Programming GPUs with CUDA - Day 1

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

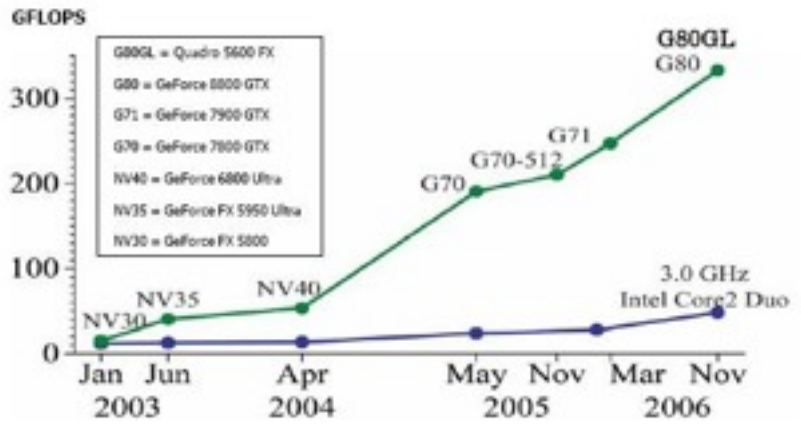
Overview

- Introduction to GPU programming
- Introduction to CUDA
- CUDA example programs
- CUDA libraries
- OpenACC
- CUDA extensions to the C programming language
- Beyond the basics - optimizing CUDA

#1 system on TOP500 list - Titan

Oak Ridge National Labs - operational in October 2012
18,688 Opteron 16-core CPUs
18,688 NVIDIA Tesla K20 **GPUs**
17.6 peta FLOPS

GPU computing timeline



before 2003 - Calculations on GPU, using graphics API
2003 - Brook “C with streams”
2005 - Steady increase in CPU clock speed comes to a halt, switch to multicore chips to compensate. At the same time, computational power of GPUs increases
November, 2006 - CUDA released by NVIDIA
November, 2006 - CTM (Close to Metal) from ATI
December 2007 - Succeeded by AMD Stream SDK
December, 2008 - Technical specification for OpenCL1.0 released
April, 2009 - First OpenCL 1.0 GPU drivers released by NVIDIA
August, 2009 - Mac OS X 10.6 Snow Leopard released, with OpenCL 1.0 included
September 2009 - Public release of OpenCL by NVIDIA
December 2009 - AMD release of ATI Stream SDK 2.0 with OpenCL support
March 2010 - CUDA 3.0 released, incorporating OpenCL
May 2011 - CUDA 4.0 released, better multi-GPU support
mid-2012 - CUDA 5.0
late-2012 - NVIDIA K20 Kepler cards
Future - CPUs will have so many cores they will start to be treated as GPUs?
Accelerators become universal?

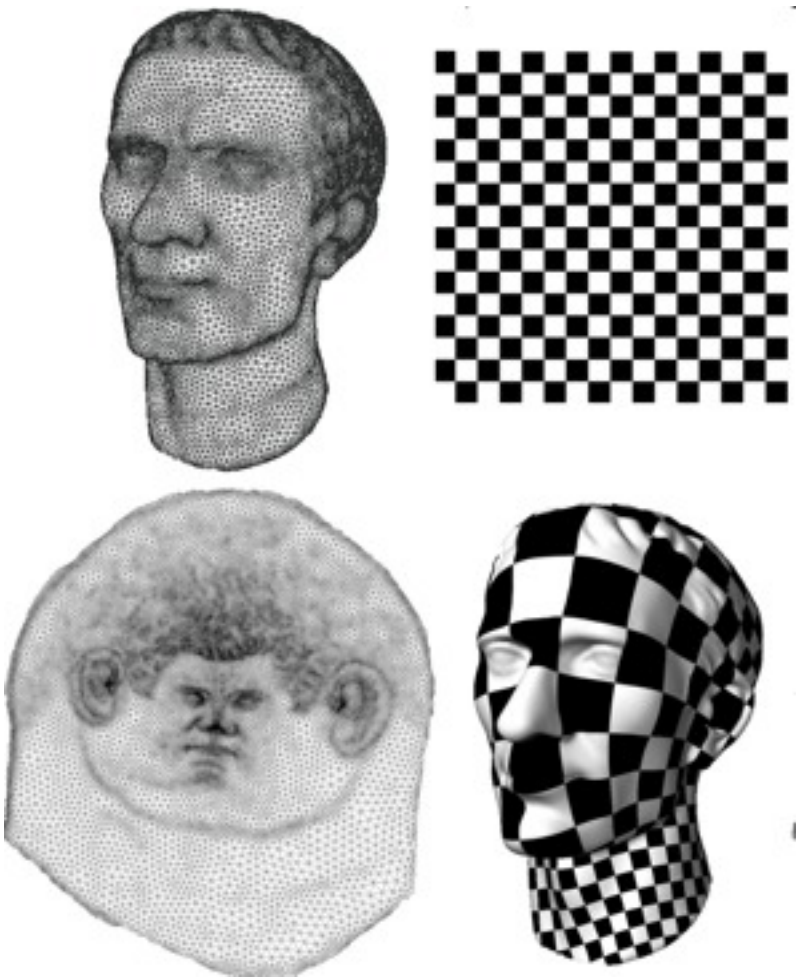
Introduction to GPU programming

- A graphics processing unit (GPU) is a processor whose main job is to accelerate the rendering of 3D graphics primitives

- Modern GPUs feature a programmable graphics pipeline, which means that it is possible to write custom programs that execute on the GPU to process graphics data
  - these are called *shaders*, because the main task of GPU programs is to apply lighting to 3D scenes

A brief tour of graphics programming

- 2D textures are wrapped around 3D meshes to assign colour to individual pixels on screen
- Lighting and shadow are applied to bring out 3D features
- Shaders allow programmers to define custom shadow and lighting techniques
  - can also combine multiple textures in interesting ways
- Resulting pixels get sent to a framebuffer for display on the monitor



Introduction to GPGPU

- With the introduction of programmable graphics pipelines, people realized that a GPU can be used for more than graphics
  - ***G***eneral ***P***urpose computing on ***G***raphics ***P***rocessing ***U***nits
- We could disguise some arbitrary data as “textures”, upload and execute an arbitrary program as a “shader”, then have the results sent to an off-screen “framebuffer”
- Programs could be created using shading languages such as GLSL, Cg and HLSL
- Communication with the device could be managed using a graphics API like OpenGL or Direct3D

Introduction to GPGPU (cont.)

- This was cool for a while, but because all computations occurred within the graphics pipeline, there were limitations:
  - limited inputs/outputs
  - limited data types, graphics-specific semantics
  - memory and processor optimized for short vectors, 2D textures
  - lack of communication or synchronization between “threads”
  - no writes to random memory locations (scatter)
  - no in-out textures (had to ping-pong in multi-pass algorithms)
  - graphics API overhead
  - graphics API learning curve

Beyond the Graphics APIs

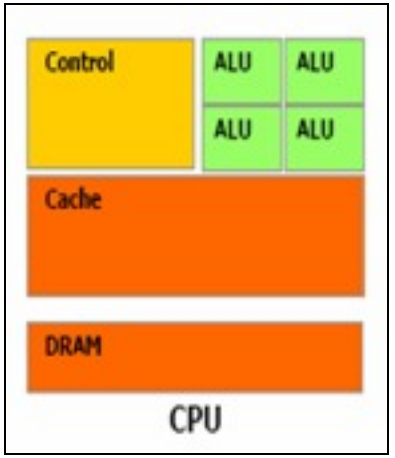
- NVIDIA now offers ***CUDA*** while AMD has moved toward ***OpenCL*** (although there are lower-level APIs available for AMD)
- These new computing platforms bypass the graphics pipeline and expose the raw computational capabilities of the hardware
- Modern GPU architectures provide flexible addressing modes (gather/scatter), shared memory and thread synchronization primatives
- Important competitor is ***OpenCL***
  - a vendor-agnostic computing platform
  - supports vendor-specific extensions akin to OpenGL
  - goal is to support a range of hardware architectures including GPUs, CPUs, Cell processors, Larrabee and DSPs using a standard low-level API
- **OpenACC** compiler directive approach is emerging as an alternative (works somewhat like OpenMP)

The appeal of GPGPU

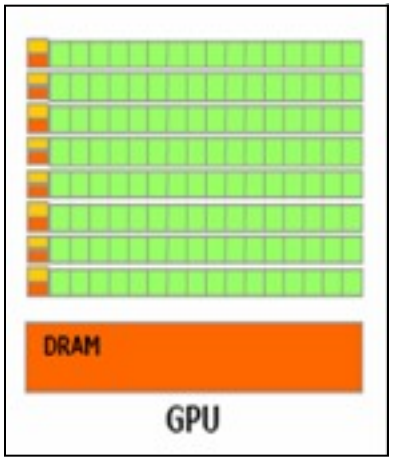
- "Supercomputing for the masses"
  - significant computational horsepower at an attractive price point
  - readily accessible hardware
- Scalability
  - programs can execute without modification on a run-of-the-mill PC with a $150 graphics card or a dedicated multi-card supercomputer worth thousands of dollars
- Bright future – the computational capability of GPUs doubles each year
  - more thread processors, faster clocks, faster DRAM, …
  - "GPUs are getting faster, faster"

Comparing GPUs and CPUs

- CPU
  - "Jack of all trades"
  - task parallelism (diverse tasks)
  - minimize latency
  - multithreaded
  - some SIMD

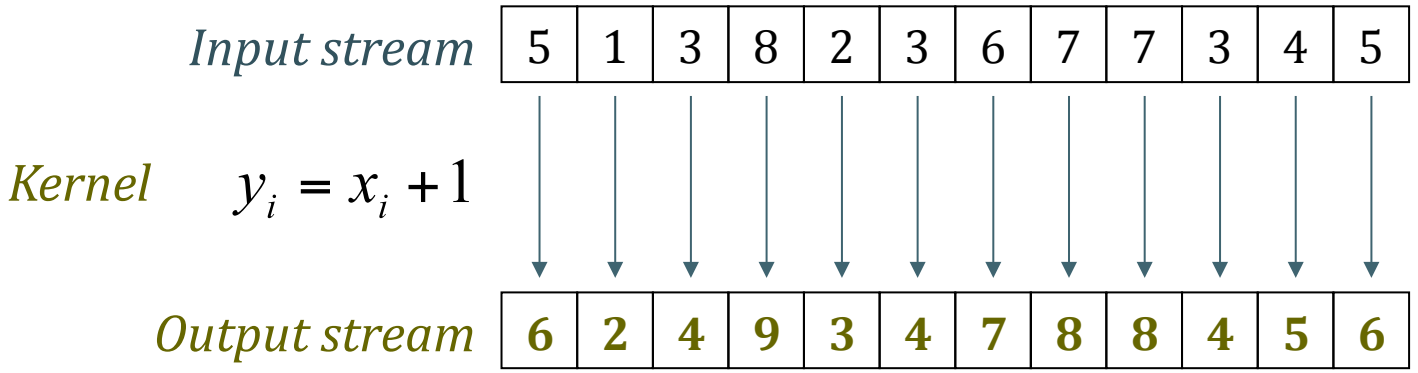


- GPU
  - excel at number crunching
  - data parallelism (single task)
  - maximize throughput
  - super-threaded
  - large-scale SIMD



Stream computing

- A parallel processing model where a computational ***kernel*** is applied to a set of data (a ***stream***)
  - the kernel is applied to stream elements in parallel



- GPUs excel at this thanks to a large number of processing units and a parallel architecture

Beyond stream computing

- Current GPUs offer functionality that goes beyond mere stream computing
- Shared memory and thread synchronization primitives eliminate the need for data independence
- Gather and scatter operations allow kernels to read and write data at arbitrary locations

CUDA

- "Compute Unified Device Architecture
- A platform that exposes NVIDIA GPUs as general purpose *compute devices*
- Is CUDA considered GPGPU?
  - yes and no
    - CUDA can execute on devices with no graphics output capabilities (the NVIDIA Tesla product line) – these are not "GPUs", per se
    - however, if you are using CUDA to run some generic algorithms on your graphics card, you are indeed performing some ***G***eneral ***P***urpose computation on your ***G***raphics ***P***rocessing ***U***nit…

What is CUDA used for?

- CUDA has been used in many different areas
  - options pricing in finance
  - electromagnetic simulations
  - fluid dynamics
  - GIS
  - geophysical data processing
  - 3D visualization solutions
  - …

- See http://www.nvidia.com/object/cuda_home_new.htm
  - long list of projects and speedups achieved

Speedup

- What kind of speedup can I expect?
  - 0x – 2000x reported
  - 10x – considered typical (vs. multi-CPU machines)
  - >= 30x considered worthwhile

- Speedup depends on
  - problem structure
    - need many identical independent calculations
    - preferably sequential memory access
  - level of intimacy with hardware
  - time investment

CUDA programming model

- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor*** capable of executing a large number of lightweight threads in parallel
- Computation on the device is performed by ***kernels***, functions executed in parallel on each data element
- Both the host and the device have their own ***memory***
  - the host and device cannot directly access each other's memory, but data can be transferred using the runtime API
- The host manages all memory allocations on the device, data transfers, and the invocation of kernels on the device

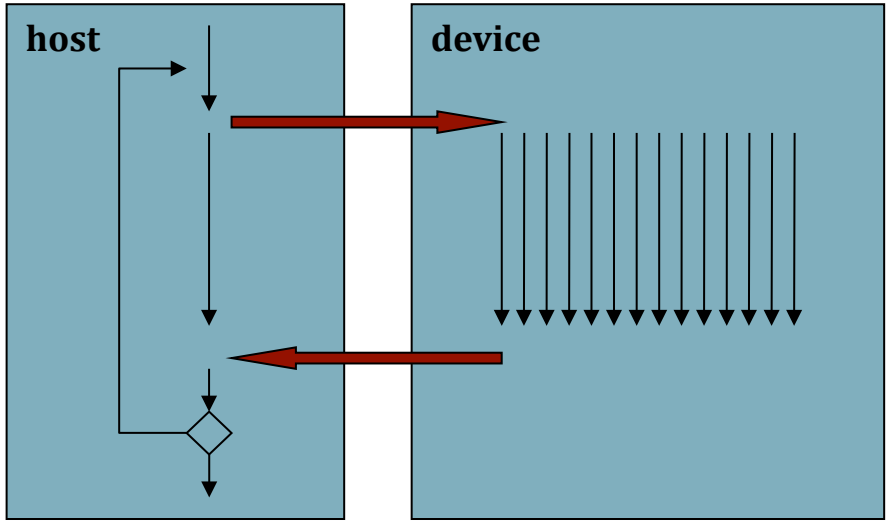
GPU applications

- The GPU can be utilized in different capacities

- One is to use the GPU as a massively parallel coprocessor for number crunching applications
  - upload data and kernel to GPU
  - execute kernel
  - download results
  - CPU and GPU can execute asynchronously

- Some applications use the GPU for both data crunching and visualization
  - CUDA has bindings for OpenGL and Direct3D

GPU as coprocessor

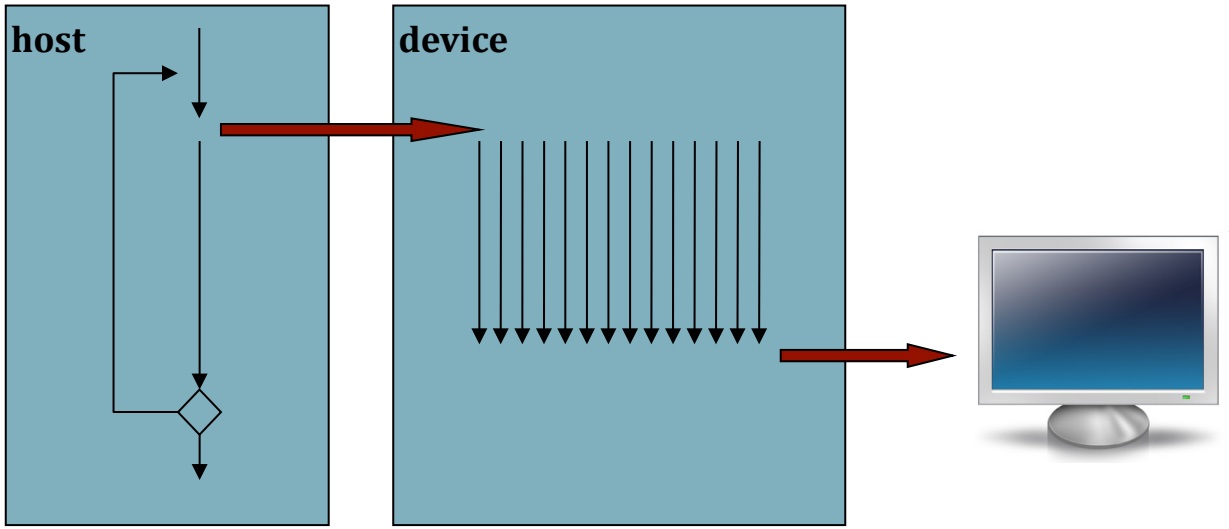


Kernel execution is asynchronous

Asynchronous memory transfers also available

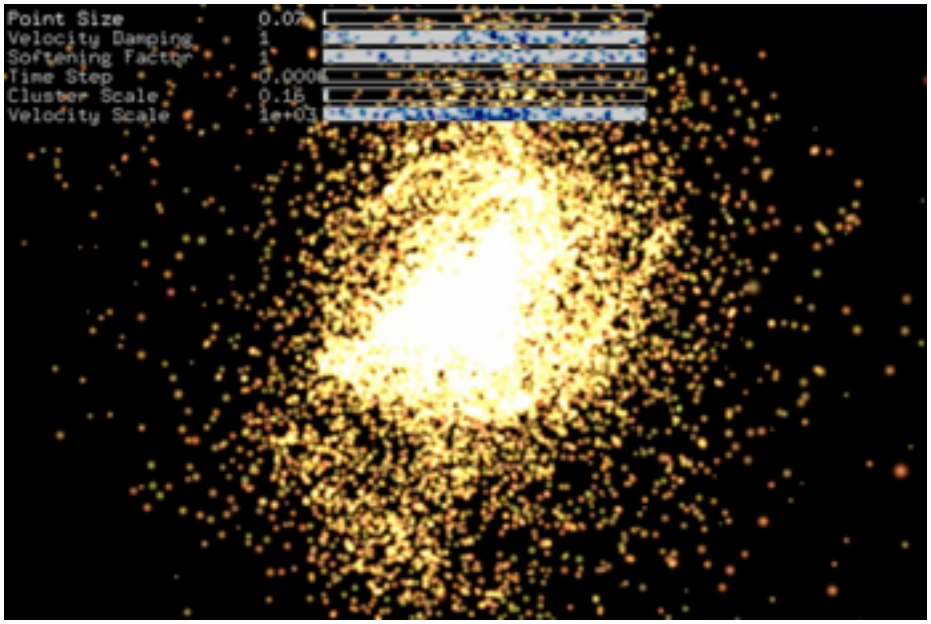
- Basic paradigm
  - host uploads inputs to device
  - host remains busy while device performs computation
    - prepare next batch of data, process previous results, etc.
  - host downloads results
- Can be iterative or multi-stage

Simulation + visualization



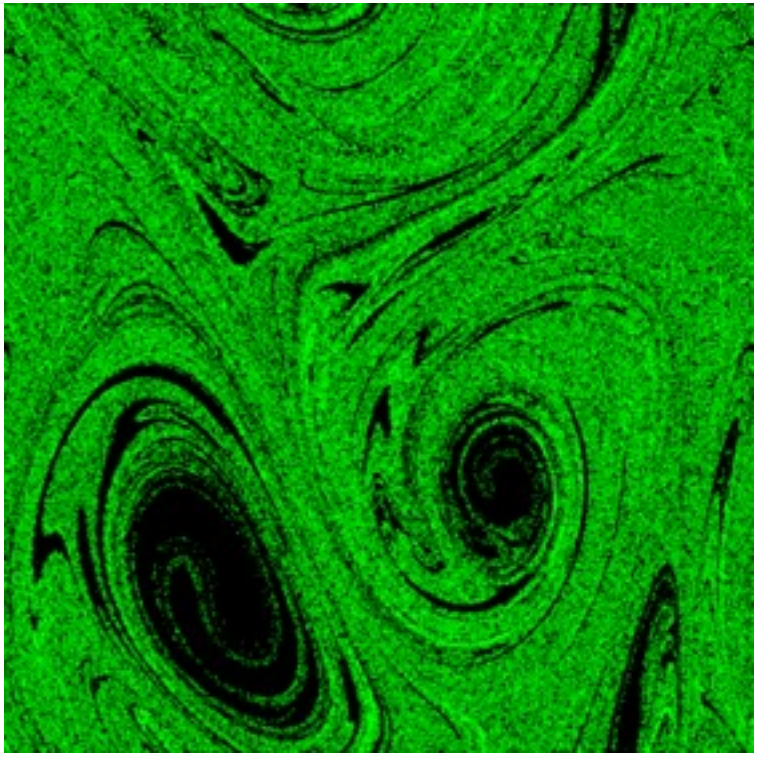
- Basic paradigm
  - host uploads inputs to device
  - host may remain busy while device performs computation
    - prepare next batch of data, etc.
  - results used on device for rendering, no download to host

Simulation + visualization (cont.)



N-Body Demo

Fluids Demo



SHARCNET GPU systems

- Aways check our software page for latest info! See also:
  `https://www.sharcnet.ca/help/index.php/GPU_Accelerated_Computing`

- *angel.sharcnet.ca*
  - 11 NVIDIA Tesla S1070 GPU servers
    - each with 4 GPUs + 16GB of global memory
    - each GPU server connected to ***two*** compute nodes (2 4-core Xeon CPUs + 8GB RAM each)
    - 1 GPU per quad-core CPU; 1:1 memory ratio between GPUs/CPUs

- *visualization workstations with a variate of GPU cards*

2012 arrival - “monk” cluster

- 54 nodes, InfiniBand interconnect, 80 Tb storage
- Node:
  8 x CPU cores (Intel Xeon 2.26 GHz)
  48 GB memory
  2 x M2070 GPU cards
- Nvidia Tesla M2070 GPU
  “Fermi” architecture
  ECC memory protection
  L1 and L2 caches
  2.0 Compute Capability
  448 CUDA cores
  515 Gigaflops (DP)

CUDA versions installed

- Different versions of CUDA available - choose one via modules

– on monk latest CUDA installed in `/opt/sharcnet/cuda/5.0.35/`

– sample projects in `/opt/sharcnet/cuda/5.0.35/samples`
  - copy to your work space (e.g. `/work/username/cuda_sdk`) & compile following instructions on the software page

    https://www.sharcnet.ca/help/index.php/CUDA

Development node: mon54 for interactive use, plus viz stations

Output of device diagnostic program

```
   ...

   [ppomorsk@mon54:~/CUDA_day1/device_diagnostic] ./device_diagnostic.x
found 2 CUDA devices
   --- General Information for device 0 ---
Name:  Tesla M2070
Compute capability:  2.0
Clock rate:  1147000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  5636554752
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  14
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)

   --- General Information for device 1 ---
Name:  Tesla M2070
...
```

Submitting GPU jobs

- See GPU Accelerated Computing article on training wiki for maximum detail

– note: queue details (mpi vs. gpu – test queue oddities)

– To submit a job to gpu queue on angel

```
sqsub -q qpu --gpp=1 -n 1 -o out.txt -r 5m ./a.out
```

Thread batching

- To take advantage of the multiple multiprocessors, kernels are executed as a ***grid*** of ***threaded blocks***

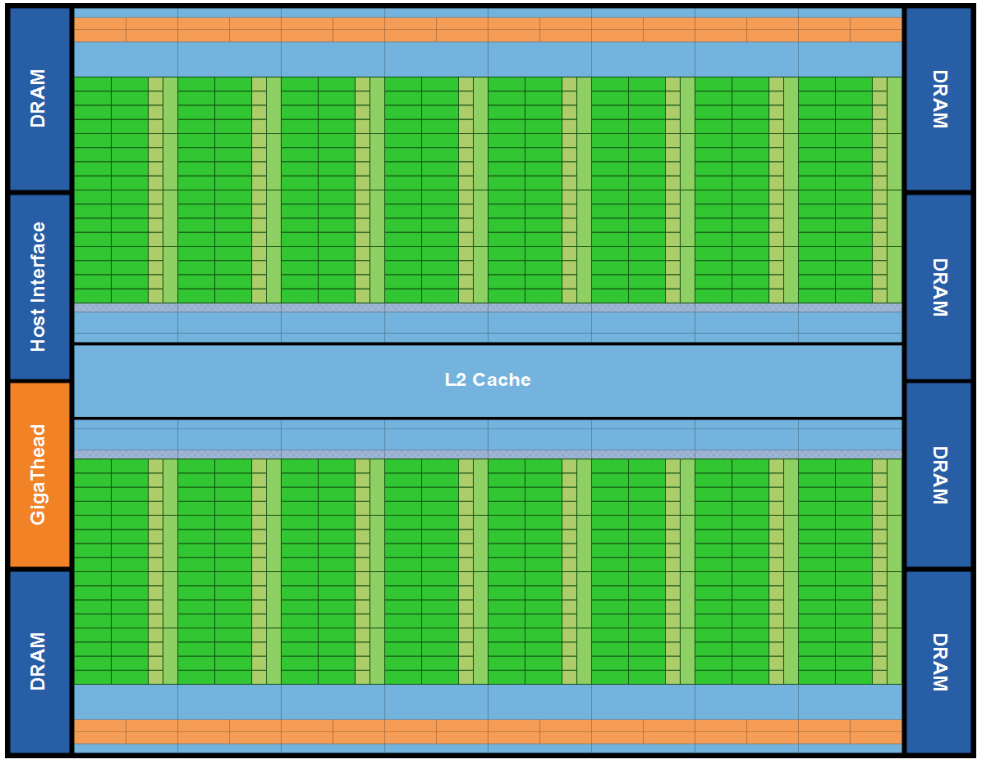
- All threads in a thread block are executed by a single multiprocessor

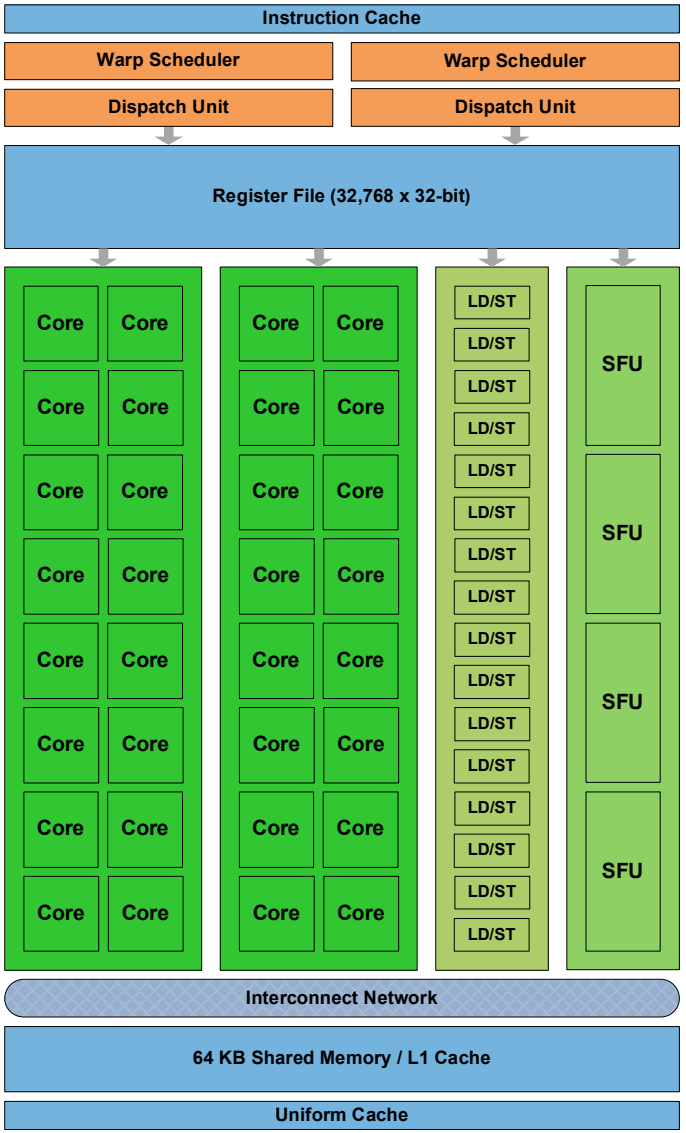
- The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.)
  - this has several important implications that will be discussed later

Hardware basics

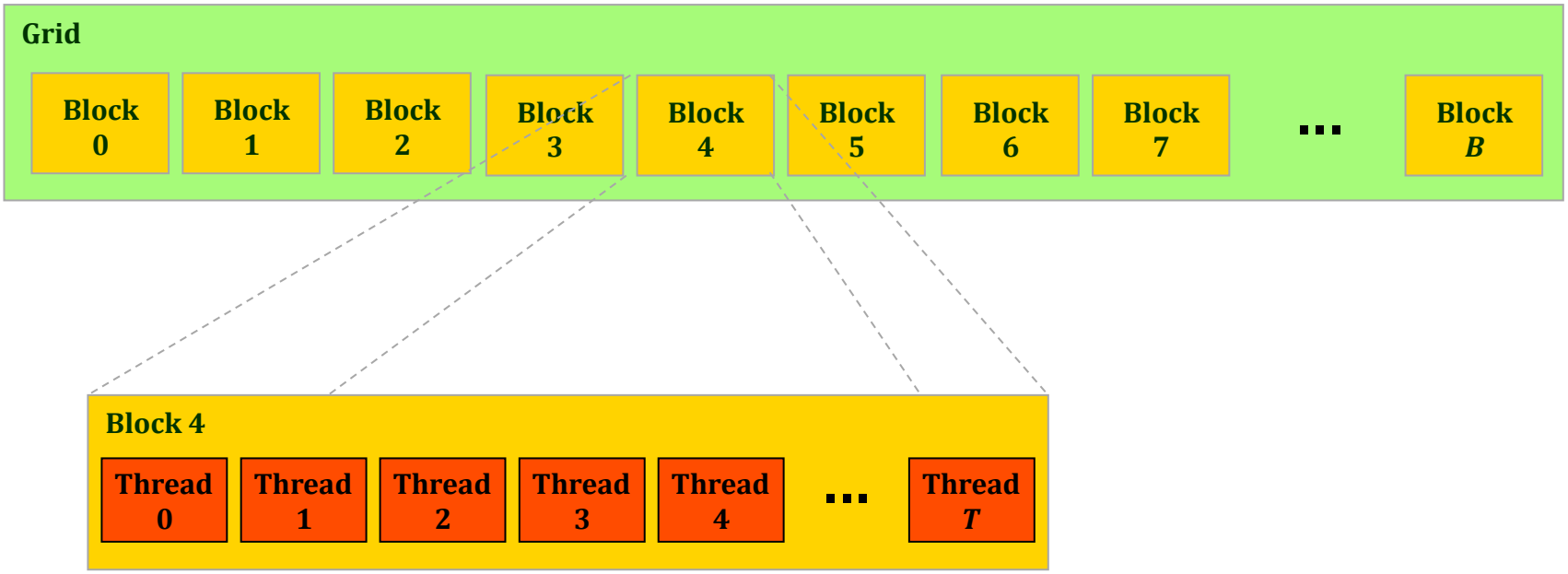
- The compute device is composed of a number of ***multiprocessors***, each of which contains a number of SIMD processors
  - Tesla M2070 has 14 multiprocessors (each with 32 CUDA cores)
- A multiprocessor can execute K ***threads*** in parallel physically, where K is called the ***warp size***
  - ***thread*** = instance of kernel
  - warp size on current hardware is 32 threads
- Each multiprocessor contains a large number of 32-bit ***registers*** which are divided among the active threads

GPU Hardware architecture - NVIDIA Fermi

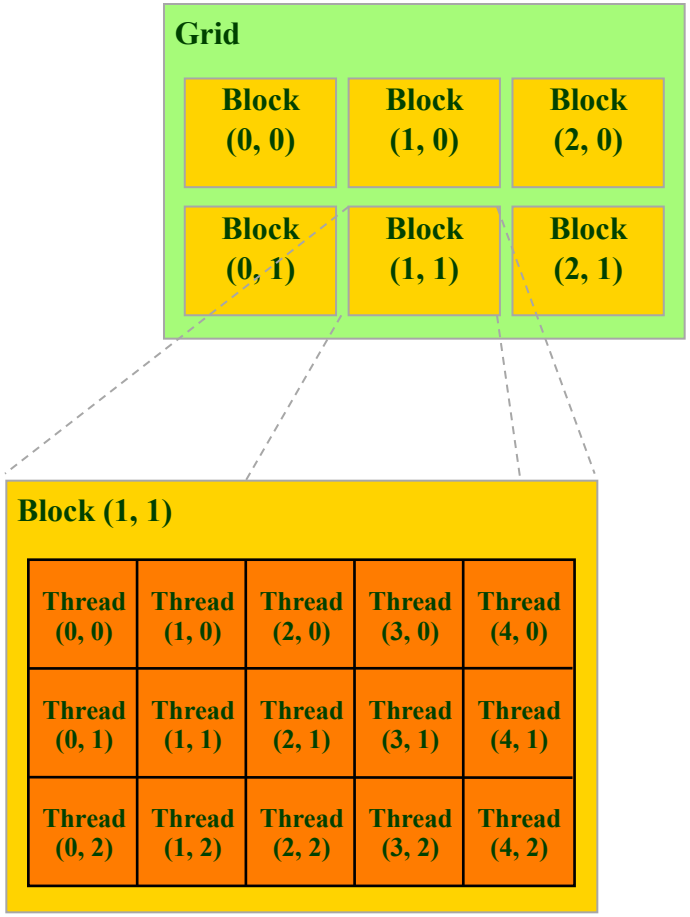




Thread batching: 1D example



Thread batching: 2D example



Thread batching (cont.)

- At runtime, a thread can determine the block that it belongs to, the block dimensions, and the thread index within the block

- These values can be used to compute indices into input and output arrays

Introduction to GPU Programming: CUDA

HELLO, CUDA!

Language and compiler

- CUDA provides a set of extensions to the C programming language
  - new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in `.cu` files
  - host and device code and coexist in the same file
  - storage qualifiers determine type of code
- Compiled to object files using `nvcc` compiler
  - object files contain executable host and device code
- Can be linked with object files generated by other C/C++ compilers

SAXPY

- SAXPY (Scalar Alpha X Plus Y) is a common linear algebra operation. It is a combination of scalar multiplication and vector addition:

$$y = \alpha \cdot x + y$$

  - x and y are vectors, α is a scalar
  - x and y can be arbitrarily large

SAXPY: CPU version

- Here is SAXPY in vanilla C:

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

  - the CPU processes vector components sequentially using a for loop
  - note that `vecY` is an in-out parameter here

SAXPY: CUDA version

- CUDA kernel function implementing SAXPY

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The **`__global__`** qualifier identifies this function as a kernel that executes on the device

SAXPY: CUDA version (cont.)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- **`blockIdx`**, **`blockDim`** and **`threadIdx`** are built-in variables that uniquely identify a thread's position in the execution environment
  - they are used to compute an offset into the data array

SAXPY: CUDA version (cont.)

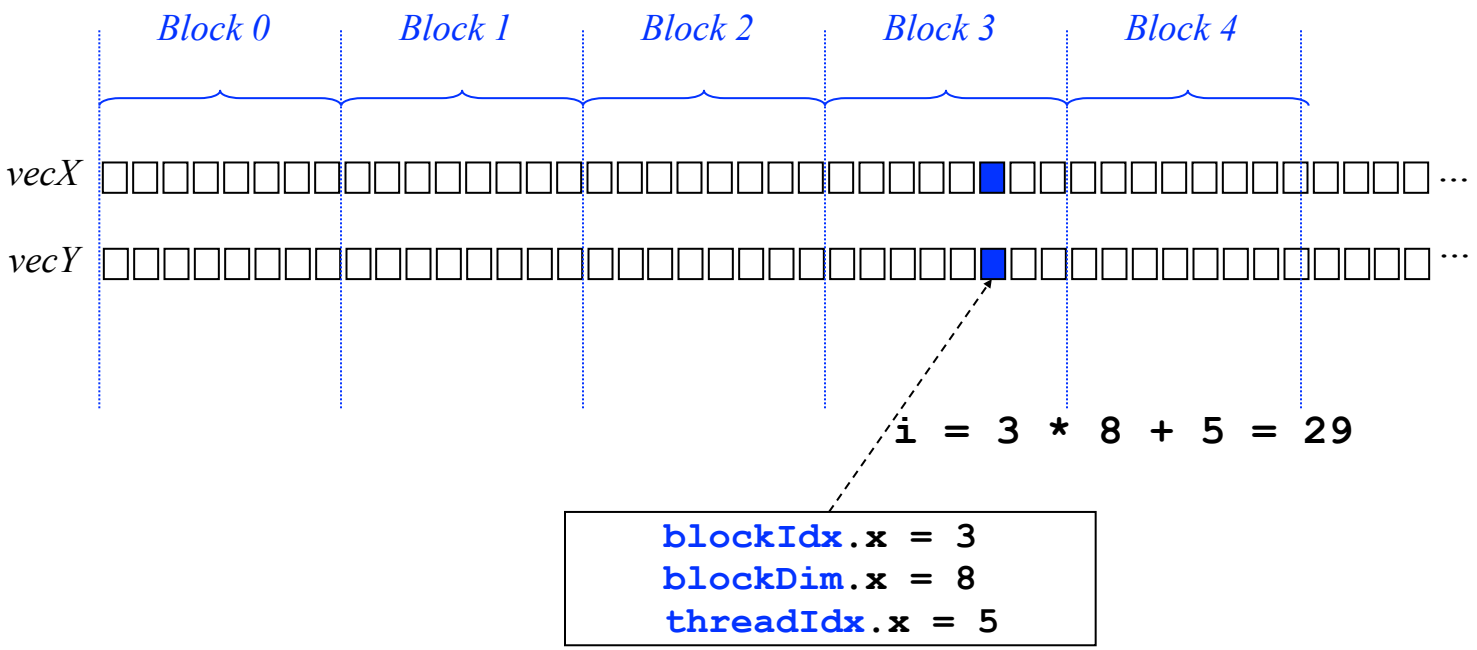
```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The host specifies the number of blocks and block size during kernel invocation:

```
saxpy_gpu<<<numBlocks, blockSize>>>(y_d, x_d, alpha, n);
```

Computing the index

```
i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<n)
   vecY[i] = alpha * vecX[i] + vecY[i];
```



Key differences

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- No need to explicitly loop over array elements – each element is processed in a separate thread
- The element index is computed based on block index, block width and thread index within the block

Key differences

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- Could avoid testing whether $i < n$ if we knew n is a multiple of block size (e.g. use padded arrays --- recall MPI_Scatter issues)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   vecY[i] = alpha * vecX[i] + vecY[i];
}
```

Host code: overview

- The host performs the following operations:
  1. initialize device
  2. allocate and initialize input arrays in host DRAM
  3. allocate memory on device
  4. upload input data to device
  5. execute kernel on device
  6. download results
  7. check results
  8. clean-up

Host code: initialization

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>

int main(int argc, char *argv[])
{
   float *x_host, *y_host;   /* arrays for computation on host*/
   float *x_dev, *y_dev;     /* arrays for computation on device */
   float *y_shadow;          /* host-side copy of device results */

   int n = 32*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i, blockSize, nBlocks;

/* here could add some code to check if GPU device is present */

...
```

Host code: memory allocation

```
...

memsize = n * sizeof(float);

/* allocate arrays on host */

x_host = (float *)malloc(memsize);
y_host = (float *)malloc(memsize);
y_shadow = (float *)malloc(memsize);

/* allocate arrays on device */

cudaMalloc((void **) &x_dev, memsize);
cudaMalloc((void **) &y_dev, memsize);

/* add checks to catch any errors */

...
```

Host code: upload data

```
...

   /* initialize arrays on host */

for ( i = 0; i < n; i++)
{
   x_host[i] = rand() / (float)RAND_MAX;
   y_host[i] = rand() / (float)RAND_MAX;
}

/* copy arrays to device memory (synchronous) */

cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);

...
```

Host code: kernel execution

```
...

   /* set up device execution configuration */
blockSize = 512;
nBlocks = n / blockSize + (n % blockSize > 0);

/* execute kernel (asynchronous!) */

saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

/* could add check if this succeeded */

/* execute host version (i.e. baseline reference results) */
saxpy_cpu(y_host, x_host, alpha, n);

...
```

Host code: download results

```
...

/* retrieve results from device (synchronous) */
cudaMemcpy(y_shadow, y_dev, memsize, cudaMemcpyDeviceToHost);

/* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */

cudaDeviceSynchronize();


/* check results */
nerror=0;
for(i=0; i < n; i++)
{
   if(y_shadow[i]!=y_host[i]) nerror=nerror+1;
}
printf("test comparison shows %d errors\n",nerror);

...
```

Host code: clean-up

```
   ...

   /* free memory on device*/
   cudaFree(x_dev);
   cudaFree(y_dev);

  /* free memory on host */
   free(x_host);
   free(y_host);
   free(y_shadow);

   return 0;
} /* main */
```

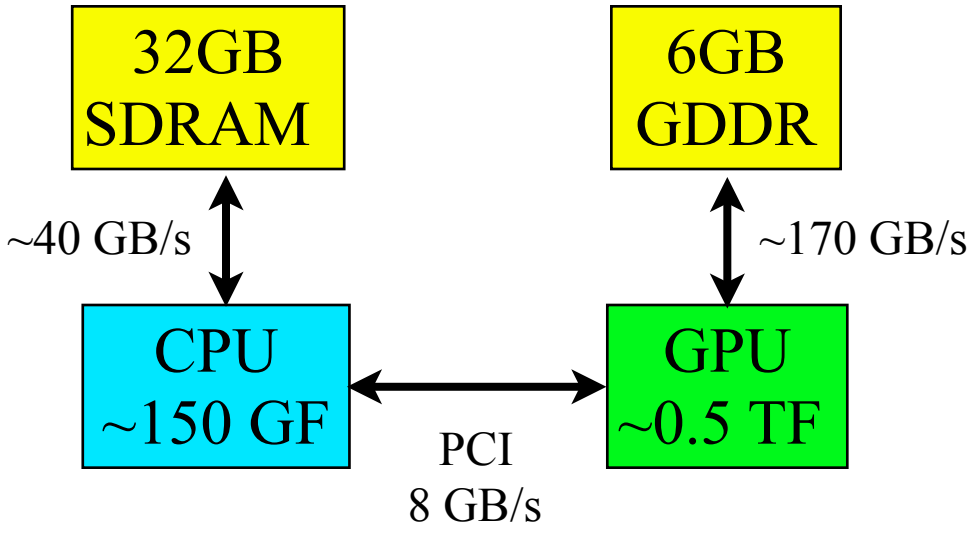
Checking for errors in CUDA calls

```
 ...
 /* check CUDA API function call for possible error */
if (error = cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice))
  {
    printf ("Error %d\n", error);
    exit (error);
  }

 ...

 saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);
 /* make sure kernel has completed*/
 cudaDeviceSynchronize();
 /* check for any error generated by kernel call*/
 if(error = cudaGetLastError())
  {
    printf ("Error detected after kernel %d\n", error);
    exit (error);
  }
```

Compiling

- nvcc -arch=sm_20 -O2 program.cu -o program.x
- -arch=sm_20 means code is targeted at Compute Capability 2.0 architecture (what monk has)
- -O2 optimizes the CPU portion of the program
- There are no flags to optimize CUDA code
- Various fine tuning switches possible
- SHARCNET has a CUDA environment module preloaded. See what it does by executing: module show cuda
- add -lcublas to link with CUBLAS libraries

Be aware of memory bandwidth bottlenecks



- The connection between CPU and GPU has low bandwidth
  - need to minimize data transfers
  - important to use asynchronous transfers if possible (overlap computation and transfer)

Using pinned memory

- The transfer between host and device is very slow compared to access to memory within either the CPU or the GPU

- One way to speed it up by a factor of 2 or so is to use pinned memory on the host for memory allocation of array that will be transferred to the GPU

```
int main(int argc, char *argv[])
{
cudaMallocHost((void **) &a_host, memsize_input)
...
cudaFree(a_host);
```

Timing GPU accelerated codes

- Presents specific difficulties because the CPU and GPU can be computing independently in parallel, i.e. asynchronously
- On the cpu can use standard function **gettimeofday(...)** (microsecond precision) and process the result
- If trying to time events on GPU with this function, must ensure synchronization
- This can be done with a call to cudaDeviceSynchronize()
- Memory copies to/from device are synchronized, so can be used for timing.
- Timing GPU kernels on the CPU may be insufficiently accurate

Using mechanisms on the GPU for timing

- This is highly accurate on the GPU side, and very useful for optimizing kernels

```
...
   cudaEvent_t start, stop;
   float kernel_timer;
...
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   cudaEventRecord(start, 0);

   saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

   cudaEventRecord(stop, 0);
   cudaEventSynchronize( stop );
   cudaEventElapsedTime( &kernel_timer, start, stop );

   printf("Test Kernel took %f ms\n",kernel_timer);
   cudaEventDestroy(start);
   cudaEventDestroy(stop);
```

Linear algebra on the GPU

- Linear algebra on the CPU: BLAS, LAPACK
- GPU analogues: CUBLAS, CULA
- CUSPARSE library for sparse matrices
- Use of highly optimised libraries is always better than writing your own code, especially since GPU codes cannot yet be efficiently optimized by compilers to achieve acceptable performance
- Writing efficient GPU code requires special care and understanding the peculiarities of underlying hardware

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA
- Included with CUDA (hence free)
- Workflow:
  1. allocate vectors and matrices in GPU memory
  2. fill them with data
  3. call sequence of CUBLAS functions
  4. transfer results from GPU memory to host
- Helper functions to transfer data to/from GPU provided

Error checks

- in following example most error checks were removed for clarity
- each CUBLAS function returns a status object containing information about possible errors
- It's very important these objects to catch errors, via calls like this:

```
if (status != CUBLAS_STATUS_SUCCESS) {
print diagnostic information and exit}
```

Initialize program

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>
#include <cublas_v2.h>


int main(int argc, char *argv[])
{
   float *x_host, *y_host; /* arrays for computation on host*/
   float *x_dev, *y_dev;   /* arrays for computation on device */

   int n = 32*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i;

   /*  could add device detection here */

   memsize = n * sizeof(float);
```

Allocate memory on host and device

```
     /* allocate arrays on host */

   x_host = (float *)malloc(memsize);
   y_host = (float *)malloc(memsize);

/* allocate arrays on device */

   cudaMalloc((void **) &x_dev, memsize);
   cudaMalloc((void **) &y_dev, memsize);

   /* initialize arrays on host */

   for ( i = 0; i < n; i++)
   {
      x_host[i] = rand() / (float)RAND_MAX;
      y_host[i] = rand() / (float)RAND_MAX;
   }

   /* copy arrays to device memory (synchronous) */

   cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
   cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);
```

Call CUBLAS function

```
cublasHandle_t handle;
cublasStatus_t status;

status  = cublasCreate(&handle);

int stride = 1;
status = cublasSaxpy(handle,n,&alpha,x_dev,stride,y_dev,stride);

/* check if cublasSaxpy launched succesfully */

if (status != CUBLAS_STATUS_SUCCESS)
 {
   printf ("Error in launching CUBLAS routine \n");
   exit (20);
 }

status = cublasDestroy(handle);
```

Retrieve computed data and finish

```
   /* retrieve results from device (synchronous) */
   cudaMemcpy(y_host, y_dev, memsize, cudaMemcpyDeviceToHost);

  /* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */

   cudaDeviceSynchronize();

   /* use data in y_host*/

   /* free memory */
   cudaFree(x_dev);
   cudaFree(y_dev);
   free(x_host);
   free(y_host);

   return 0;
}
```

OpenACC

- New standard for parallel computing developed by compiler makers. See: http://www.openacc-standard.org/
- Specified in late 2011, released in 2012
- SHARCNET has the PGI compiler on monk which supports it
- OpenACC works somewhat like OpenMP
- Goal is to provide simple directives to the compiler which enable it to accelerate the application on the GPU
- The tool is aimed at developers aiming to quickly speed up their code without extensive recoding in CUDA
- As tool is very new and this course focuses on CUDA, only a brief demo of OpenACC follows

SAXPY with OpenACC

```
...
#include <openacc.h>

void saxpy_openacc(float *restrict vecY, float *vecX, float alpha, int n)
{
     int i;
#pragma acc kernels
     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}

...
   /* execute openacc accelerated function on GPU */
   saxpy_openacc(y_shadow, x_host, alpha, n);
...
```

- OpenACC automatically builds a kernel function that will run on GPU
- Memory transfers between device and host handled by OpenACC and need not be explicit

Compiling SAXPY with OpenACC

```
[ppomorsk@mon54:~] module unload intel
[ppomorsk@mon54:~] module load pgi
[ppomorsk@mon54:~/CUDA_day1/saxpy] pgcc -acc -Minfo=accel -fast saxpy_openacc.c
saxpy_openacc:
     25, Generating copyin(vecX[0:n])
         Generating copy(vecY[0:n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     26, Loop is parallelizable
         Accelerator kernel generated
         26, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes
[ppomorsk@mon54:~/CUDA_day1/saxpy] export ACC_NOTIFY=1
[ppomorsk@mon54:~/CUDA_day1/saxpy] export PGI_ACC_TIME=1
[ppomorsk@mon54:~/CUDA_day1/saxpy] ./a.out
launch kernel  file=/home/ppomorsk/CUDA_day1/saxpy/saxpy_openacc.c function=saxpy_openacc
line=26 device=0 grid=128 block=256 queue=0

Accelerator Kernel Timing data
/home/ppomorsk/CUDA_day1/saxpy/saxpy_openacc.c
  saxpy_openacc
    25: region entered 1 time
        time(us): total=4241617 init=4240714 region=903
                  kernels=22 data=461
        w/o init: total=903 max=903 min=903 avg=903
        26: kernel launched 1 times
            grid: [128]  block: [256]
            time(us): total=22 max=22 min=22 avg=22
```

Is OpenACC always this easy?

- No, the loop we accelerated was particularly easy for the compiler to interpret. It was very simple, and each iteration was completely independent of the others
- If the accelerate directive is placed before a more complicated loop, the compiler refuse to accelerate the region, complaining of errors
- More specific compiler directives must hence be provided for more complicated functions
- Memory transfers must be handled explicitly if we don't want to transfer memory to/from device every time kernel is called
- For complex problems OpenACC grows as complex as CUDA, but it might get better in the future

Introduction to GPU Programming: CUDA

CUDA EXTENSION TO THE C PROGRAMMING LANGUAGE

Storage class qualifiers

Functions

`__global__`	Device kernels callable from host
`__device__`	Device functions (only callable from device)
`__host__`	Host functions (only callable from host)

Data

`__shared__`	Memory shared by a block of threads executing on a multiprocessor.
`__constant__`	Special memory for constants (cached)

CUDA data types

- C primatives:
  - `char`, `int`, `float`, `double`, …
- Short vectors:
  - `int2`, `int3`, `int4`, `uchar2`, `uchar4`, `float2`, `float3`, `float4`, …
  - no built-in vector math (although a utility header, `cutil_math.h`, defines some common operations)
- Special type used to represent dimensions
  - `dim3`
- Support for user-defined structures, e.g.:

```
struct particle
{
    float3 position, velocity, acceleration;
    float mass;
};
```

Library functions available to kernels

- Math library functions:
  - `sin`, `cos`, `tan`, `sqrt`, `pow`, `log`, ...
  - `sinf`, `cosf`, `tanf`, `sqrtf`, `powf`, `logf`, ...
- ISA intrinsics
  - `__sinf`, `__cosf`, `__tanf`, `__powf`, `__logf`, ...
  - `__mul24`, `__umul24`, ...
- Intrinsic versions of math functions are faster but less precise

Built-in kernel variables

`dim3 gradDim`
- number of blocks in grid

`dim3 blockDim`
- number of threads per block

`dim3 blockIdx`
- number of current block within grid

`dim3 threadIdx`
- index of current thread within block

CUDA kernels: limitations

- No recursion (on devices older than CC 2.0)
- No variable argument lists
- No dynamic memory allocation
- No pointers-to-functions
- No static variables inside kernels

Launching kernels

- Launchable kernels must be declared as '`__global__ void`'

```
__global__ void myKernel(paramList);
```

- Kernel calls must specify device execution environment
  - grid definition – number of blocks in grid
  - block definition – number of threads per block
  - optionally, may specify amount of shared memory per block (more on that later)

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramList);
```

Thread addressing

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramlist);
```

- **GridDef** and **BlockDef** can be specified as **dim3** objects
  - grids can be 1D, 2D or 3D
  - blocks can be 1D, 2D or 3D

- This makes it easy to set up different memory addressing for multi-dimensional data.

Thread addressing (cont.)

- 1D addressing example: 100 blocks with 256 threads per block:

```
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```

- 2D addressing example: 10x10 blocks with 16x16 threads per block:

```
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```

- Both examples launch the same number of threads, but block and thread indexing is different
  - `kernel1` uses `blockIdx.x`, `blockDim.x` and `threadIdx.x`
  - `kernel2` uses `blockIdx.[xy]`, `blockDim.[xy]`, `threadIdx.[xy]`

Thread addressing (cont.)

- One-dimensional addressing example:

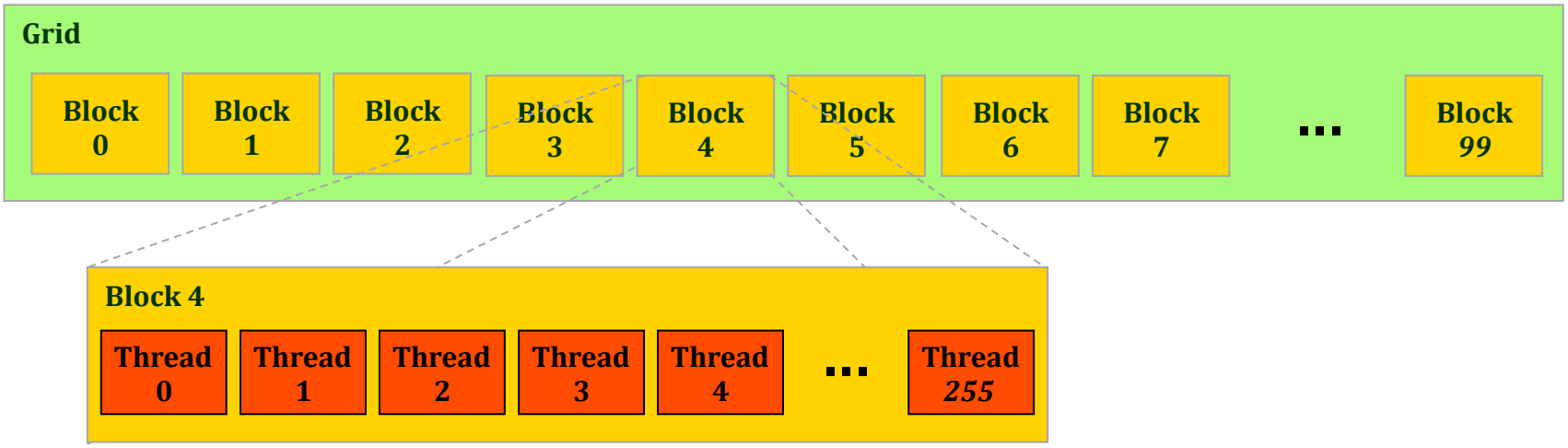
```
__global__ void kernel1(float *idata, float *odata)
{
    int i;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    odata[i] = func(idata[i]);
}
```

- Two-dimensional addressing example:

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
    int x, y, i;

    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;
    i = y * pitch + x;
    odata[i] = func(idata[i]);
}
```

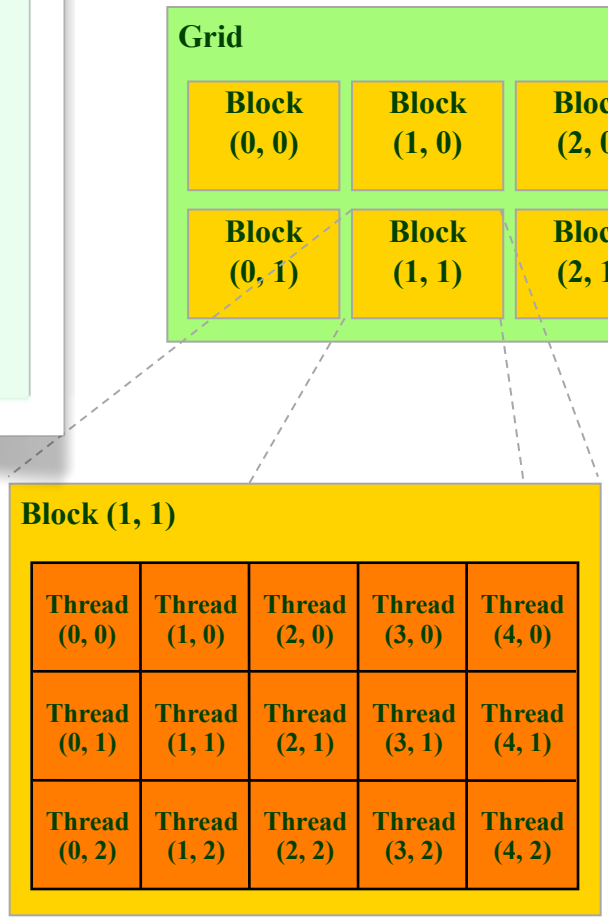
Thread addressing (cont.)

```
__global__ void kernel1(float *idata, float *odata)
{
    int i;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    odata[i] = func(idata[i]);
}
...
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```



Thread addressing (cont.)

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
    int x, y, i;

    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;
    i = y * pitch + x;
    odata[i] = func(idata[i]);
}
...
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```



Introduction to GPU Programming: CUDA

OPTIMIZATION STRATEGIES

Beyond the basics…

- Exposing parallelism
- Memory address coalescing
- Shared memory
- Thread synchronization

Exploiting fully the parallelism of the problem

- A GPU has a large number of cores, to take full advantage of the GPU they must all be given something to do.
- It is hence beneficial to have the work to be done decomposed among a large number of threads.
  - GPU architecture can easily handle large numbers of threads without overhead (unlike CPU)
  - for this to work optimally threads belonging to the same blockmust be executing similar (ideally exactly the same) instructions, operating on different data
  - this means one must avoid divergent branches within a block
  - size of block should be multiple of 32 (warp size), must not exceed the maximum for device

Important caveat: is more threads always useful?

- Each thread consumes some resources, mainly registers and shared memory. Given that these resources are limited, the number of threads "alive" at any one time (i.e. actively running on the hardware) is also limited.

- Hence the benefit of adding more threads tends to plateau.
  - one can optimize around the resources needed, especially registers, to improve performance

Avoiding transfers between GPU and device

- That is a huge bottleneck, but unavoidable since GPU has limited capabilities, most significantly no access to file system (note: AMD's APU Fusion avoids this problem)
- CPU essential because GPU cannot be independent. All kernels must be launched from the CPU which is the overall controller
  - changed on Kepler architecture released in late 2012 on which kernels can launch other kernels
- Using pinned memory helps a bit
- Using asynchronous transfers (overlapping computation and transfer) also helps

Optimizing access to global memory

- A GPU has a large number of cores with great computational power, but they must be "fed" with data from global memory

- If too little computation done on core relative to memory transfer, then it becomes the bottleneck.
  - most of the time is spent moving data in memory rather than number crunching
  - for many problems this is unavoidable

- Utilizing the memory architecture effectively tends to be the biggest challenge in CUDA-fying algorithms

GPU memory is high bandwidth/high latency

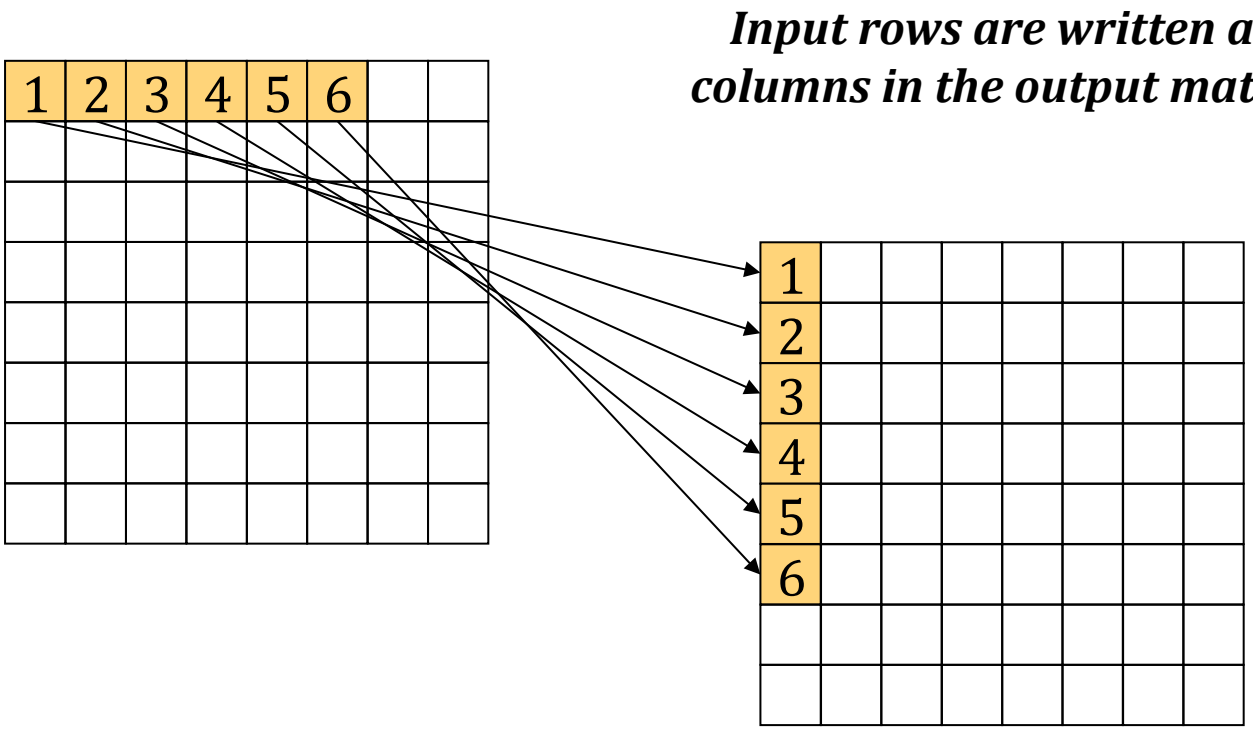
- A GPU has potentially high bandwidth for data transfer from global memory to cores. However, the latency for this transfer for any individual thread is also high (hundreds of cycles)
- Using many threads, latency can be overcome by hiding it among many threads.
  - group of threads requests some memory, while it is waiting for it to arrive, another group is computing
  - the more threads you have, the better this works
- The pattern of global memory access is also very important, as cache size of the GPU is very limited.

Global memory access is fast when coalesced

- It is best for adjacent threads belonging to the same warp (group of 32 threads) to be accessing locations adjacent in memory (or as close as possible)
- Good access pattern: thread i accesses global memory array member a[i]
- Inferior access pattern: thread i accesses global memory array member as a[i*nstride] where nstride >1
- Clearly, random access of memory is a particularly bad paradigm on the GPU

For some problems coalesced access is hard

- Example: matrix transpose
- A bandwidth-limited problem that is dominated by memory access

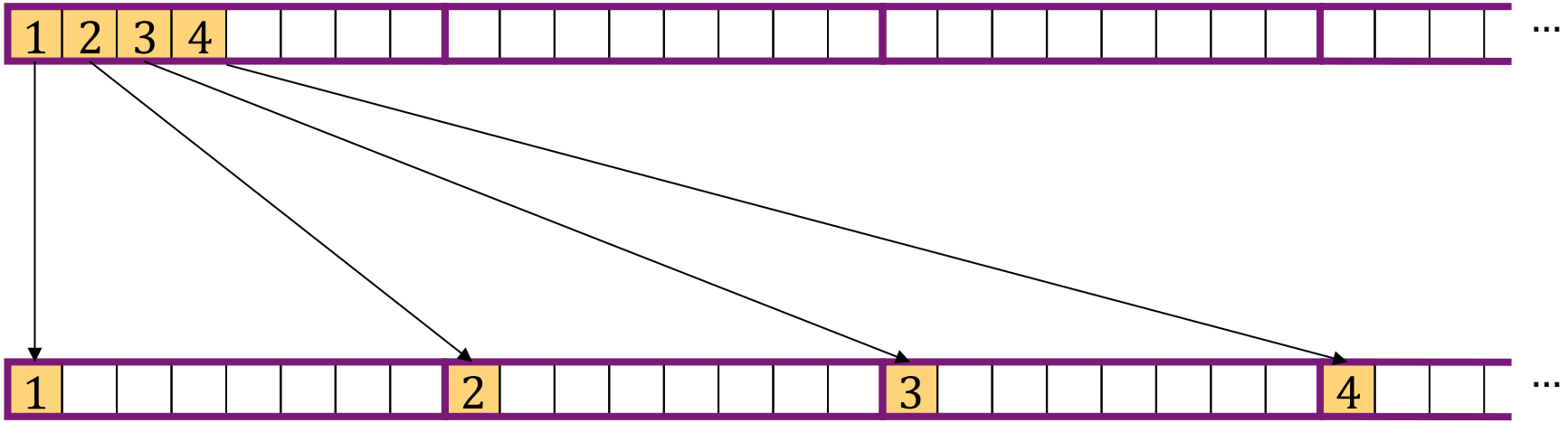


The naïve matrix tranpose

```
__global__ void transpose_naive(float *odata, float *idata, int width,int height)
{
    int xIndex, yIndex, index_in, index_out;

    xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height)
    {
        index_in = xIndex + width * yIndex;
        index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

Naïve matrix transpose (cont.)

Since the matrices are stored as 1D arrays, here's what is actually happening:



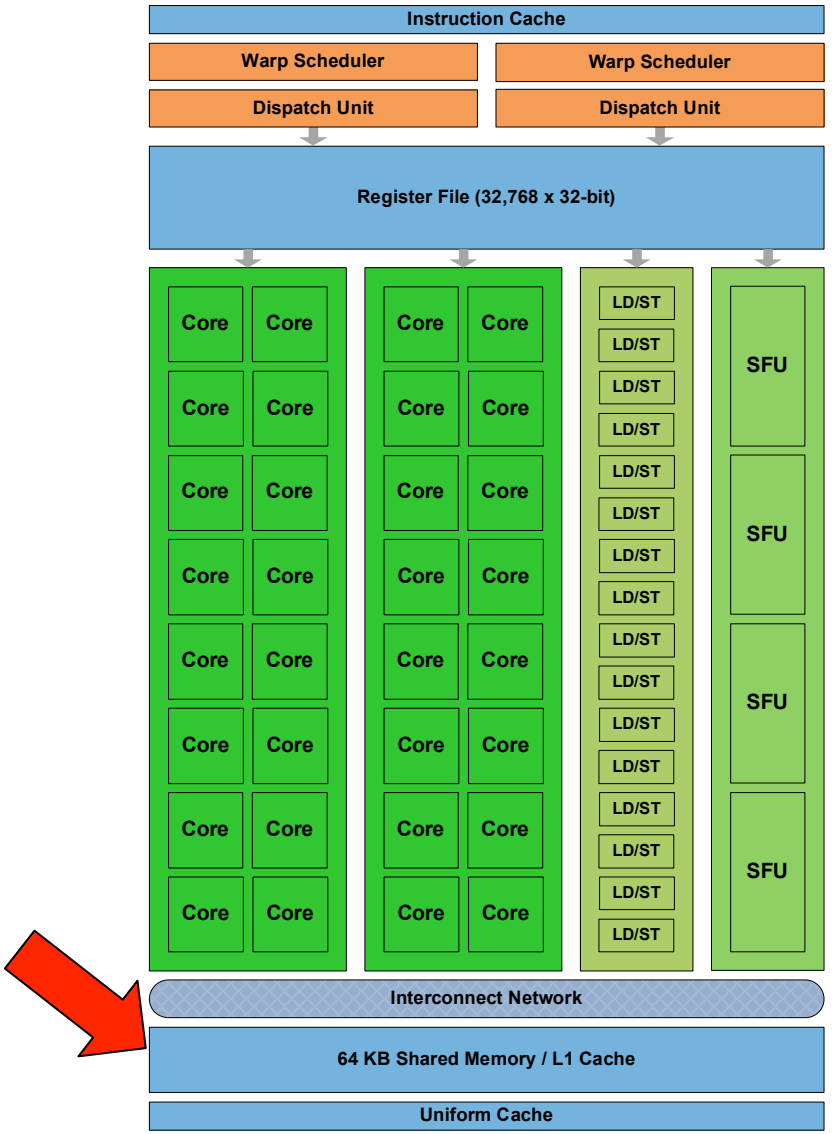
Can this problem be a avoided?

- Yes, by using a special memory which does not have a penalty when accessed in a non-coalesced way
- On the GPU this is the shared memory
- Shared memory accesses are faster than even coalesced global memory accesses. If accesing same data multiple times, try to put it in shared memory.
- Unfortunately, it is very small (48 KB or 16KB)
- Must be managed by the programmer

Shared memory

- Each multiprocessor has some fast on-chip shared memory
- Threads within a thread block can communicate using the shared memory
- Each thread in a thread block has R/W access to all of the shared memory allocated to a block
- Threads can synchronize using the intrinsic

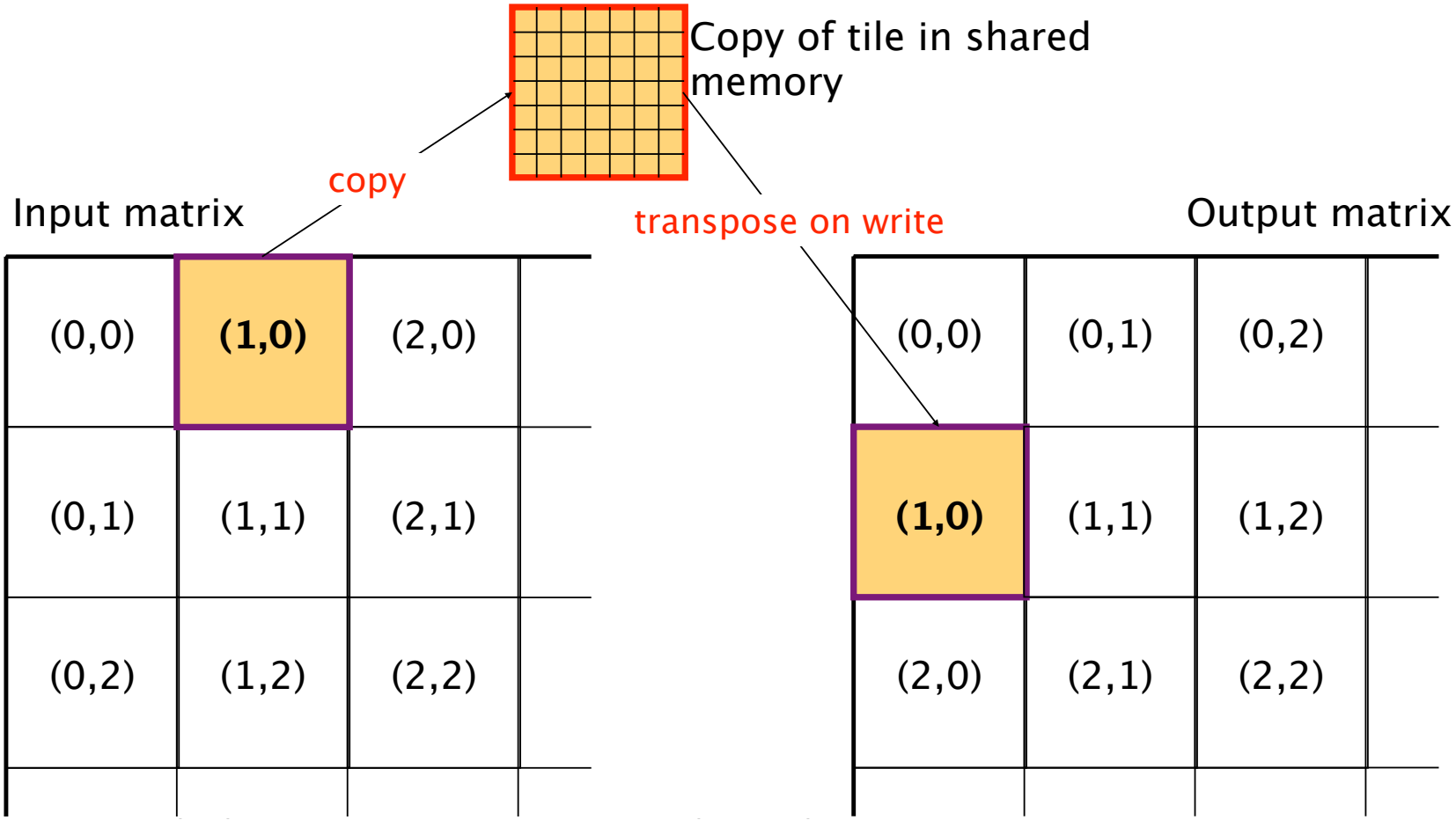
```
__syncthreads();
```



Using shared memory

- To coalesce the writes, we will partition the matrix into 32x32 tiles, each processed by a different thread block
- A thread block will temporarily stage its tile in shared memory by copying it from the input matrix using coalesced reads
- Each tile is then transposed as it is written out to its properly location in the output matrix
- The main difference here is that the tile is written out using coalesced writes

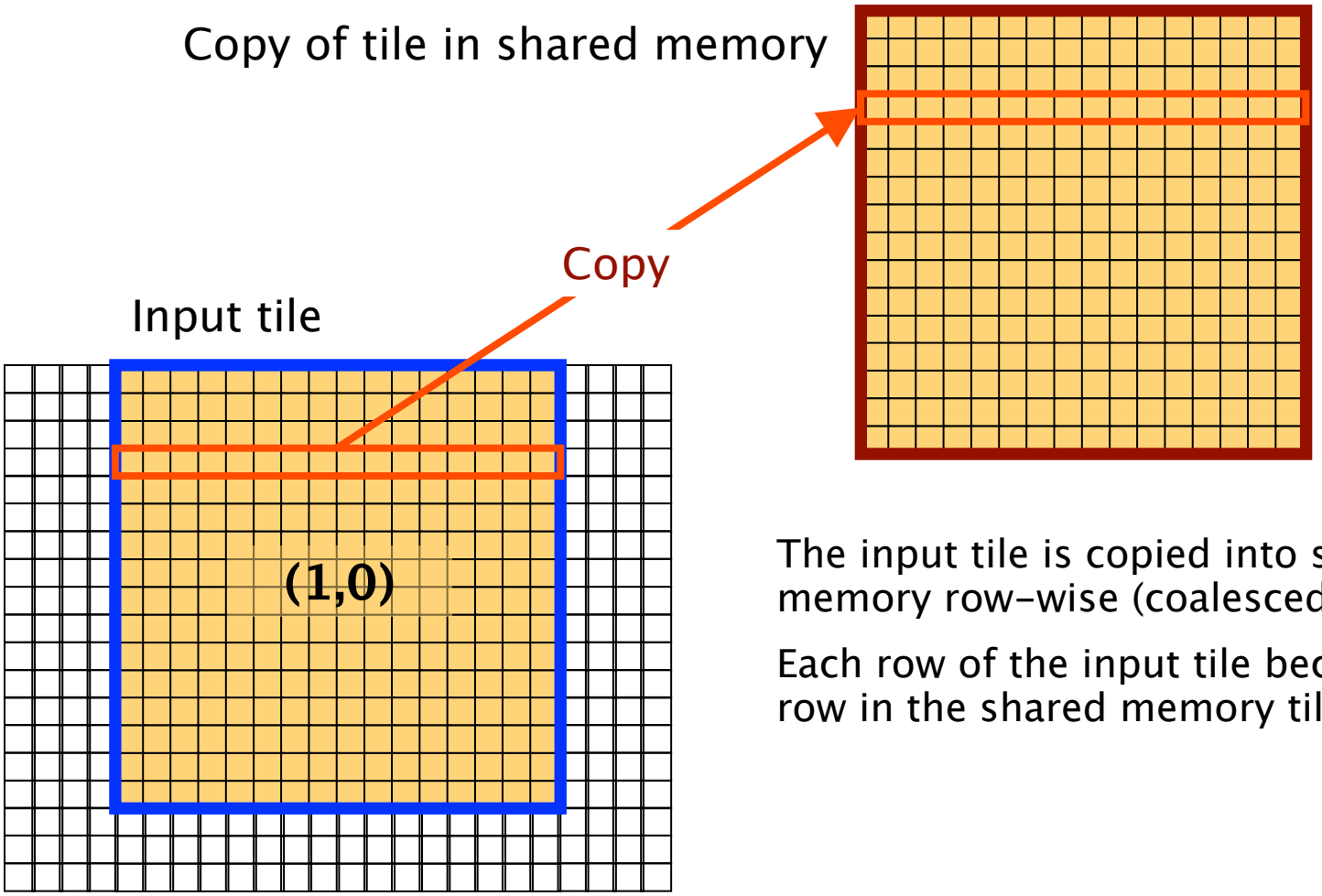
Optimized matrix transpose



Optimized matrix transpose (1)

```
__global__ void transpose(float *odata, float *idata,
                          int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM];
    unsigned int xIndex, yIndex, index_in, index_out;

    /* read the matrix tile into shared memory */
    xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if ((xIndex < width) && (yIndex < height))
    {
        index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    /* write the transposed matrix tile to global memory */
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if ((xIndex < height) && (yIndex < width))
    {
        index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

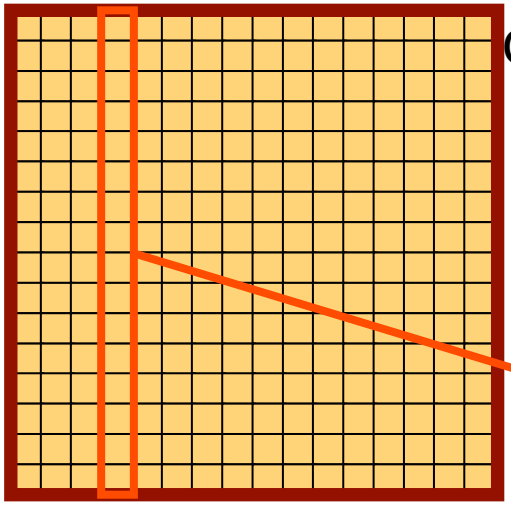
Optimized matrix transpose (cont.)



The input tile is copied into shared memory row-wise (coalesced reads).

Each row of the input tile becomes a row in the shared memory tile.

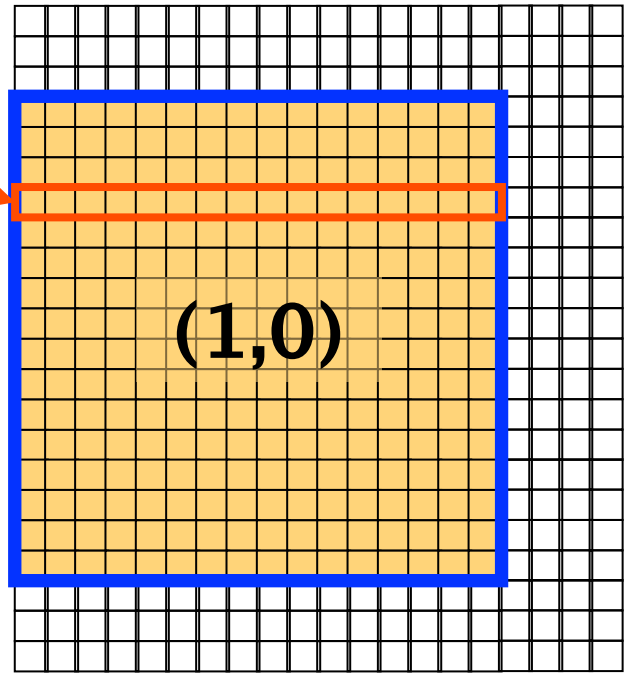
Optimized matrix transpose (cont.)



Copy of tile in shared memory

Output tile

Transpose



The output tile is written row-wise (coalesced writes) by copying the corresponding columns from shared memory.

One additional complication: bank conflicts

- Not a big concern but something to keep in mind
- Shared memory ***bank conflicts*** occur when the tile in shared memory is accessed column-wise
- Illustration of the need to really know the hardware when coding for GPU
- Bank conflicts matter only in highly optimised code where other sources of inefficiency have been eliminated

Shared memory banks

- To facilitate high memory bandwidth, the shared memory on each multiprocessor is organized into equally-sized ***banks*** which can be accessed simultaneously

- However, if more than one thread tries to access the same bank, the accesses must be serialized, causing delays
  - this situation is called a ***bank conflict***

- The banks are organized such that consecutive 32-bit words are assigned to consecutive banks

Shared memory banks (cont.)

- There are 32 banks, thus:

```
bank# = address % 32
```

- The number of shared memory banks is closely tied to the warp size
- Shared memory accesses are serviced such that the threads in the first half of a warp and the threads in the second half of the warp will not cause conflicts
- Thus we have `NUM_BANKS = WARP_SIZE`

Bank conflict solution

- In the matrix transpose example, bank conflicts occur when the shared memory is accessed column-wise as the tile is being written

- The threads in each warp access addresses which are offset from each other by `BLOCK_DIM` elements (with `BLOCK_DIM = 32`)

- Given 32 shared memory banks, that means that all accesses hit the same bank!

Bank conflict solution

- The solution is surprisingly simple – instead of allocating a `BLOCK_DIM x BLOCK_DIM` shared memory tile, we allocate a `BLOCK_DIM x (BLOCK_DIM+1)` tile

- The extra padding breaks the pattern and forces concurrent threads to access different banks of shared memory
  - the columns are no longer aligned on 32-word offsets
  - no additional changes to the device code are needed

Optimized matrix transpose (2)

```
 __global__ void transpose(float *odata, float *idata,
                           int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM + 1];
    unsigned int xIndex, yIndex, index_in, index_out;

    /* read the matrix tile into shared memory */
    xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if ((xIndex < width) && (yIndex < height))
    {
        index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    /* write the transposed matrix tile to global memory */
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if ((xIndex < height) && (yIndex < width))
    {
        index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Helpful tools

- CUDA 5.0+ is out since 2012. More recent versions may be out as a free preview for developers. Anyone can sign up to be a developer.
- CUDA 5.0+ includes Nsight, an Integrated Development Environment (IDE) for Linux/Mac based on Eclipse. IDE incorporates CUDA-aware editor, profiler and debugger in one close-integrated package. Try it out!
- There is a Visual Studio edition of Nsight for Windows
- On SHARCNET the DDT visual debugger has powerful GPU debugging capability

Further reading

- CUDA Programming Guide

- CUDA sample projects
  - many contain extended documentation
  - similarity to the matrix transpose, the reduction project is an excellent step-by-step walkthrough of how to optimize code for the hardware (read/write coalescing, shared memory, bank conflicts, etc.)

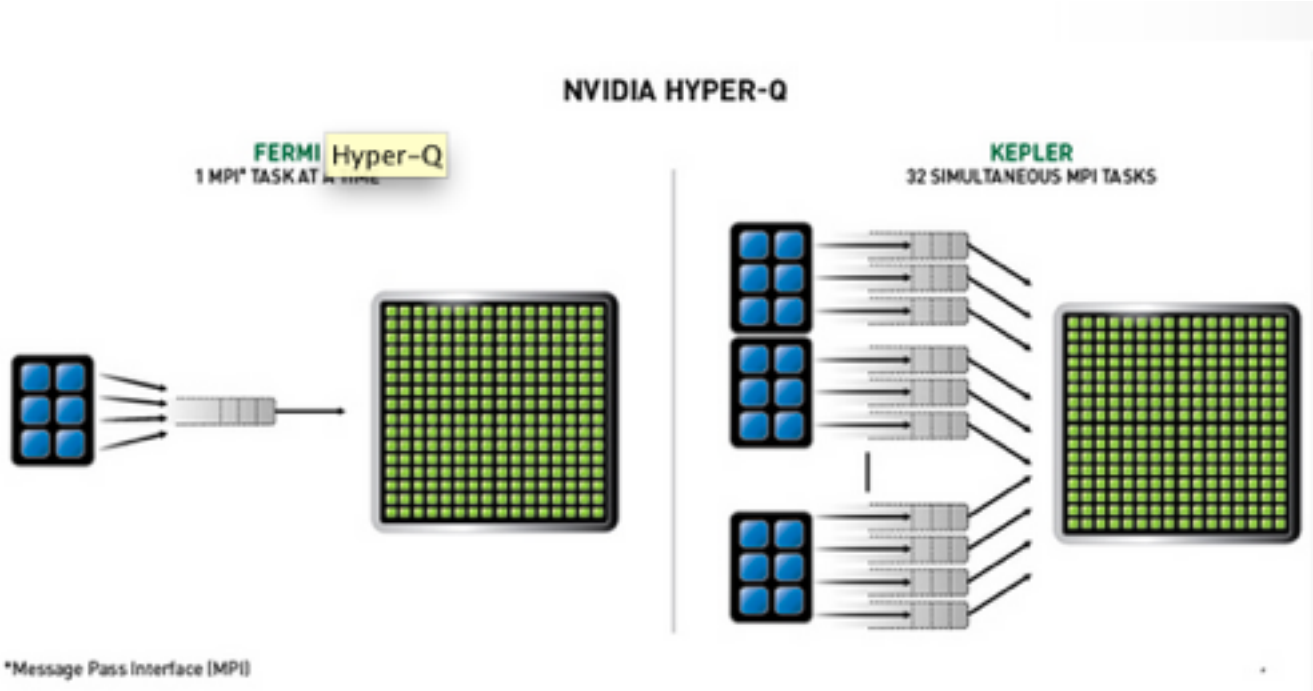
- Lots of documentation/presentations/tutorials online
- NVIDIA website - lots of materials

Just released - NVIDIA Kepler K20 and K20X

	Tesla K10	Tesla K20	Tesla K20X
CC	3.0	3.5	3.5
Dynamic Parallelism HyperQ	NO	YES	YES
DP floating point	0.19 TF	1.17 TF	1.31 TF
SP floating point	4.58 TF	3.52 TF	3.95 TF
CUDA cores	2x1536	2496	2688
Memory	8 GB	5 GB	6 GB

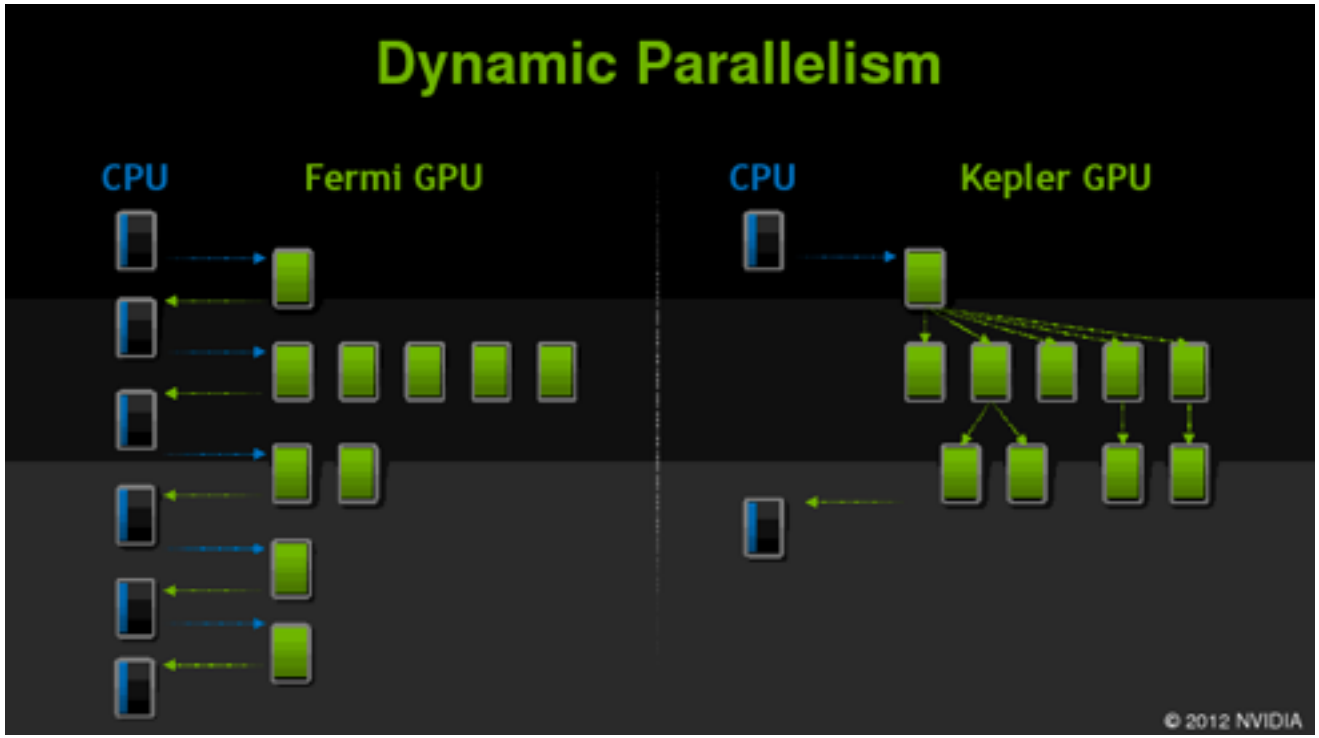
HyperQ

- multiple CPU cores can access the GPU at once



Dynamic parallelism

- Kernels can be launched on the GPU



Dynamic parallelism

- Kernels can be launched by other kernels, syntax similar to that on GPU