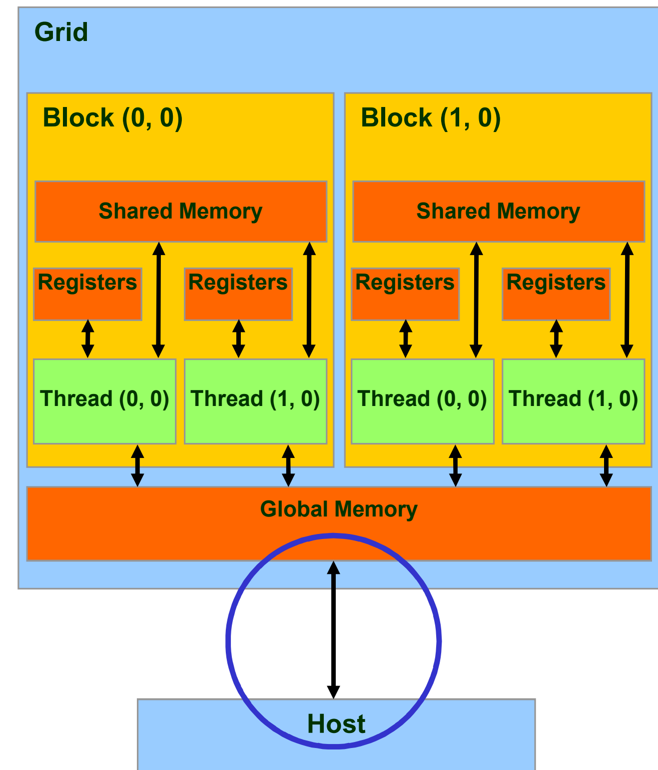
Recommended for allocating 2D and 3D objects. The allocation is appropriately padded to meet the alignment requirement.

CUDA Host-Device Data Transfer

```
cudaError_t cudaMemcpy(void *             dst,
                       const void *       src,
                       size_t             count
                       enum cudaMemcpyKind kind)
```

cudaMemcpy *synchronously* copies count bytes from src to dst

- kind is one of
  - **cudaMemcpyHostToHost**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**
  - **cudaMemcpyDeviceToDevice**
- *Asynchronous* transfer: **cudaMemcpyAsync**
- PCIe 2 x16 peak bandwidth = 8 GB/s



http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

CUDA Host-Device Data Transfer example

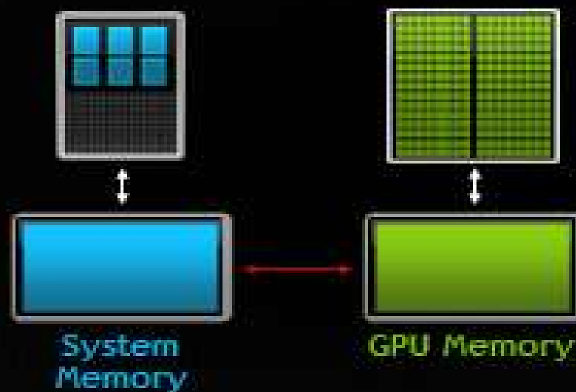
```
#define N 20480
__global__ void daxpy(double a, double *x, double *y)
{  ...  }
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(double);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

Unified Memory in CUDA 6

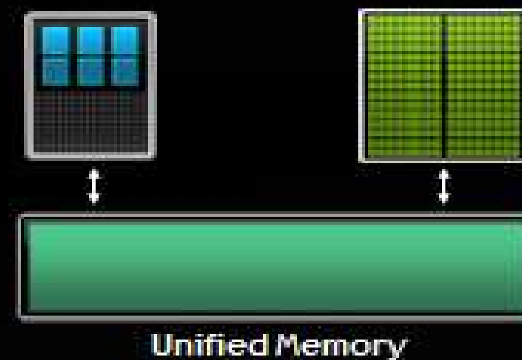
Unified Memory

Dramatically Lower Developer Effort

Developer View Today



Developer View With Unified Memory



Unified Memory

- Both CPUs and GPUs see a single coherent memory image with a common address space
- Unified Memory eliminates the need for explicit data movement via cudaMemcpy*
- cudaMallocManaged allocates memory that will be automatically managed by the Unified Memory system

```
cudaError_t cudaMallocManaged(void ** devPtr,
                                       size_t  size,
                                  unsigned int flags)
```

Without Unified Memory

```
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(double);
    // allocate host memory
    x = (double *)malloc(size);
    y = (double *)malloc(size);
    // assign some values to x and y
    // allocate device memory
    cudaMalloc(&dx, size);
    cudaMalloc(&dy, size);
    // copy from host to device
    cudaMemcpy(dx, x, size,
               cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size,
               cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy from device to host
    cudaMemcpy(y, dy, size,
               cudaMemcpyDeviceToHost);
    // deallocate memory
    cudaFree(dx); cudaFree(dy);
    free(x); free(y);
}
```

With Unified Memory

```
#define N 20480

int main(void) {
    double *x, *y, a;
    size_t size = N*sizeof(double)

    // allocate unified memory
    cudaMallocManaged(&x, size);
    cudaMallocManaged(&y, size);
    // assign some values to x and y

    // launch the kernel function
    daxpy<<<N/256,256>>>(a, x, y);

    cudaDeviceSynchronize();

    for(int i=0; i<N; i++)
      printf("y[%d] = %e\n", i, y[i]);

    // deallocate memory
    cudaFree(x); cudaFree(y);
}
```

Kernel launches are asynchronous

Brief Note on Asynchronous Concurrent Execution

CUDA exposes the following operations as *independent* tasks that *can* operate *concurrently* with one another:

- Computation on the host
- Computation on the device
- Memory transfers from the host to the device
- Memory transfers from the device to the host
- Memory transfers within the memory of a given device
- Memory transfers among devices

CUDA Device Management Functions

- **cudaDeviceSynchronize :** blocks until the device has completed all preceding requested tasks
- **cudaGetDeviceCount :** returns the number of compute-capable devices
- **cudaSetDevice :** sets device to be used for GPU execution
- **cudaGetDevice :** returns which device is currently being used
- **cudaGetDeviceProperties :** returns information about the compute-device
- **cudaDeviceGetAttribute :** returns device attribute value
- **cudaSetDeviceFlags :** sets flags to be used for device executions
- **cudaGetDeviceFlags :** gets the flags for the current device
- **cudaDeviceReset :** destroys and cleans up all resources associated with the current device in the current process
- etc.

  http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html

CUDA Function Type Qualifiers

Function Type Qualifier	Executed on	Callable from
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host & device
__host__ float HostFunc()	host	host

- **__device__** and **__host__** can be used together
- **__global__** defines a *kernel* function (a kernel run N times in parallel by N different CUDA threads); must return **void**
- A call to a **__global__** function is *asynchronous*, which returns before the device has completed its execution

Kernel Execution Configuration

- A kernel function must be called with an execution configuration
- The execution configuration is an expression of the form <<< Dg, Db, Ns, S >>>, where
  - Dg specifies the dimension and size of the grid (of type dim3)
  - Db specifies the dimension and size of each block (of type dim3)
  - Ns specifies the size of shared memory allocated per block (optional)
  - S specifies the the associated stream (optional)

```
__global__ void KernelFunc(...);

KernelFunc<<< Dg, Db, Ns, S >>> (...);
```

CUDA Variable Type Qualifier

	resides in	**lifetime**	**accessible from**
__device__	global memory	application	all threads and hosts
__constant__	constant memory	application	all threads and hosts
__shared__	shared memory	thread block	threads within the block

- A **__device__** variable can be additionally qualified with **__manage__**. Such a variable can be directly referenced from host code (*Unified Memory* access).
- Another way to allocate *Unified Memory* is to use **cudaMallocManaged()**.

Common Runtime Component: Mathematical Functions

- **pow, sqrt, cbrt, hypot**
- **exp, exp2, expm1**
- **log, log2, log10, log1p**
- **sin, cos, tan, asin, acos, atan, atan2**
- **sinh, cosh, tanh, asinh, acosh, atanh**
- **ceil, floor, trunc, round**
- etc.
  - Available in both device and host code
  - Floating-point functions are overloaded for both single and double precisions (IEEE compliant)
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Device Runtime Component: Mathematical Functions

Some mathematical functions (e.g., **sin(x)**) have a less accurate, but faster device-only version (e.g., **__sin(x)**)

- **__pow**
- **__log, __log2, __log10**
- **__exp**
- **__sin, __cos, __tan**

http://docs.nvidia.com/cuda/cuda-math-api/index.html

Host Runtime API

- Provides functions to deal with:
  - Device management (including multi-device systems)
    - e.g., on multi-device system, a host thread can set the device it operates on at any time by calling cudaSetDevice()
  - Memory management
  - Error handling
  - etc.
- There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called

http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Device Runtime Component: Synchronization Functions

- void **__syncthreads**()
- Synchronizes **all** threads in *a block*
- Variants
  - int **__syncthreads_count**(int predicate)
  - int **__syncthreads_and**(int predicate)
  - int **__syncthreads_or**(int predicate)
- Used to coordinate communication between the threads of the same block
- Used to avoid *read-after-write*, *write-after-read*, *write-after-write* hazards when accessing the same addresses in shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

Device Runtime Component: Atomic Functions

- An atomic function performs a *read-modify-write atomic* operation on one 32-bit or 64-bit word residing in global or shared memory
- Atomic functions can only be used in device function
  - int atomicAdd(int* address, int val);
  - float atomicAdd(float* address, float val);
  - etc.

Device Runtime Component: Memory Fence Functions

CUDA assumes a device with a *weakly-ordered* memory model:

- The order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread;
- The order in which a CUDA thread reads data from shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the read instructions appear in the program for instructions that are independent of each other.
- Memory fence functions can be used to enforce some ordering:
  - void **__threadfence_block**()
  - and variants

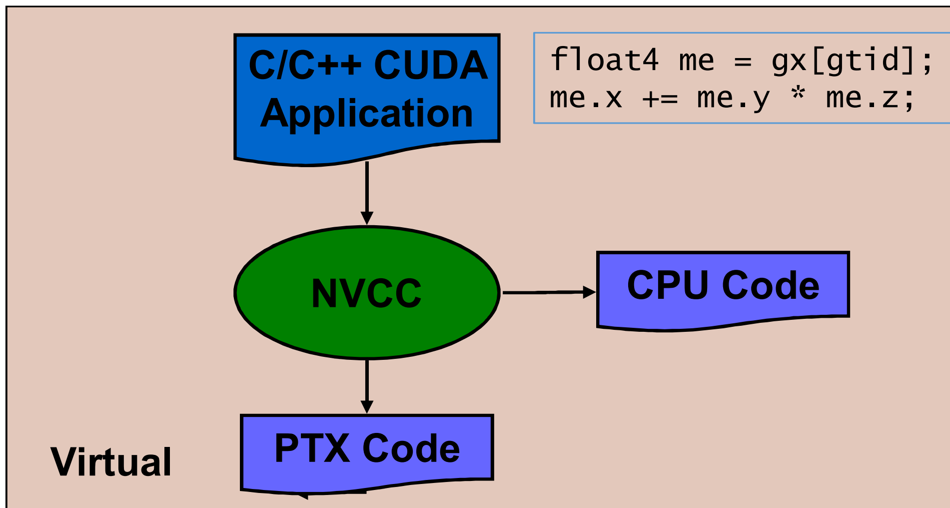
Compilation

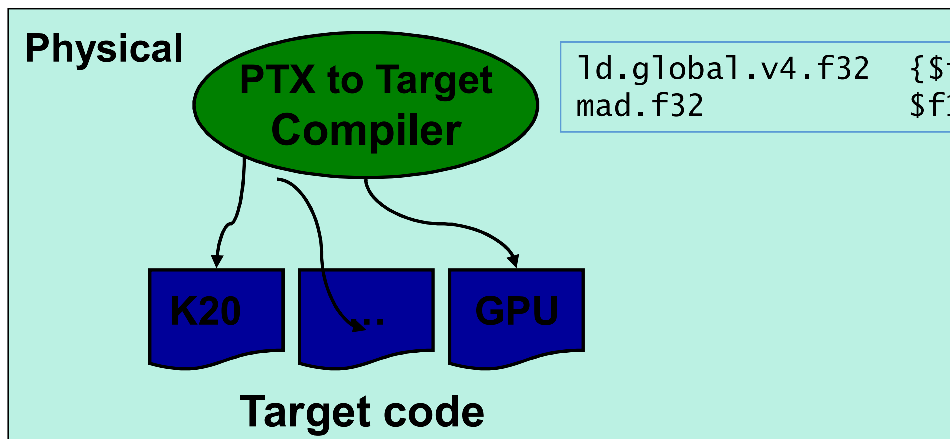
- Any source file (with .cu suffix) containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
- nvcc outputs:
  - **C** code for host, which is then compiled with *gcc* (by default) on Linux
  - **PTX** intermediate assembly code for device, which is
    - Either compiled to object code directly
    - Or interpreted at runtime

For example:

```
$ nvcc daxpy.cu -o daxpy.x
```

Compiling a CUDA Program






```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

Parallel **T**hread e**X**ecution (**PTX**)

- Virtual Machine and ISA
- Programming model
- Execution resources and state

http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

nvcc

- To learn more about the CUDA Compiler Driver **nvcc**,
  - consult *nvcc* documentation: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html
  - or check the online help with `nvcc -h`
- By default, **nvcc** invokes *gcc* for host code compilation on Linux.
- You can use the `-ccbin` option to specify a different compiler for host code compilation. For example, if you prefer Intel C/C++ compiler, use the option `-ccbin icpc`.
- Use the `-Xcompiler` option to specify options directly to the compiler/preprocessor.

nvcc (cont'd)

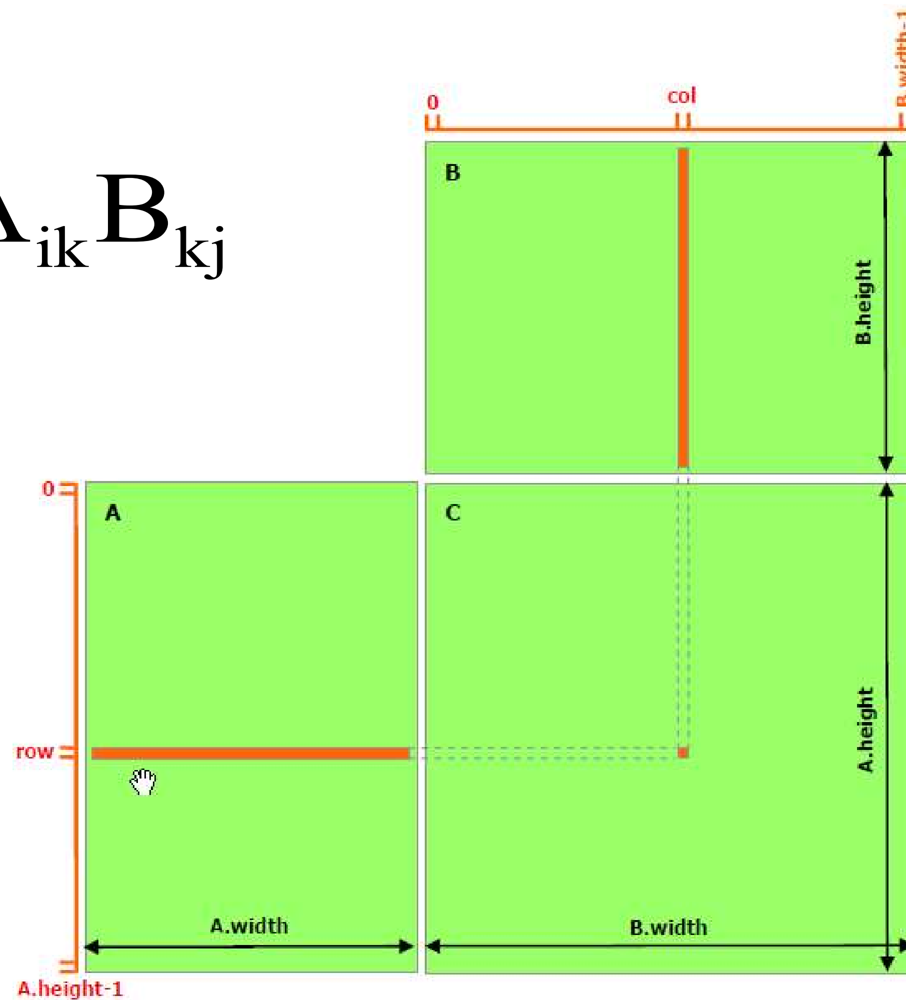
- By default, **nvcc** compiles codes for *Fermi* GPUs (the default is `-arch=compute_20 -code=sm_20,compute_20`)
  - The features of a GPU device depend on its **compute capability**, represented by a version number (*X.Y*)
  - The compute capability of Fermi GPUs is 2.0
  - The compute capability of Tesla K20 is 3.5
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
- To compile codes for Nvidia Tesla K20 (a *Kepler* GPU), use option `-gencode arch=compute_35,code=sm_35`

A better *nvcc* command for *Tesla K20*:

```
$ nvcc -Xcompiler "-O3" \
       -gencode arch=compute_35,code=sm_35 \
       daxpy.cu -o daxpy.x
```

Case Study: Matrix Multiplication

$$C_{ij} = \sum_{k} A_{ik} B_{kj}$$

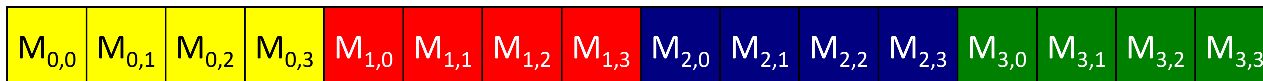


Memory Layout of a Matrix in C

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

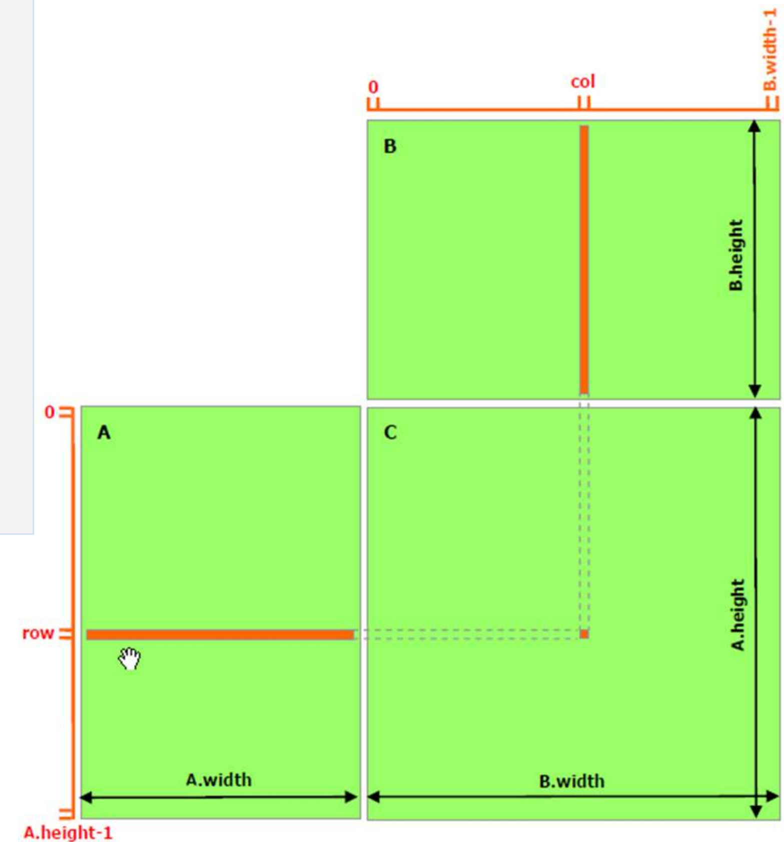
```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

M



Matrix Multiplication: A Simple Host Version in C

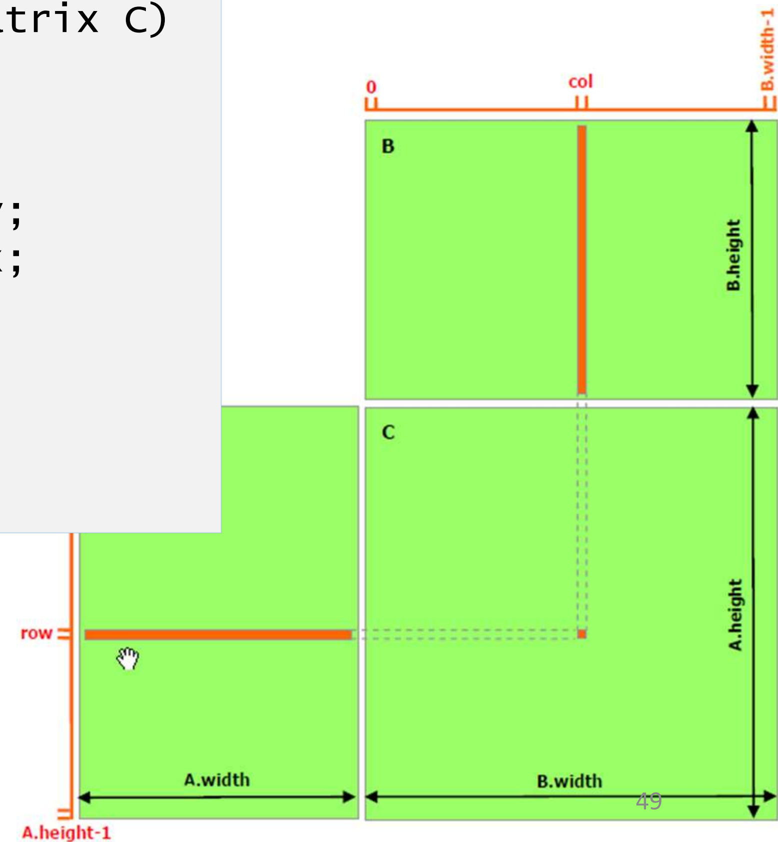
```
// Matrix multiplication on the host
void MatMulOnHost(Matrix A, Matrix B, Matrix C)
{
  for (int row = 0; row < A.height; ++row)
    for (int col = 0; col < B.width; ++col) {
      float Cvalue = 0;
      for (int k = 0; k < A.width; ++k)
        Cvalue += A.elements[row * A.width + k]
                * B.elements[k * B.width + col];
      C.elements[row * C.width + col] = Cvalue;
    }
}
```



Simple, but unoptimized!
Why?

Matrix Multiplication: A Simple CUDA Kernel

```
// Matrix multiplication kernel
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int k = 0; k < A.width; ++k)
        Cvalue += A.elements[i * A.width + k]
                * B.elements[k * B.width + j];
    C.elements[row * C.width + col] = Cvalue;
}
```



Matrix Multiplication: Host Code

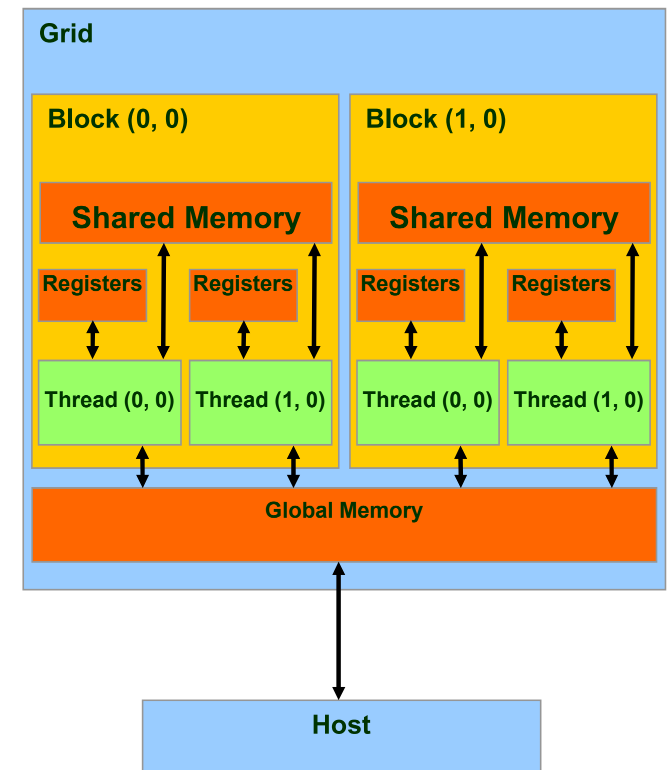
```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    // Matrix d_B (skipped)
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    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);
    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);
}
```

Problems with the simple Implementation

- C = A * B
- Each thread computes one element of C
  - Each thread loads a row of A from global memory (repetitive and very expensive!)
  - Each thread loads a column of B from global memory
  - Each thread performs one multiply and addition for each pair of A and B elements
  - Compute to off-chip memory access ratio close to 1:1 (memory bandwidth limited)
- In total, A is read *B.width* times and B is read *A.height* times from global memory

Recap: CUDA Memory Hierarchy

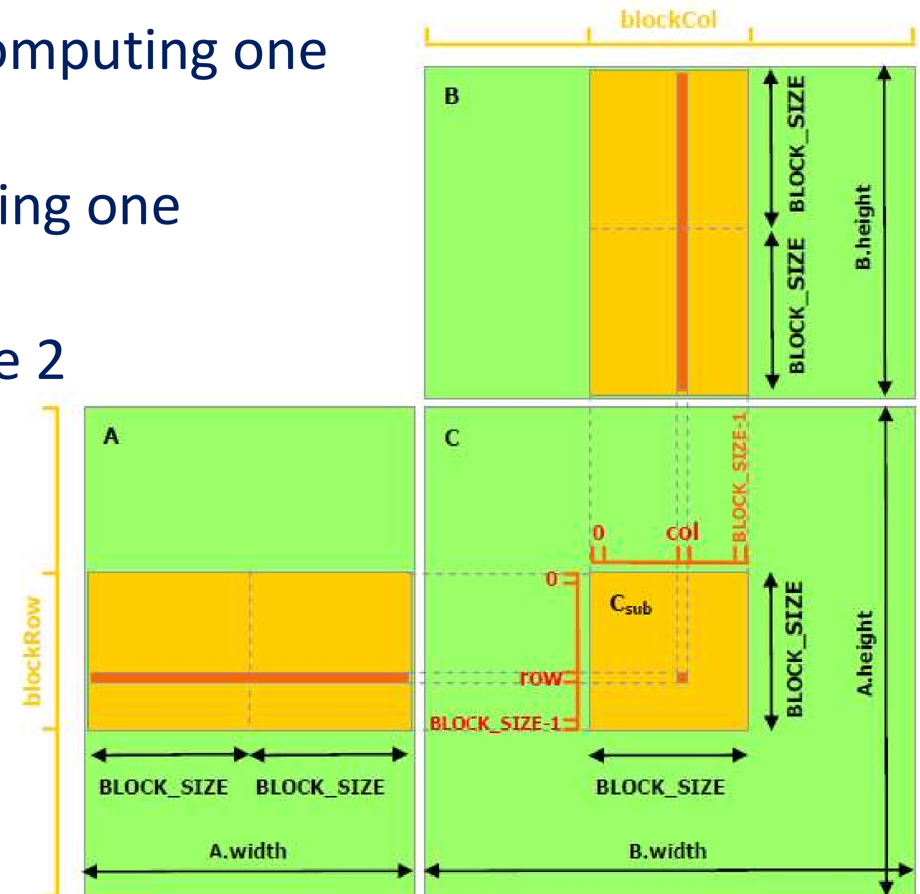
- Each thread has private local memory (fastest)
- Each thread block has shared memory visible to all threads of the block
  - Latency is an order of magnitude lower than global memory
  - Bandwidth is 4x-8x higher than global memory
- All threads have access to the global memory (slowest)
- All threads have access to the read-only constant and texture memory



Matrix Multiplication with Shared Memory

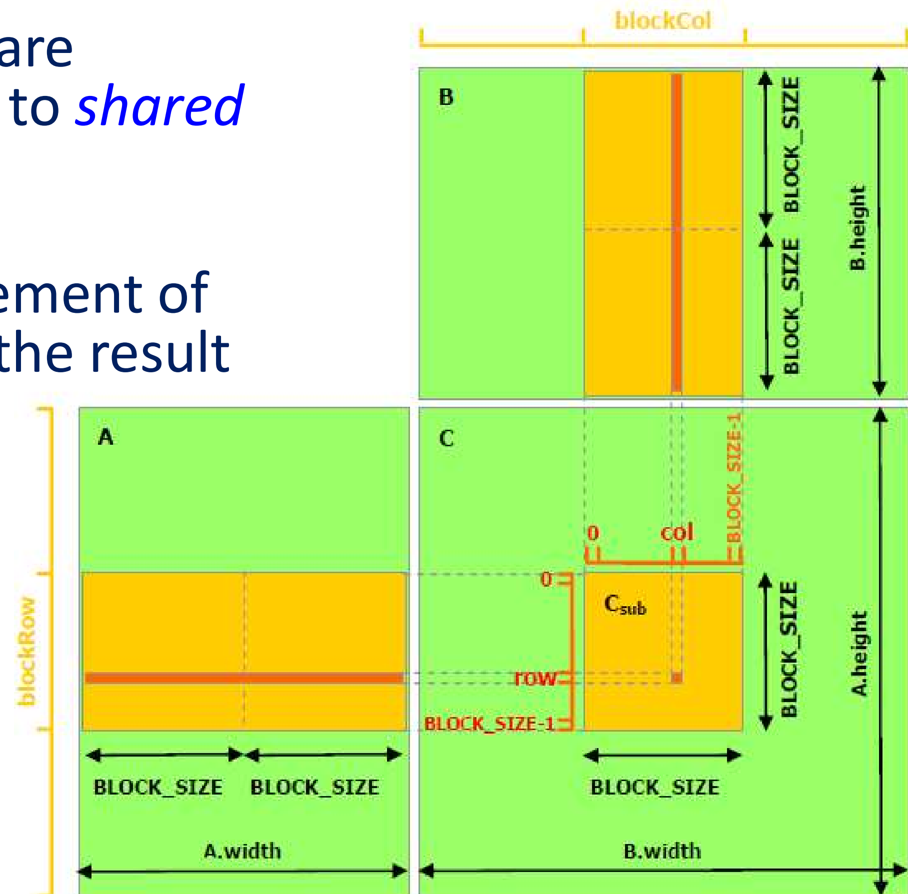
- Each thread block is responsible for computing one square sub-matrix *C_{sub}*
- Each thread is responsible for computing one element of *C_{sub}*
- To fit into the device's resources, these 2 rectangular sub-matrices of A & B are divided into square matrices of dimension *(block_size, block_size)*

$$C_{sub} = A_{(A.width,\ block_size)} * B_{(block_size,\ A.width)} = \Sigma\ A_{(block_size,\ block_size)} * B_{(block_size,\ block_size)}$$



Matrix Multiplication with Shared Memory (cont'd)

1. Load the 2 corresponding square matrices from *global memory* to *shared memory*
   - Each thread loads one element of each matrix
2. Each thread computes one element of the product; and accumulate the result into a *register*
3. Loop
4. Write final result to *global memory*



Revised Matrix Type

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
```

M

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

__device__ functions

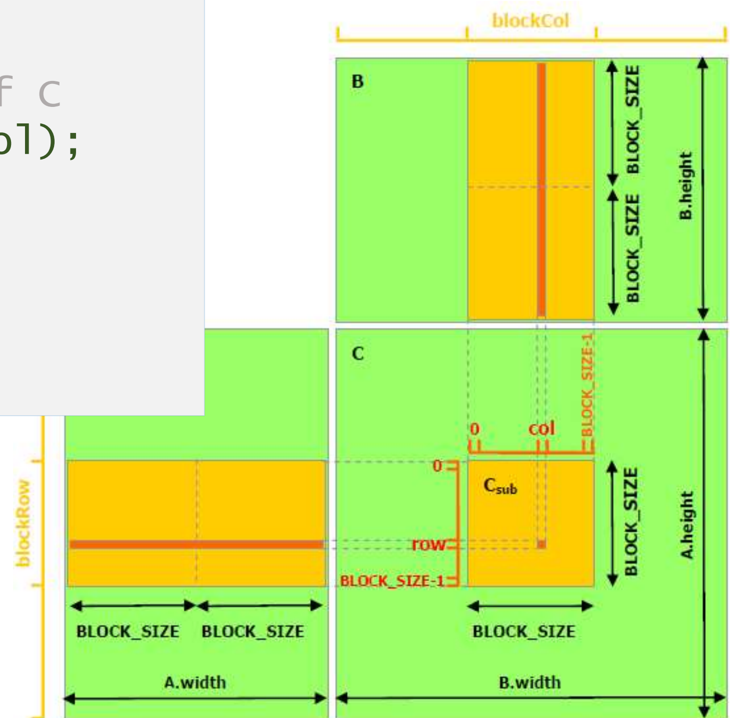
```
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

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

__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;
}
```

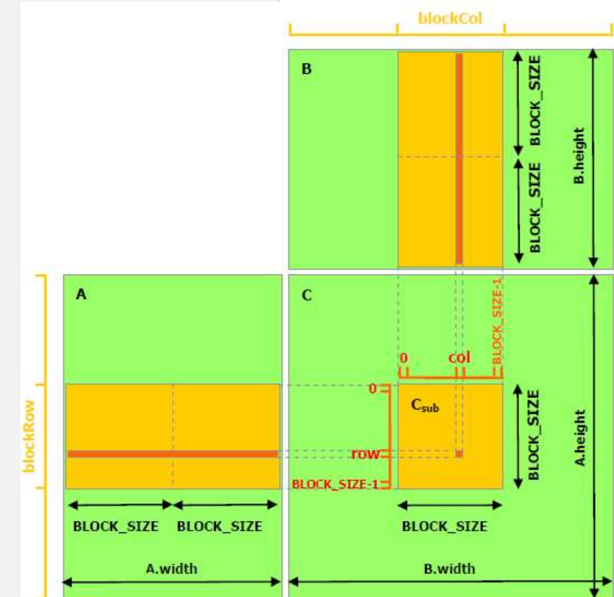
Matrix Multiplication: Revised CUDA Kernel

```
// Matrix multiplication kernel
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    // Each thread computes one elements of Csub
    float Cvalue = 0;
    // Thread row and column with Csub
    int row = threadIdx.y;
    int col = threadIdx.y;
```



Matrix Multiplication: Revised CUDA Kernel (cont'd)

```
    // Loop over sub-matrices of A and B
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrices of A and B
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub
        __shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];
        __shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];
        // Each threads loads one element of each matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        __syncthreads();
    }
    // Write Csub to device memory
    SetElement(Csub, row, col, Cvalue);
}
```



Matrix Multiplication: Revised Host Code

```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    // Matrix d_B (skipped)
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    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);
    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);
}
```

Matrix Multiplication: without vs. with Shared Memory

- Without Shared Memory
  - Each thread loads a row of A a column of B and from global memory. In total, A is read *B.width* times and B is read *A.height* times from global memory
  - Each threads performs one multiply and addition for each pair of A and B elements. Compute to off-chip memory access ratio close to 1:1
- With Shared Memory
  - Each thread loads an element per sub-matrix of A and B and from global memory. In total, A is read (*B.width/block_size*) times and B is read (*A.height/block_size*) times from global memory
  - Each threads performs $block_size^2$ multiply and addition for each pair of A and B elements. Compute to off-chip memory access ratio close to $block_size^2$: 1

Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need thousands of threads for full utilization
- Maximize on-chip work
  - on-chip memory is orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - warp loads and consumes a complete 128-byte cache line

Optimizing GPU Performance

- Understand the GPU architecture
- Understand how applications maps to architecture
- Use lots of threads and blocks
- Often better to redundantly compute in parallel
- Access memory in local regions
- Leverage high memory bandwidth
- Enable global memory coalescing
- Optimize memory copies
- Keep data in GPU device memory
- Experiment and measure

Thrust

- Thrust is a C++ template library for CUDA based on the Standard Template Library (STL)
- Part of CUDA SDK: http://docs.nvidia.com/cuda/thrust/index.html
- Provides STL-like templated interfaces to several algorithms and data structures designed for high performance heterogeneous parallel computing

Algorithms	Data Structures

`thrust::sort`
`thrust::reduce`
`thrust::sort`
Etc.
	`thrust::device_vectore`
	`thrust::host_vector`
	`thrust::device_ptr`
	Etc

- Thrust allows programmers to use other backends, including OpenMP and Intel TBB on multicore machines

  https://github.com/thrust/thrust/wiki/Device-Backends

Thrust DAXPY Example

```
struct saxpy_functor
{
  const double a;
  saxpy_functor(float _a) : a(_a) {}
  __host__ __device__
    double operator()(const double& x, const double& y) const {
      return a * x + y;
    }
};

void saxpy(double A, thrust::device_vector<double>& X,
           thrust::device_vector<double>& Y)
{
  // Y <- A * X + Y
  thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}
```

What about OpenCL?

- OpenCL is a standardized, cross-platform API designed to support **portable** parallel application development on heterogeneous computing systems, including multicore CPUs, GPUs, DSPs, FPGAs and other processors
- OpenCL has a more complex platform and device management model than CUDA
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology

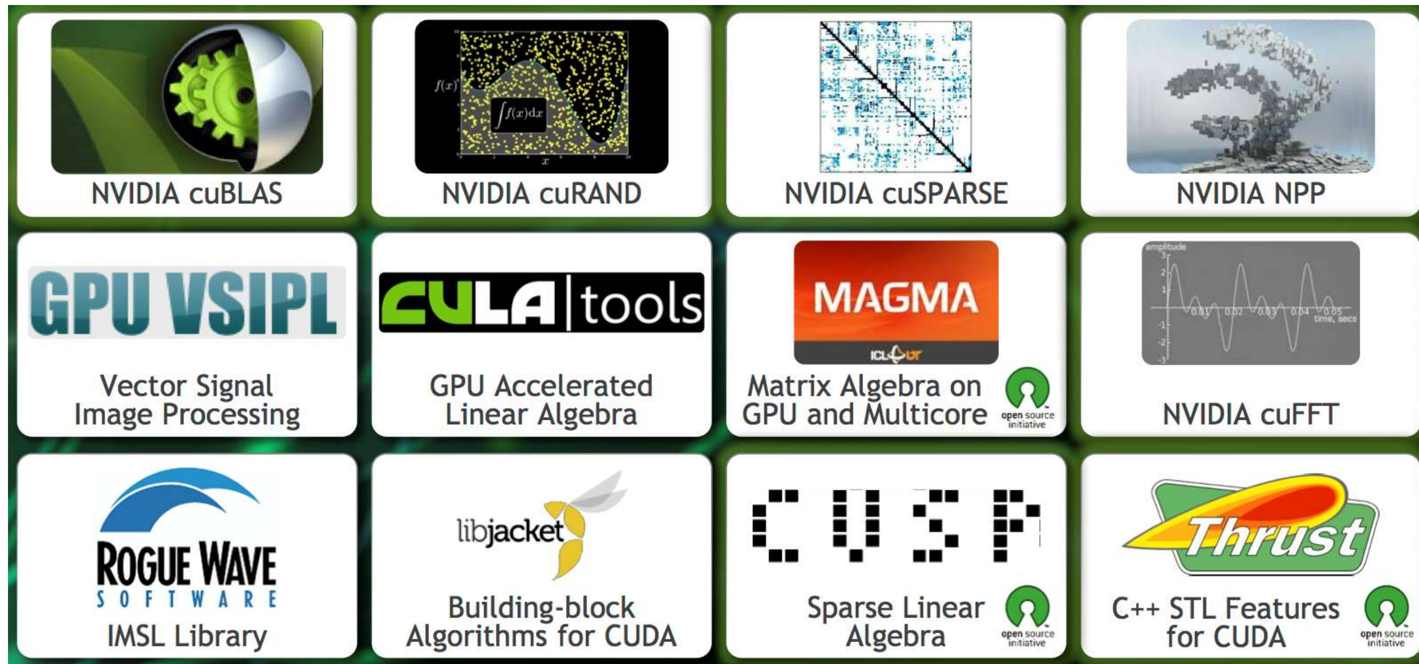
OpenCL parallelism concepts	CUDA equivalent
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work item	Thread
Work group	Block

- OpenCL 1.0 was released in August 2009, 1.1 in 2010, 1.2 in 2011, 2.0 in 2013 and 2.1 in 2015. But Nvidia driver only supports OpenCL 1.1.

3 ways to Accelerate Applications with GPU

1. Programming Languages
    - CUDA C, CUDA Fortran, OpenCL, etc
    - Maximum flexibility
2. GPU-Accelerated Libraries
    - cuBLAS, cuRAND, cuFFT, etc
    - "Drop-in" acceleration
3. OpenACC directives
    - Similar to OpenMP, supporting both C/C++ and Fortran
    - Relatively easy, compared to writing CUDA kernels

GPU-Accelerated Libraries



https://developer.nvidia.com/gpu-accelerated-libraries

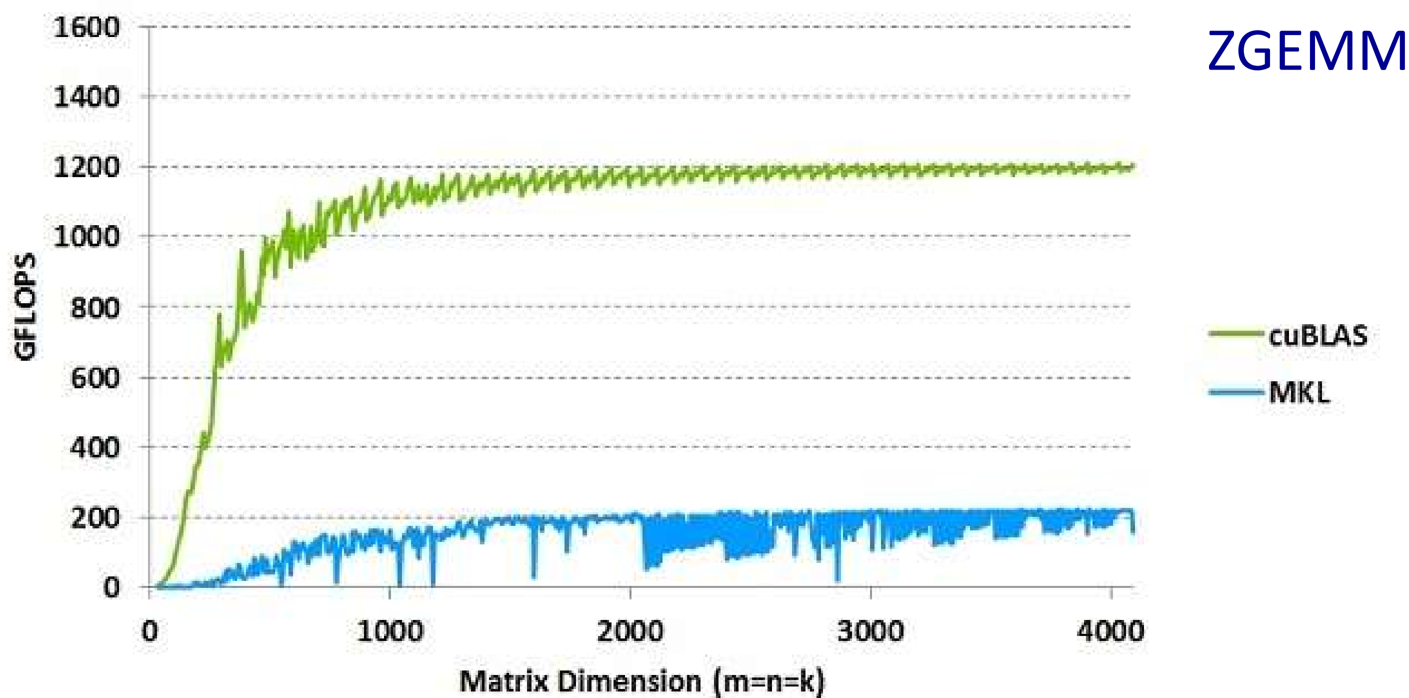
BLAS

- BLAS = Basic Linear Algebra Subprograms
  - Level 1: vector-vector operations that are linear (O(n)) in data and linear (O(n)) in work, e.g., AXPY
  - Level 2: matrix-vector operations that are quadratic ($O(n^2)$) in data and quadratic ($O(n^2)$) in work, e.g., GEMV
  - Level 3: operations that are quadratic ($O(n^2)$) in data and cubic ($O(n^3)$) in work, e.g., GEMM
- The first letter of the subprogram name indicates the precision used:
  - S: Real single precision, e.g., SGEMM
  - D: Real double precision, e.g., DGEMM
  - C: Complex single precision, e.g., CGEMM
  - Z: Complex double precision, e.g., ZGEMM

BLAS Implementations

- CPU
  - **Netlib** reference implementation
  - **ATLAS**
  - **GotoBLAS / GotoBLAS2 / OpenBLAS**
  - **Intel Math Kernel Library (MKL):** *hand-optimized* specifically for Intel processors
- GPU
  - cuBLAS: *not* a drop-in replacement of standard BLAS; one must use the cuBLAS / cuBLAS-XT API
    - http://docs.nvidia.com/cuda/cublas/index.html
  - NVBLAS: *is* a drop-in replacement of standard BLAS; can accelerate most BLAS Level-3 routines
    - http://docs.nvidia.com/cuda/nvblas/index.html

cuBLAS vs MKL



- cuBLAS 6.5 on K40m, ECC ON, input and output data on device
- MKL 11.0.4 on Intel IvyBridge 12-core E5-2697 v2 @2.7GHZ

https://developer.nvidia.com/cublas

GEMM

- GEMM = **Ge**neral **M**atrix-**M**atrix multiplication
- C = α * op(A) * op(B) + β * C
- BLAS level 3

Fortran BLAS:

```
SUBROUTINE SGEMM ( TRANSA, TRANSB, M, N, K, ALPHA,
                   A, LDA, B, LDB, BETA, C, LDC)
    CHARACTER * TRANSA, TRANSB
    INTEGER     M, N, K, LDA, LDB, LDC
    REAL        ALPHA, BETA
    REAL        A(LDA, *), B(LDB, *), C(LDC, *)
```

cuBLAS:

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const float *alpha,
                           const float *A, int lda,
                           const float *B, int ldb,
                           const float *beta,
                           float *C, int ldc)
```

Matrix Multiplication: using cuBLAS

```
cublasStatus_t cublasSgemm(
                    cublasHandle_t handle,
                    cublasOperation_t transa,
                    cublasOperation_t transb,
                    int m, int n, int k,
                    const float *alpha,
                    const float *A, int lda,
                    const float *B, int ldb,
                    const float *beta,
                    float *C, int ldc)
```

```
#include <cuda_runtime.h>
#include <cublas_v2.h>

    const float alpha = 1.0f;
    const float beta  = 0.0f;

    // create the handle
    cublasHandle_t handle;
    cublasCreate(&handle);

    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                m, n, k, &alpha,
                d_B, WB, d_A, WA,
                &beta, d_C, WA);

    // destroy the handle
    cublasDestroy(handle);
```

Typos?

No! Why?

Linking with cuBLAS

```
$ gcc matrixMulCUBLAS.c -l cudart -lcublas \
      -o matrixMulCUBLAS
```

Introduction to OpenACC

- http://www.openacc.org/
- OpenACC (for **Open Acc**elerators) API describes a collection of **compiler directives** to specify loops and regions of code to be offloaded from a host CPU to an attached accelerator
- Supports Fortran and C/C++
- Allows programmers to write high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Allows programmers to start *simple*
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details

History of OpenACC

- Initially collaboration between CAPS Enterprise, Cray Inc., Portland Group (PGI), and NVIDIA
- Built from OpenMP-style directives
  - `#pragma omp parallel` vs. `#pragma acc parallel`
- OpenACC 1.0 specification was released in November 2011 at SuperComputing (SC) 2011
- OpenACC 2.5 was released in October 2015
- Compilers available from Cray, CAPS, and PGI (acquired by Nvidia in 2013)
- Creators of OpenACC are all members of the OpenMP Working Group on accelerators. Potential API merge with OpenMP in the future?

SAXPY using OpenACC & OpenMP

C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp parallel for
#pragma acc kernels
for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
// perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$omp parallel do
!$acc kernels
do i=1,n
    y(i) = a*x(i) + y(i)
enddo
!$acc end kernels
!$omp end parallel do
end subroutine saxpy

...
! perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

Compilation

- On Hyades, We can use PGI compilers to compile OpenACC programs
- Compile the code as a serial program (all directives ignored):

```
$ pgcc daxpy.c -o daxpy.x
$ pgfortran daxpy.f90 -o daxpy.x
```

- Compile the code as an OpenMP program (OpenACC directives ignored):

```
$ pgcc -mp daxpy.c -o daxpy.x
$ pgfortran -mp daxpy.f90 -o daxpy_omp.x
```

- Compile the code as an OpenACC program (OpenMP directives ignored):

```
$ pgcc -acc -Minfo=accel -ta=nvidia,kepler \
      -Mcuda=6.5 daxpy.c -o daxpy_acc.x
$ pgfortran -acc -Minfo=accel -ta=nvidia,kepler \
      -Mcuda=6.5 daxpy.f90 -o daxpy_acc.x
```

OpenACC Directive Syntax

Free-form Fortran	`!$acc directive [clause [[,] clause] ...]`
	`        structured_block`
	`!$acc end directive`

C/C++	`#pragma acc directive [clause [[,] clause] ...] new-line` `        {structured_block}`

- In Fortran fixed-for source files, **!$OMP C$OMP *$OMP** are accepted sentinels and must occupy columns 1-5

OpenACC Directives Overview

`parallel`	starts parallel execution on the accelerator
`kernels`	defines a region that is to be converted to a sequence of kernels for execution on the accelerator
`data`	defines contiguous data to be allocated on the accelerator
`host_data`	makes the address of accelerator data available on the host
`loop`	defines types of parallelism to apply to proceeding loop
`cache`	defines elements or subarrays that should be fetched into cache
`declare`	defines that a variable should be allocated in accelerator memory
`update`	update all or part of host memory from device memory, or vice versa
`wait`	forces program to wait for completion of asynchronous activity

OpenACC Execution Model

- Host-directed execution with an attached accelerator device
  - Compute intensive regions are offloaded to the accelerator device under control of the host
- The device executes
  - **parallel regions**, which typically contain work-sharing loops
  - or **kernels regions**, which typically contain one or more loops which are executed as kernels
- Confused? Then read “OpenACC Kernels and Parallel Constructs” by Michael Wolfe (PGI):

  https://www.pgroup.com/lit/articles/insider/v4n2a1.htm

The kernels Directive

Each loop executed as a *separate* kernel on the GPU.

```
!$acc kernels
    do i=1,n
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
    end do

    do i=1,n
        a(i) = b(i) + c(i)
    end do
!$acc end kernels
```

kernel 1

kernel 2

OpenACC Execution Model on CUDA

- The OpenACC execution model has three levels: **gang**, **worker**, and **vector**
- For GPUs, the mapping is implementation-dependent. Some possibilities:
  - gang==block, worker==warp, and vector==threads of a warp
  - gang==block, vector==threads of a block, with *worker* omitted
- Depends on what the compiler thinks is the best mapping for the problem

OpenACC Clauses Overview

Each directive can have zero or more clauses associated.

Example clauses are:

`if (`*condition*`)`	condition used to determine if command should be executed (data transfer, accelerator computation, etc.)
`async` *[(expression)]*	tells the current command to be executed asynchronously. Used with **wait** for synchronization.

Clauses found in either **kernels** or **parallel** directives:

`reduction (`*op:list*`)`
`private (`*list*`)`
`firstprivate (`*list*`)`

similar to OpenMP

Clauses – parallel and loop

Clauses – **parallel** directive

`num_gangs (e)`	specify the number of gangs to execute in the region
`num_workers (e)`	specify number of workers to launch in each gang
`vector_length (e)`	define vector length to use

Clauses – **loop** directive

`collapse (n)`	specifies # of loops associated
`gang (e)`	distribute across gang
`worker (e)`	distribute across workers (within gang)
`vector (e)`	operate in SIMD (within gang and worker)
`seq`	execute sequentially on the accelerator
`independent`	tell the compiler loops are data-independent

Mapping OpenACC to CUDA threads & blocks

Perhaps 50 blocks, 256 threads per block

```
#pragma acc kernels loop
for (int i = 0; i < n; ++i) y[i] += a*x[i];
```

100 blocks, 128 threads per block

```
#pragma acc kernels loop gang(100), vector(128)
for (int i = 0; i < n; ++i) y[i] += a*x[i];
```

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
  #pragma acc loop gang, vector
  for (int i = 0; i < n; ++i) y[i] += a*x[i];
}
```

Data Clauses

`copy (list)`	Allocates memory on device and copies data from host to device when entering region and copies data to the host when exiting region
`copyin (list)`	Allocates memory on device and copies data from host to device when entering region
`copyout (list)`	Allocates memory on device and copies data to the host when exiting region
`create (list)`	Allocates memory on device but does not copy
`present (list)`	Data is already present on device from another containing data region
`deviceptr (list)`	Pointers in the list are device pointers
`device_resident (list)`	Allocate memory on device instead of host

Data Clauses

`pcopy (list)`	Allocates memory on device and copies data from host to device when entering region and copies data to the host when exiting region
`pcopyin (list)`	Allocates memory on device and copies data from host to device when entering region
`pcopyout (list)`	Allocates memory on device and copies data to the host when exiting region
`pcreate (list)`	Allocates memory on device but does not copy
`present (list)`	Data is already present on device from another containing data region
`deviceptr (list)`	Pointers in the list are device pointers
`device_resident (list)`	Allocate memory on device instead of host

Checks for presence before issuing data command.
pcopy is equivalent to **present_or_copy**, etc.

Clauses – host_data and update

Clauses – **host_data** directive

`use_device (list)`	make the device address data available in host code

Clauses – **update** directive

`host (list)`	variables to copy from device to host
`device (list)`	variables to copy from host to device

Combing Clauses

Similar to OpenMP, we can combine directives

- `#pragma acc parallel loop` *`[clause [[,] clause] ...]`*
- `#pragma acc kernels loop` *`[clause [[,] clause] ...]`*

A loop must directly follow, similar to `parallel for` in OpenMP

OpenACC Runtime Routines

Header file:

Fortran	`use openacc`
	or
	`#include "openacc_lib.h"`

C/C++	`#include "openacc.h"`

List of runtime routines:

```
int acc_get_num_devices (acc_device_t);
```

gets number of devices of passed type

```
int acc_set_device_type (acc_device_t);
```

sets device type to use

```
acc_device_t acc_get_device_type ();
```

gets current device type

```
void acc_set_device_num (int, acc_device_t);
```

sets device based on index and type

```
void acc_get_device_num (acc_device_t);
```

gets current device number

Runtime Routines – Synchronization

```
int acc_async_test (int);
```

tests to see if a specified asynchronous tasks are completed

```
int acc_async_test_all ();
```

tests to see if all asynchronous tasks are completed

```
void acc_async_wait (int);
```

waits until specified asynchronous task is completed

```
void acc_async_wait_all ();
```

waits until all asynchronous tasks are completed

Runtime Routines – Setup and Teardown

```
void acc_init (acc_device_t);
```

initializes OpenACC runtime for passed device type

```
void acc_shutdown (acc_device_t);
```

shuts down connection to passed device type

```
int acc_on_device (acc_device_t);
```

tells the program whether it’s executing on passed device type

```
void* acc_malloc (size_t);
```

allocates memory on the device

```
void acc_free (void*);
```

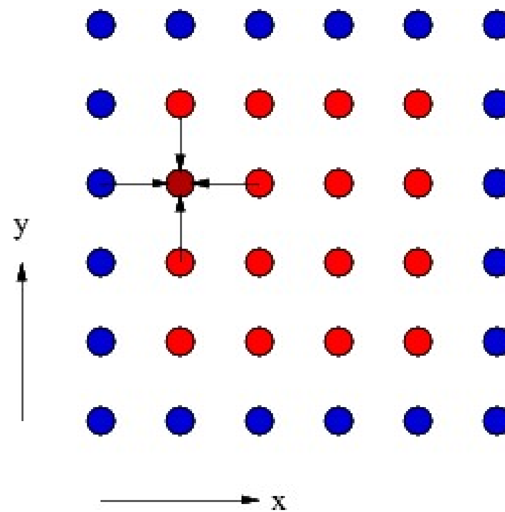
rees memory on the device

Matrix Multiplication: with OpenACC

```
  // compute matrix multiplication
#pragma acc kernels copyin(a,b) copy(c)
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      for (k = 0; k < SIZE; ++k) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

Case Study: Jacobi Iteration

Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i, j) = \frac{A_{k+1}(i-1, j) + A_{k+1}(i+1, j) + A_{k+1}(i, j-1) + A_{k+1}(i, j-1)}{4}$$

Jacobi Iteration: C Code

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

  for ( int j = 1; j < n-1; j++ ) {
    for ( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }

  for ( int j = 1; j < n-1; j++) {
    for ( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Jacobi Iteration: OpenMP C Code

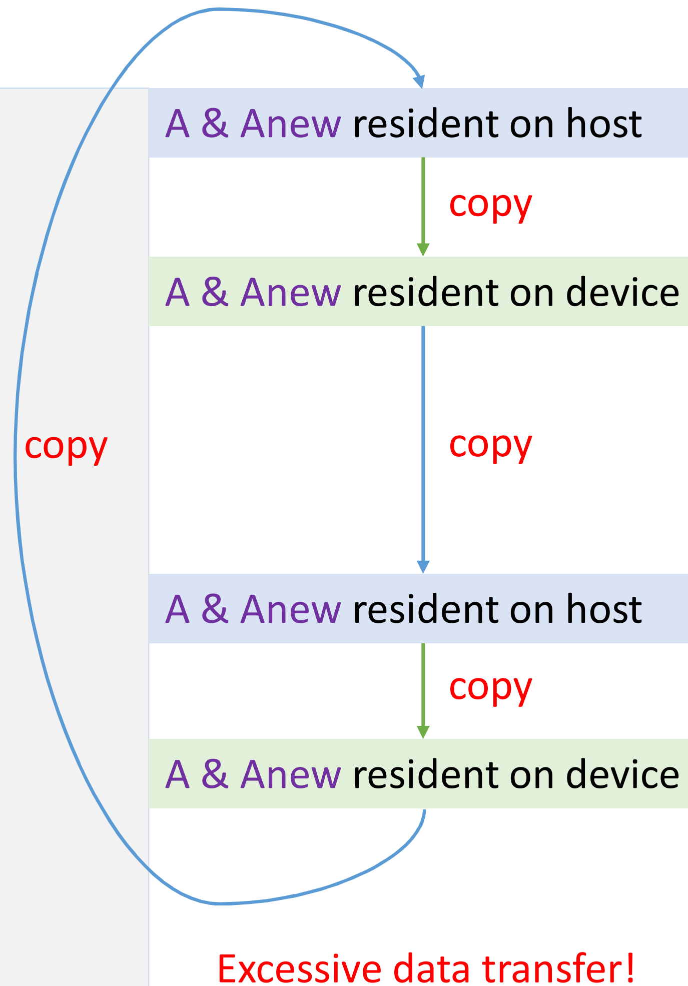
```
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
  for ( int j = 1; j < n-1; j++ ) {
    for ( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for ( int j = 1; j < n-1; j++ ) {
    for ( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Jacobi Iteration: OpenACC C v1

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc kernels loop reduction(max:err)
  for ( int j = 1; j < n-1; j++ ) {
    for ( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for ( int j = 1; j < n-1; j++) {
    for ( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

What went wrong?

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc kernels loop reduction(max:err)
  for ( int j = 1; j < n-1; j++ ) {
    for ( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for ( int j = 1; j < n-1; j++) {
    for ( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```



Excessive data transfer!

Jacobi Iteration: OpenACC C v2

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Jacobi Iteration: OpenACC C v3

```
#pragma acc data copy(A), copyin(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for(int i = 1; i < m-1; i++)
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
  }
#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc kernels loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = 0.25 * (Anew[j][i+1] + Anew[j][i-1] +
                        Anew[j-1][i] + Anew[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]);
    }
  }
  iter+=2;
}
```

specify # of gangs/vectors (i.e., grid/block dimension)

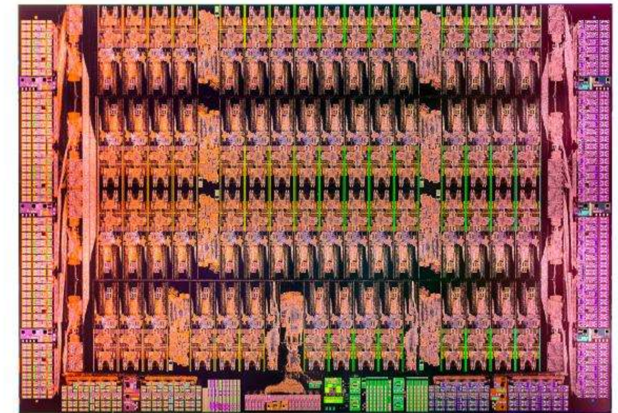
replace memcpy kernel with a second instance of the stencil kernel

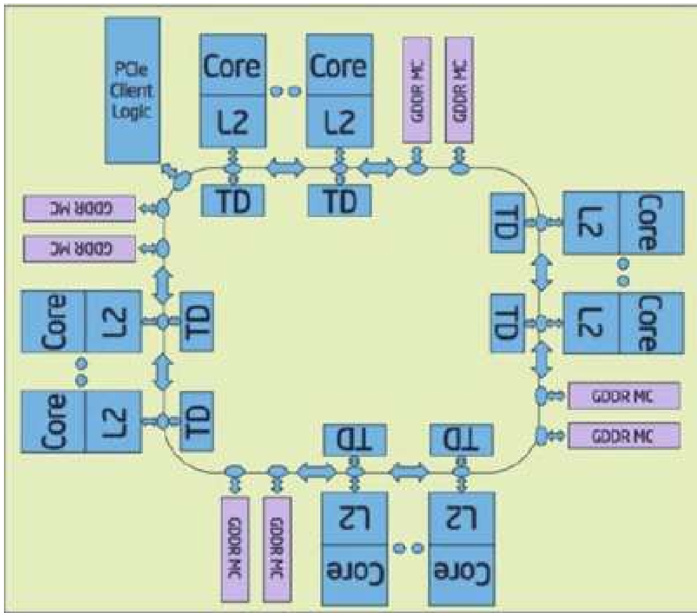
Introduction to Intel Xeon Phi

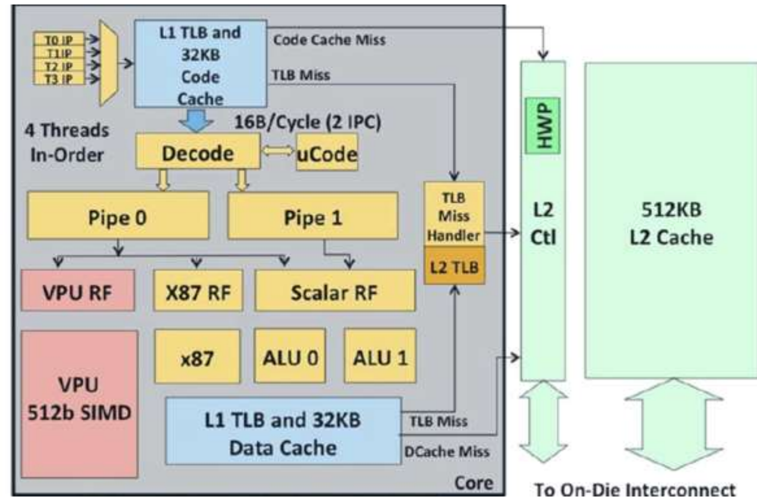
- Intel Xeon Phi is the brand name for Intel's manycore coprocessors
- Many Integrated Core (MIC) Architecture
  - Developed from earlier work on **Larrabee** (a cancelled GPGPU chip)
  - **Knights Ferry**, Intel's MIC prototype board, was announced in May 2010
  - **Knights Corner**, Intel's 1st MIC commercial product line, was announced in 2011
  - **Knights Landing**, the 2nd generation MIC product, was announced in June 2013
  - **Knights Hill**, the codename for the 3rd generation MIC architecture, was announced at SC14
- Xeon Phi 31S1P (Knights Corner) coprocessors power the world's fastest supercomputer – the Chinese **Tianhe-2** system
- In Hyades, there is a MIC node, *Aesyle*, with two Xeon Phi 5110P coprocessors

Intel Xeon Phi 5110P

- Knights Corner architecture
- 22nm process
- 60 cores (in-order, dual-issue x86 design)
- 4 threads per core
- Core speed: 1.053 GHz
- 512-bit AVX
- Double precision performance: 1.01 TFLOPS
  = 1.053 (GHz) x 60 (cores) x 512/64 x 2 (FMA)
- Memory: 8GB GDDR5
  5 (GT/s) x 16 (channels) x 4 (B) = 320 GB/s
- PCI express 2.0 x16
  500 (MB/s) x 8/10 x 16 = 8 GB/s (16 GB/s duplex)







http://www.tomshardware.com/reviews/xeon-phi-larrabee-stampede-hpc,3342-3.html

Xeon Phi runs Linux!

```
[dong@aesyle ~]$ ssh mic0

dong@aesyle-mic0:~$ uname -a
Linux aesyle-mic0.ucsc.edu 2.6.38.8+mpss3.4.1 #1 SMP
Fri Oct 17 16:05:10 PDT 2014 k1om GNU/Linux

dong@aesyle-mic0:~$ grep processor /proc/cpuinfo
processor   : 0
...
processor   : 239

dong@aesyle-mic0:~$ cat /proc/meminfo
MemTotal:         7882352 kB
```

Xeon Phi runs Linux! (cont'd)

```
dong@aesyle-mic0:~$ df -h
Filesystem                Size      Used Available Use% Mounted
on
none                      6.4G    154.2M      6.2G   2% /
none                      3.8G     44.0K      3.8G   0% /dev
10.7.7.1:/export/home
                         35.5T      6.2T     29.3T  17% /home
10.8.8.142@o2ib:10.8.8.143@o2ib:/pfs
                        145.4T    126.5T     18.9T  87% /pfs
```

On Hyades, the shared file systems, NFS & Lustre, are mounted on the Xeon Phi coprocessors, as well as on the hosts

MIC Execution Modes

- Native Mode

  Applications run directly on the Xeon Phi coprocessors

- Symmetric Mode

  Applications run on both the host processors and the Xeon Phi coprocessors at the same time

- Offload Mode

  An application starts execution on the host; as the computation proceeds it offloads part or all of the computation from its processes or threads to the coprocessors.

Native Execution Mode

- The same serial, OpenMP, and MPI programs can be compiled to run on Xeon Phi, *without* any modification of the source code
- However, the code must be cross-compiled for the k1om architecture
- x86 or x86-64 binaries can't run on Xeon Phi

Native Serial Program

- Cross-compiler the serial program for Xeon Phi on the host, using the Intel Compilers:

```
$ icc -mmic serial.c -o serial.k1om
$ ifort -mmic serial.f90 -o serial.k1om
```

- If you prefer GCC:

```
$ /usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gcc \
      serial.c -o serial.k1om
```

- On Hyades, the same shared file systems are mounted on Xeon Phi, so you can run the binary directly with:

```
$ ssh mic0 /pfs/dong/serial.k1om
```

- Otherwise, you'll have to upload the binary from the host to Xeon Phi first, using scp/sftp

Native OpenMP Program

- Cross-compiler the OpenMP program for Xeon Phi on the host, using Intel Compilers:

```
$ icc -mmic -openmp omp.c \
      -o omp.k1om
$ ifort -mmic -openmp omp.f90 \
        -o omp.k1om
```

- Run the OpenMP executable natively on Xeon Phi with:

```
$ ssh mic0 OMP_NUM_THREADS=60 \
       /pfs/dong/omp.k1om
```

Native MPI Program

- Cross-compiler the MPI program for Xeon Phi on the host, using Intel Compilers and Intel MPI:

```
$ mpiicc -mmic mpi.c -o mpi.k1om
$ mpiifort -mmic mpi.f90 \
        -o mpi.k1om
```

- Run an MPI session of 60 processes natively on Xeon Phi with:

```
$ ssh mic0 mpirun -n 60 \
       /pfs/dong/mpi.k1om
```

Symmetric Execution Mode

- Compiler the MPI code for both the x86-64 and the x1om architectures, using Intel Compilers and Intel MPI:

```
$ mpiicc mpi.c -o mpi.x86-64
$ mpiicc -mmic mpi.c -o mpi.k1om
```

- Run an MPI program on all the processor and coprocessor cores:

```
$ mpirun \
  -n 12 -host aesyle /pfs/dong/mpi.x86-64 : \
  -n 60 -host mic0 /pfs/dong/mpi.k1om : \
  -n 60 -host mic1 /pfs/dong/mpi.k1om
```

- Load-balance will be a serious issue!

Offload Execution Mode

- Compiler Assisted Offload (CAO)
  - Explicit

    Using directives (Intel-specific or OpenMP 4.0); explicitly directing data movement and code execution
  - Implicit

    Using Shared Virtual Memory (Intel Cilk Plus)
- Automatic Offload (AO)
  - Computational intensive calls to Intel Math Kernel Library (MKL)
  - MKL automatically manage details
  - More than offload: work division between host and MIC

Further Readings

- **The Evolution of GPUs for General Purpose Computing**, *by* Ian Buck (Nvidia): http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf
- **CUDA Toolkit Documentation:** http://docs.nvidia.com/cuda/
- **CUDA C Programming Guide:** http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- **Professional CUDA C Programming**, *by* Cheng, Grossman, & McKercher, Wrox, 2014

  ebook available at UCSC library
- **PGI CUDA Fortran Programming Guide and Reference:** https://www.pgroup.com/doc/pgicudafortug.pdf
- **CUDA Fortran for Scientists and Engineers**, *by* Fatica & Ruetsch, Elsevier 2014 http://www.sciencedirect.com/science/book/9780124169708

Further Readings (cont'd)

- **OpenACC 1.0 Specification:** http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf
- **OpenACC 1.0 Quick Reference Guide:** http://www.openacc.org/sites/default/files/OpenACC_API_QuickRefGuide.pdf
- **OpenACC 2.5 Specification:** http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf
- **OpenACC 2.5 Quick Reference Guide:** http://live-openacc.pantheonsite.io/sites/default/files/OpenACC_2.5_ref_guide.pdf
- **PGI Accelerator Compilers with OpenACC Getting Started Guide:** https://www.pgroup.com/doc/openacc_gs.pdf
- **OpenACC Programming and Best Practices Guide:** http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf

Further Readings (cont'd)

- **Intel Xeon Phi Coprocessor High-Performance Programming**, *by* Jeffers & Reinders, Morgan Kaufmann, 2013

  ebook available at UCSC library
- **Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers**, *by* Rezaur Rahman, Apress, 2013
  http://www.apress.com/9781430259268
- **Fortran vs. C offload directives and functions:**
  https://software.intel.com/en-us/articles/fortran-vs-c-offload-directives-and-functions
- **OpenMP Specifications:**
  http://openmp.org/wp/openmp-specifications/
- **OpenMP 4.0 Specifications:**
  http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- **OpenMP 4.0.2 Examples:**
  http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf