GPU Architecture Overview

Nazım Dugan
nazim.dugan@unibas.ch

What is GPGPU?

- Graphics processors are similar to vector processors. Therefore, they can be used for high performance computing applications.

- GPUs are cheap and common because of the gaming industry.

- GPUs are used together with CPUs as accelerators. GPUs have their own memory which can communicate with system memory through PCI bridge.

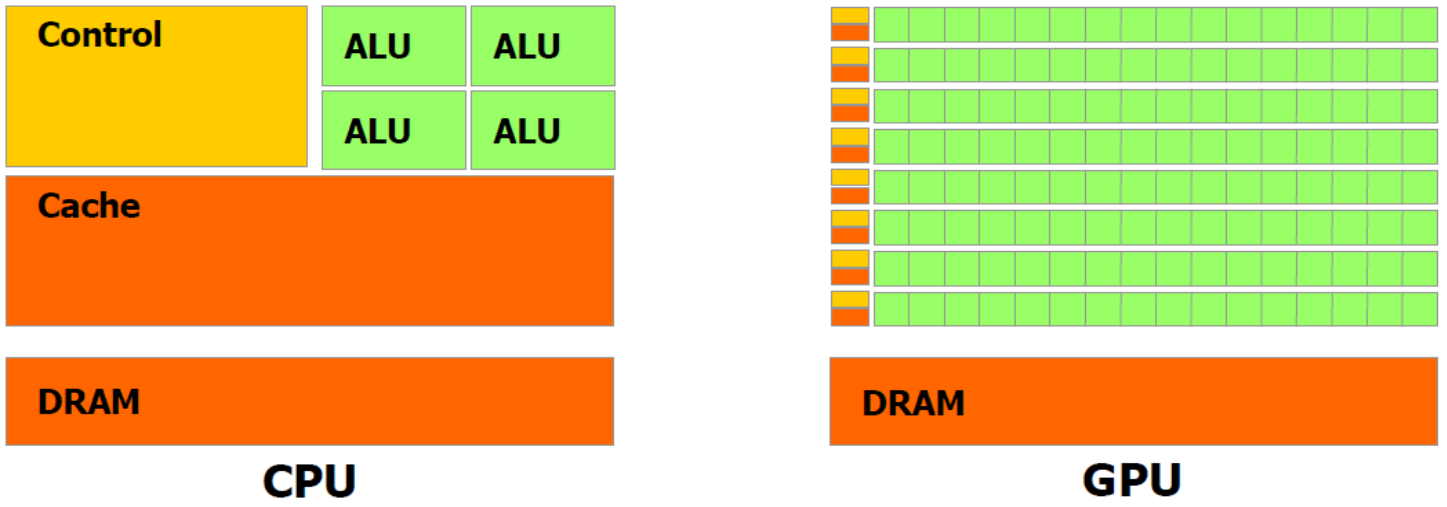
History

- Starting from 90's graphics APIs such as OpenGL were used for non graphical applications.

- In 2006 CUDA was introduced by NVIDIA
  Just for NVIDIA GPUs

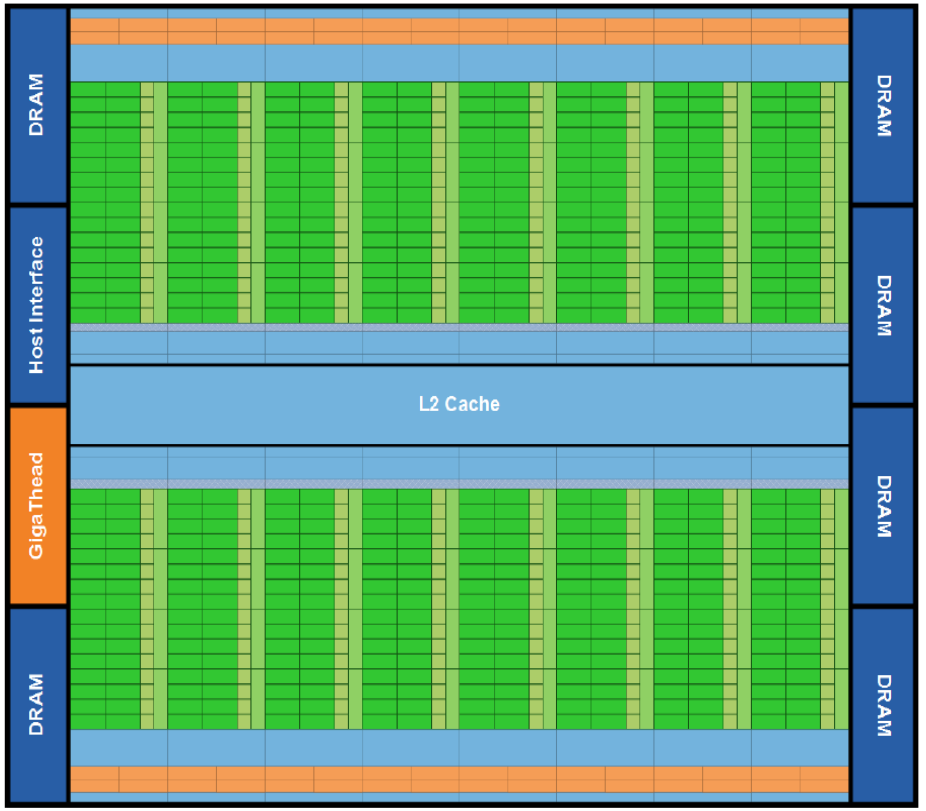
- In 2008 OpenCL was developed by Apple
  For both NVIDIA and ATI GPUs

CPU vs GPU

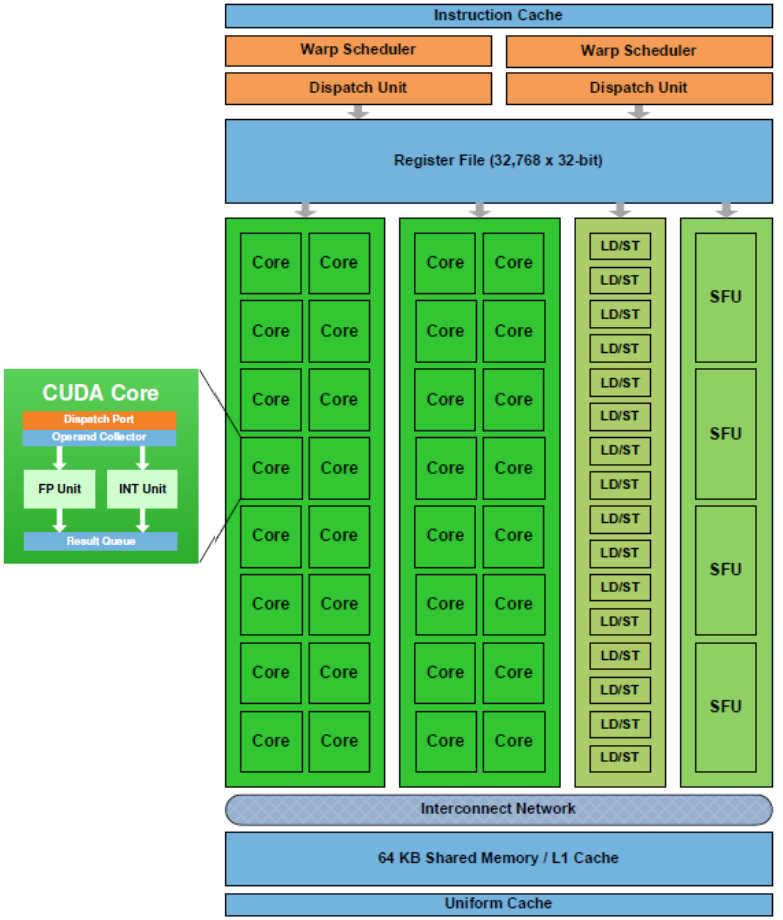


GPU has more computing units compared to CPU. It is suitable for SIMD (same instruction multiple data) computation.

GPU Architecture



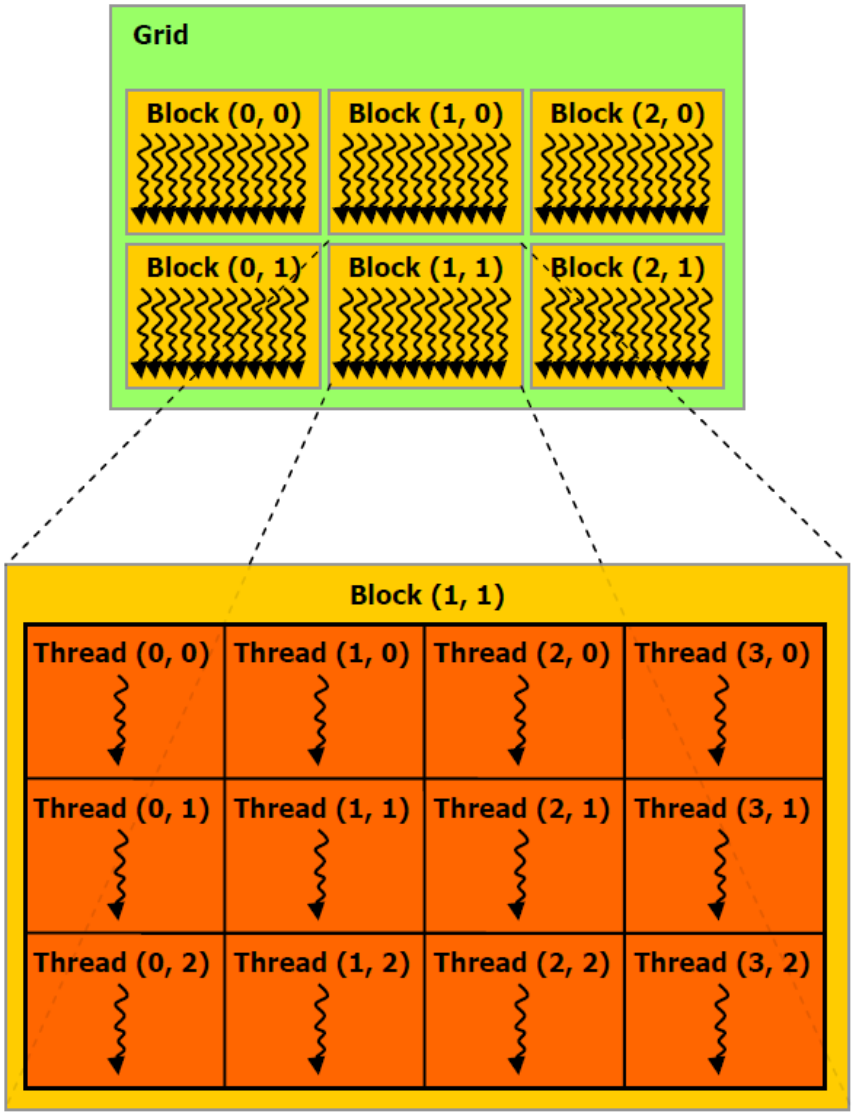
Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).



Fermi Streaming Multiprocessor (SM)

Read **NVIDA Fermi Compute Architecture Whitepaper** for details.

GPU SIMD processing



- Each block runs on a single **stream multiprocessor (SM)**.
- Threads in a block runs in groups of 32, called as a **warp**.
- A warp runs in a **SIMD** fashion.
- Threads of a block can access to the same **shared memory**.
- Threads in a block can be synchronized.
- Multiple blocks can be active on a single SM depending on the available resources (registers, shared memory.
- Occupancy = active warps / max active warps

GPU Properties

- Optimized for high degree of parallelism.
- High memory bandwidth.
- Software managed cache (shared memory)
- Data should be transferred to device memory over relatively slow PCI bus. Therefore, complexity of operations should justify the moving data. Complexity should be larger than O(N) in order to benefit from GPU.
- Optimized for single precision. Fermi is two times slower in double precision. (Previous cards are much slower in double precision.)

GPU Properties

Tesla Specifications (CC < 2.0)

- 8 CUDA cores per SM (Logically 32 threads in a warp)
- 16 KB / SM on-chip RAM for shared memory
- 16 bank access to shared memory

Fermi Specifications (CC ≥ 2.0)

- 32 or more CUDA cores per SM
- 64 KB / SM of on-chip RAM with a configurable partitioning of shared memory and L1 cache
- 32 bank access to shared memory
- Improved double precision floating point performance over Tesla
- Concurrent kernel execution

Kepler Specifications (CC ≥ 3.0)

- SMX: 192 cores (6 x Fermi SM)
- Dynamic Parallelism: GPU kernels can spawn new threads without going back to CPU.
- Hyper-Q: Improved concurrent kernel runs

CUDA Basics

CUDA Installation

- Check CUDA Compatibility:

  http://www.nvidia.com/object/cuda_gpus.html

  (AMD graphic cards are not CUDA compatible but OpenCL compatible.)

- Install latest driver and CUDA Toolkit from:

  http://developer.nvidia.com/cuda-downloads

  (Also download: GPU Computing SDK code samples)

Compute Capability

- CC 1.x : Previous cards
- CC 2.x : Fermi
- CC 3.x : Kepler

Two APIs

- CUDA Driver API: Low level, more control on device

- C runtime for CUDA: High level, easy programming

Device Query

Output of NVIDIA SDK **deviceQuery** program:

```
Device 0: "GeForce GTX 480"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1535 MBytes (1609760768 bytes)
  (15) Multiprocessors x ( 32) CUDA Cores/MP:    480 CUDA Cores
  GPU Clock rate:                                1401 MHz (1.40 GHz)
  Memory Clock rate:                             1848 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

> nvidia-smi -a : Device name, hardware sensors, utilization ...

exercise 0

Compile and run NVIDIA SDK **deviceQuery** program.

Path: *NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery*

CUDA Extensions to C

- Filename: ***.cu**
- **#include <cuda.h>** or **<cuda_runtime.h>** (Not really needed, nvcc includes automatically. Only needed when using CUDA libraries with other compilers)

Host Code

- Cuda functions such as **cudaMalloc, cudaMemcpy, cudaFree**
- Kernel run: functionName **<<< execution configuration >>>** (kernel arguments);
- Variable type qualifiers: **__device__** , **__constant__** , **__shared__**

Device Code

- Function type qualifiers: **__global__** , **__device__**
- Variable type qualifier: **__shared__**
- Thread identifiers: **threadIdx , blockDim, blockIdx**, **gridDim**
- Synchronization: **__syncthreads() . . .**

How to Compile

Use NVIDIA **nvcc** compiler:

> nvcc -arch sm_20 program.cu

Linking CUDA with Fortran:

• .cu file entry function should be defined as **extern "C"** and function name should have an additional "_" character:

```
extern "C" void kernel_wrapper_(parameters)
```

where all the parameters should be pointers.

• Function call from Fortran:

```
CALL kernel_wrapper(parameters)
```

• Compile .cu file as: *nvcc -arch sm_20 -c cudaFile.cu*
• Link as *: gfortran fortranFile.f90 cudaFile.o -lcudart*

• Be careful about array index differences in C and Fortran when using multidimensional arrays.

See **examples/call_cuda.f90** and **examples/fortranInterface.cu**

Function Type Qualifiers

__global__ : executed on device, callable from host only, return type must be void

__device__ : executed on device, callable from device only

__host__ : executed on host, callable from host only

If not specified, function type is **__host__**

Simple Program

Device Kernel: Runs on the GPU

```
__global__ void addOne(double *a) {

        int b = blockIdx.x;
        int t = threadIdx.x;

        int i = b * blockDim.x + t;

        a[i]++;
}
```

Each thread computes a single value of data array. Computation will be parallelized as long as there are enough threads.

n = blockDim.x * gridDim.x

global function return type should be **void**

CPU equivalent: Serial computation with loop

```
void addOne_cpu(int n, double *a) {

        for (int i=0; i<n; i++)
                a[i]++;
}
```

threadIdx : thread id

blockDim : number of threads in a block

blockIdx : block id

gridDim : number of blocks

Simple Program

Host Code: Runs on the CPU

```
int main() {

        int n = 2048;

        double *data = (double*) malloc(n * sizeof(double));
        for (int i=0; i<n; i++) {
                data[i] = (double)i;
        }

        /**** CUDA ****/
        double *data_dev;
        cudaMalloc((void**) &data_dev, n * sizeof(double));

        cudaMemcpy(data_dev, data, n * sizeof(double) , cudaMemcpyHostToDevice);

        dim3 nBlocks(32,1,1);
        dim3 nThreads(64,1,1);
        addOne <<< nBlocks, nThreads >>> (data_dev);

        cudaMemcpy(data, data_dev, n * sizeof(double) , cudaMemcpyDeviceToHost);

        cudaFree(data_dev);
        /**** CUDA ****/

        cout << "data[n-1] = " << data[n-1] << endl;
        free(data);
}
```

Simple Program

Create memory buffer on device memory:

```
double *data_dev;
cudaMalloc((void**) &data_dev, n * sizeof(double));
```

Copy data to device memory:

```
cudaMemcpy(data_dev, data, n * sizeof(double) , cudaMemcpyHostToDevice);
```

Run kernel:

```
dim3 nBlocks(32,1,1);
dim3 nThreads(64,1,1);
addOne <<< nBlocks, nThreads >>> (data_dev);
```

Fermi Maximum grid size: 65535 x 65535 x 65535

Fermi Maximum block size: 1024 x 1024 x 64
total number of threads ≤ 1024
(May be even less if registers are not enough
Total registers per block : 32768)
-maxrregcount N flag limits register usage)

Copy data back to host:

```
cudaMemcpy(data, data_dev, n * sizeof(double) , cudaMemcpyDeviceToHost);
```

Free device memory:

```
cudaFree(data_dev);
```

exercise 1

- Compile Simple Program of the previous slides.

- Run the program with different configuration parameters and check output:
  1) grid size = 32, block size = 64
  2) grid size = 32, block size = 32
  3) grid size = 2, block size = 1024
  4) grid size = 1, block size = 2048

- Check the error code of the kernel run for 3rd and 4th configurations

You can check the errors by **cudaGetLastError()** function:

```
cudaError_t error = cudaGetLastError();
cout << "error code = " << error << " : " << cudaGetErrorString(error) << endl;
```

exercise 2

- Convert simple program kernel in such a way that it computes the whole array even if **n** is not equal to **blockDim.x * gridDim.x** but a multiple of it. So each thread should process **n / (blockDim.x * gridDim.x)** consecutive elements.

- Have timings for:

1) sending data to device
2) kernel run
3) getting data back to host
4) cpu computation

CUDA kernel runs are non-blocking.

Use **cudaThreadSynchronize()** function before getting system time.

Have kernel run timings for :

1) nBlocks = 256, nThreads = 128
2) nBlocks = 512, nThreads = 64
3) nBlocks = 1024, nThreads = 32

4) nBlocks = 1024, nThreads = 64
5) nBlocks = 1024, nThreads = 256
6) nBlocks = 1024, nThreads = 512
7) nBlocks = 1024, nThreads = 1024

8) nBlocks = 8192, nThreads = 1024

__device__ functions

executed on device, callable from device only

addOne kernel with device function:

```
__device__ void addOne_block(double *blockData) {

        // thread id is enough for computation
        int t = threadIdx.x;

        blockData[t]++;
}

__global__ void addOne(int n, double *data) {

        int b = blockIdx.x;

        // each block gets its data pointer as function argument
        addOne_block(data + b*blockDim.x);
}
```

There is no constraint on **device** function return type.

Memory types

Host Memory: System memory which can be accessed by host code

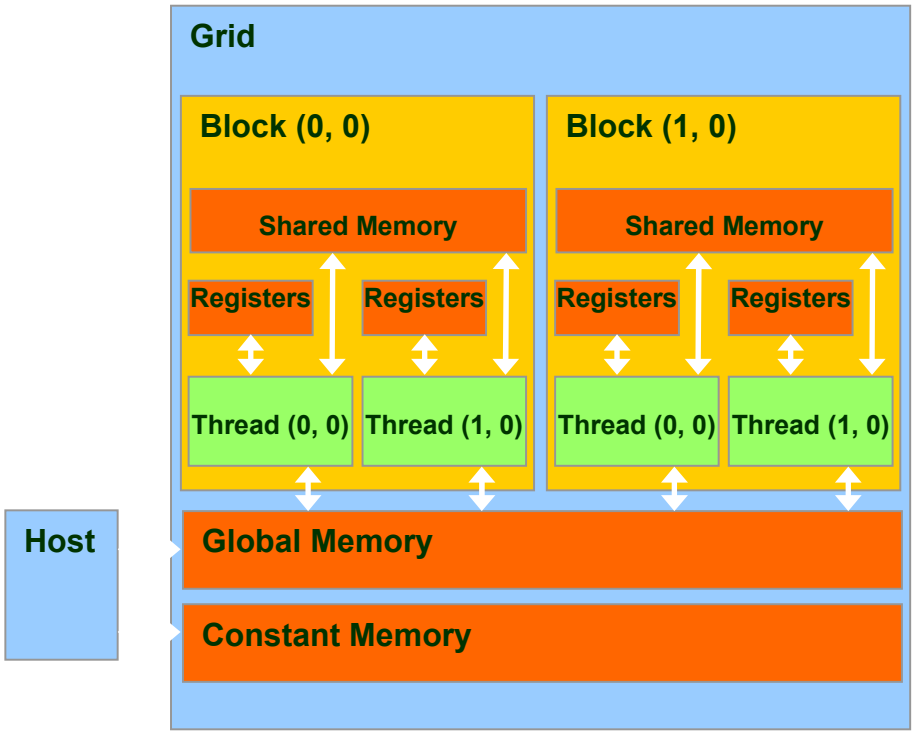
Global Memory: Device memory accessible by all threads, off-chip: slow

Shared Memory: Fast on-chip memory, accessible by all threads of a block through 32 banks, bank conflicts may occur if two threads try to access through same bank.

Constant Memory: Resides on device memory, cached on SM read only cache space.

Texture and Surface Memory: Resides on device memory. Optimized for 2D data access.
(For Fermi, not very important.)

Local Memory: Used when register space is not enough, on the device global memory, coalescing conditions are automatically satisfied.
(no manual usage)



Host – Device Data Transfer

Host – Device communication is slow.

- Minimize host-device data transfers.
- Send data all in once to avoid overheads.

NVIDIA SDK **bandwithTest** program:

```
Device 0: GeForce GTX 480
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     5763.4

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     6109.1

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     148063.7
```

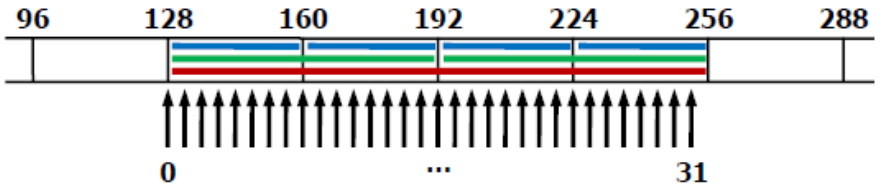
Coalesced Global Memory Access

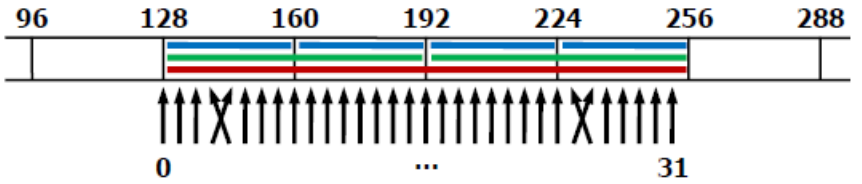
• Global memory access has large overhead (400 – 600 clock cycles).

• A warp of threads can access to the global memory simultaneously if all the data is in the same 128 byte segment.

• For single precision: 32 x 4 = 128 bytes. Full warp access is coalesced.
(Non-coalesced access is 10x slower.)

• For double precision: 16 x 8 = 128 bytes. Half warp (16 threads) access is coalesced.
(Non-coalesced access is 4x slower.)





example:

Coalesced access:
Threads of a warp process sequential data

```
int nt = blockDim.x;
int t = threadIdx.x;

for (int j=0; j<nPerThread; j++)
        a[j*nt + t]++;
```

Non-coalesces access:
Each thread processes sequential data

```
int t = threadIdx.x;

for (int j=0; j<nPerThread; j++)
        a[t*nPerThread + j]++;
```

exercise 3 (10 pts)

- Re-organize the **addOne** kernel of *exercise 2* to have coalesced memory access.

- Compare the computation times (without data transfer times) of following versions for double precision data size of 2^23 = 8388608:

    1) GPU non coalesced access
    2) GPU coalesced access

  with kernel configuration parameters:

    1) nBlocks = 1024, nThreads = 256
    2) nBlocks = 1024, nThreads = 512
    3) nBlocks = 1024, nThreads = 1024
    4) nBlocks = 8192, nThreads = 1024

- Discuss the effect of **nPerThread** on the timing ratios of coalesced and non coalesced versions

- **Optional:** Put some unused data in the beginning of the device data array. (You should enlarge the device allocation).
  You would observe that the code slows down when the unused data size is not a multiple of 128 bytes (16 doubles).

Thread Divergence

Threads of a warp (32 threads) execute in a SIMD manner. If they are diverged with if statements then their execution is serialized.

example:

```
if (threadIdx.x%32 < 16) {     // divergence in warp execution

if (threadIdx.x < blockDim.x/2) { // divergence not in warp execution
```

See **examples/divergence.cu**

Memory Access Cautions

If multiple threads write to same memory address, their access order is not defined. Therefore, you may get different results on each run.

example:

```
int t = threadIdx.x;
a[0] = t;
```

Atomic functions: Used when sequential access is necessary.

atomicAdd, atomicSub, atomicExch, atomicMin ....

example:

All threads add 1 to the value of same element. Use atomic add:

```
a[0] += 1; // gives wrong result

atomicAdd(&a[0], 1);
```

Synchronization

Syncthreads() : Applies a barrier on the activities of threads in a block until all the threads of the block finishes their tasks up to that point.

example: In the following kernel, threads increment values of elements Initialized by other threads. Therefore, synchronization is necessary.

```
__global__ void myKernel(int n, double *a) {

        int t = threadIdx.x;
        int nt = blockDim.x;

        // initialize values
        for (int i=0; i<nPerThread; i++)
                a[nt*i+t] = nt*i+t;

        __syncthreads();

        // increment values with inverse order
        for (int i=0; i<nPerThread; i++)
                a[n-(nt*i+t)-1] +=1;

}
```

Threads of different blocks can only be synchronized by kernel relaunches.

cudaThreadSynchronize() function may be used at host code to synchronize all threads.

exercise 4

- Use the kernel of the previous slide to modify the values of an array of size 512 x 16. Make the computation with only one thread block of size 512.

- Run the kernel with and without synchronization of the threads and check the value of the last element for the two cases.

exercise 5

In this exercise you will work on a modified version of **addOne** kernel which accesses block data as a 2D array.

(necessary modifications in the code are specified for each task. *Ex (*1*)*)

0) Run the code in its current state and have timing result.

1) Interchange the definitions of **tx** and **ty** in the addOne kernel and have timing result.

2) Modify the code to have a 2D thread block of 16x16

3) Have timing result for 2D thread block version of the code for data sizes 4096 and 4624.
How slow is it for the latter case?

Variable Type Qualifiers

__device__ : declares a variable on device memory, has lifetime of the application. (host code only)

__constant__ : optionally used with **__device__**, declares a variable in constant memory space, has lifetime of the application. (host code only)

__shared__ : optionally used with **__device__**, declares a variable in shared memory space, has lifetime of the block. (can also be used in device code without **__device__** specifier)

If not specified in the device code variable usually resides in registers.

Caution: For global __device__ variables, you dont have to send the pointer to kernel. However, you have to use **cudaMemcpyToSymbol** or **CudaMemcpyFromSymbol** instead of **cudaMemcpy.**

See **examples/globalDeviceVar.cu** and **examples/constanMemory.cu** **(**from SIGCSE 2011 workshop web page**)**

Shared Memory

• Fast on-chip memory, accessible by all threads of a block.

Used for:
• When multiple threads in a block use the same data from global memory, shared memory can be used to access the data from global memory only once.

• Coalesced global memory access may not be possible due to data non-locality. Shared memory enables coalesced access to global memory in such a situation.

Fermi Technical Information:

• Default: 48 KB / block (16 KB / block L1 cache)

Adjustable to be 16 KB/block (48 KB/block L1 cache):

```
cudaFuncSetCacheConfig(kernel_name, cudaFuncCachePreferL1);
```

("**-Xptxas -dlcm=cg**" flag disables L1 cache. See **examples/matmul_L1cache.cu**)

• 32 bank access

Caution: Using too much shared memory would reduce occupancy.
Use it with care

Shared Memory - Usage

```
//static (kernel code)
__shared__ double data[size];

//static (host code)
__device__ __shared__ double data[size];


//dynamic

// kernel code
extern __shared__ double data[];

// host code
int sharedMemSize = arraySize * sizeof(double);
kernel <<< gridSize, blockSize, sharedMemSize >>> (arguments);
```

Caution: All the dynamically allocated shared variables of a kernel start at the same memory address. If you want to have more than one:

```
extern __shared__ double data[];
double *a = data;
double *b = &data[Length_of_a];
```

Shared Memory – Bank Conflicts

Threads access to shared memory over 32 banks. If two threads try to access over the same bank, a bank conflict occurs and their accesses are serialized.

For Fermi, bank conflicts do not occur if threads of a warp access sequential data:

```
__shared__ double data[size];

int b = blockIdx.x;
int t = threadIdx.x;

int i = b * blockDim.x + t;

data[t] = global_data[i];
```

Bank conflict example:

```
int bankSize = 32;
double a = data[t * bankSize];
```

Broadcast:
All threads in a half warp access the same shared memory location. Fermi devices have the additional ability to multi cast shared memory accesses.
(see **examples/broadcast.cu**)

```
double a = data[t/16];
```

exercise 6

Modify simple program kernel to make the additions in shared memory. Choose block size as 128 and make each thread to process 32 elements. (Data size for a single block will be 128 x 32 x 8 = 32768 bytes which is less than the shared memory size.) Total data size **n** should be a multiple of 128x32 and grid size should be **n** / (128x32) . The kernel should do the following steps:

- Copy all the data for the corresponding block to the shared memory.

- Do the additions in shared memory.

- Copy all the data back to the original positions in the global memory.

(Do you need synchronization between these steps?)

Have two kernels doing the above jobs with and without bank conflicts and compare their computation times for a double precision array size of (128 x 32) x 256 .

Make sure that the global memory accesses are coalesced in both kernels so that you can measure the sole effect of the bank conflicts.

Profiling

Cuda profiling tools enable monitoring of kernel resource usage and some properties like occupancy.

System variables:

export CUDA_PROFILE=1
export CUDA_PROFILE_CSV=1
export CUDA_PROFILE_CONFIG=config.txt
export CUDA_PROFILE_LOG=profile.csv

config.txt file:

Includes optional keywords like timestamp, regperthread, gridsize, threadblocksize, dynsmemperblock, stasmemperblock, memtransferdir, memtransfersize, streamid, gld_request, gst_request.

Simple program profiling results for *n=128 (nBlock=4, nThreads=32)*

```
method,gputime,cputime,regperthread,occupancy
memcpyHtoD,2.880,5.000
_Z6addOnePd,3.072,11.000,6,0.083
memcpyDtoH,1.504,15.000
```

Simple program profiling results for *n=65536 (nBlock=128, nThreads=512)*

```
method,gputime,cputime,regperthread,occupancy
memcpyHtoD,89.056,187.000
_Z6addOnePd,10.816,11.000,6,1.000
memcpyDtoH,83.520,208.000
```

-Xptxas -v flag gives info about register, memory usage

exercise 7

• Run and profile the shared memory addition kernel without bank conflicts of the previous exercise with four different execution configuration parameters:

(Set data length n = 1048576, nPerThread = 8)

Configuration 1 : nBlocks = 256, nThreads = 512
Configuration 2 : nBlocks = 512, nThreads = 256
Configuration 3 : nBlocks = 1024, nThreads = 128
Configuration 4 : nBlocks = 2048, nThreads = 64
Configuration 4 : nBlocks = 4096, nThreads = 32

Shared memory / block = (nThreads x nPerThread x 8) bytes for each configuration.

For each execution configuration have the kernel computation times and use profiling to monitor the occupancy. How the computation time and occupancy changes with shared memory usage and nThreads? Which one is the optimum configuration?

(**Occupancy** = number of active warps on a SM / max number of active warps on a SM

Fermi max active warps / SM: 48 (1536 threads)
Fermi max active blocks / SM: 8

)

Reduction Example

Below kernel sums up values of an array partially.

```
__global__ void reduction(double *data, double *result) {

        int gid = threadIdx.x + blockIdx.x * blockDim.x;
        int id = threadIdx.x;

        for(int s=blockDim.x/2; s>0; s>>=1) {
                if (id < s) {
                        data[gid] += data[gid + s];
                }
                __syncthreads();
        }

        if (id == 0)
                result[blockIdx.x] = data[gid];
}
```

Bitwise >>= operator makes division by 2^n

s >>=1 : s = s/2;
s >>=2 : s = s/4;

Doing the reduction in the shared memory would be beneficial since data elements are read and written many times.

exercise 8 (10 pts)

- Modify the reduction kernel of the previous slide to make the reduction in shared memory.

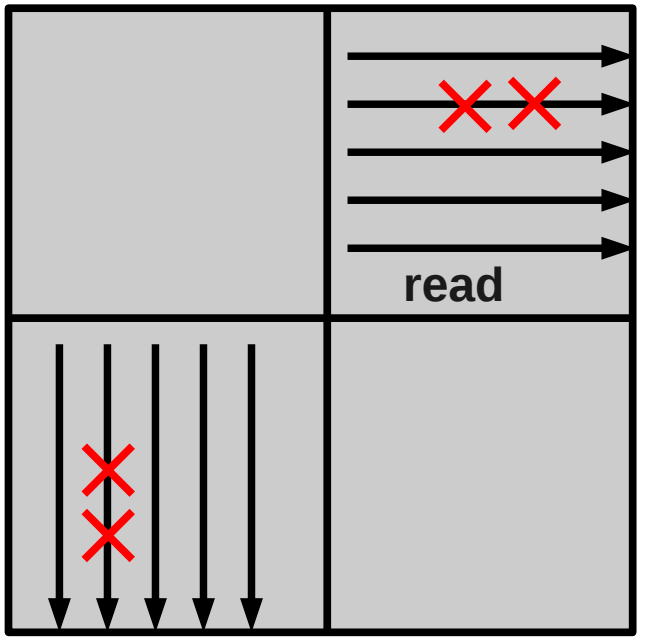
- Set block size to its maximum value in order to increase data reuse.

- Have kernel run times with and without shared memory for a data array size of 4194304.

Matrix Transposition Example

NVIDIA SDK naive Transpose:
(no shared memory)

```
// Each block transposes/copies a tile of TILE_DIM x TILE_DIM elements
// using TILE_DIM x BLOCK_ROWS threads, so that each thread transposes
// TILE_DIM/BLOCK_ROWS elements.  TILE_DIM must be an integral multiple of BLOCK_ROWS

#define TILE_DIM    16
#define BLOCK_ROWS  16


// width and height must be integral multiples of TILE_DIM
__global__ void transposeNaive(float *odata, float* idata, int width, int height, int nreps)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i] = idata[index_in+i*width];
    }
  }
}
```



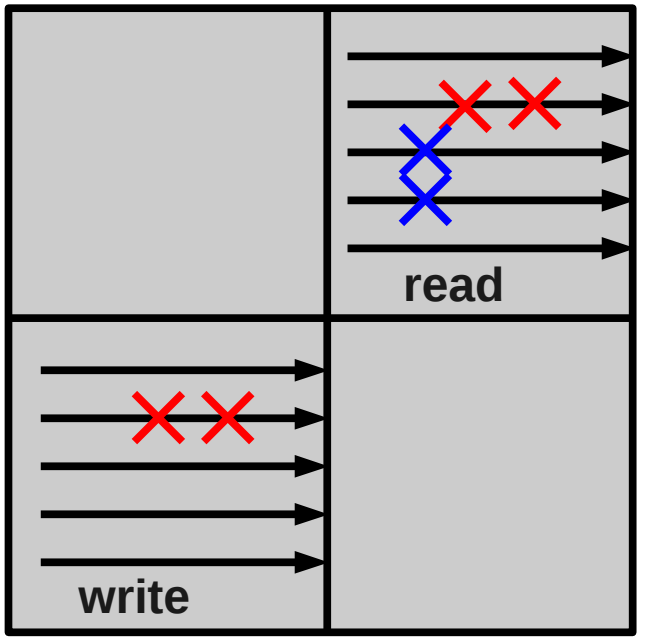
Matrix Transposition – shared mem

Coalesced global memory access is not possible due to data non-locality. In such a situation:

- Transfer data to shared memory by coalesced access
- Do calculation in shared memory
- Write back to global memory by coalesced access

NVIDIA SDK Coalesced Transpose:
(with shared memory)

```
__global__ void transposeCoalesced(float *odata, float *idata, int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```



exercise 9

Compile and run NVIDIA SDK **transpose** program.

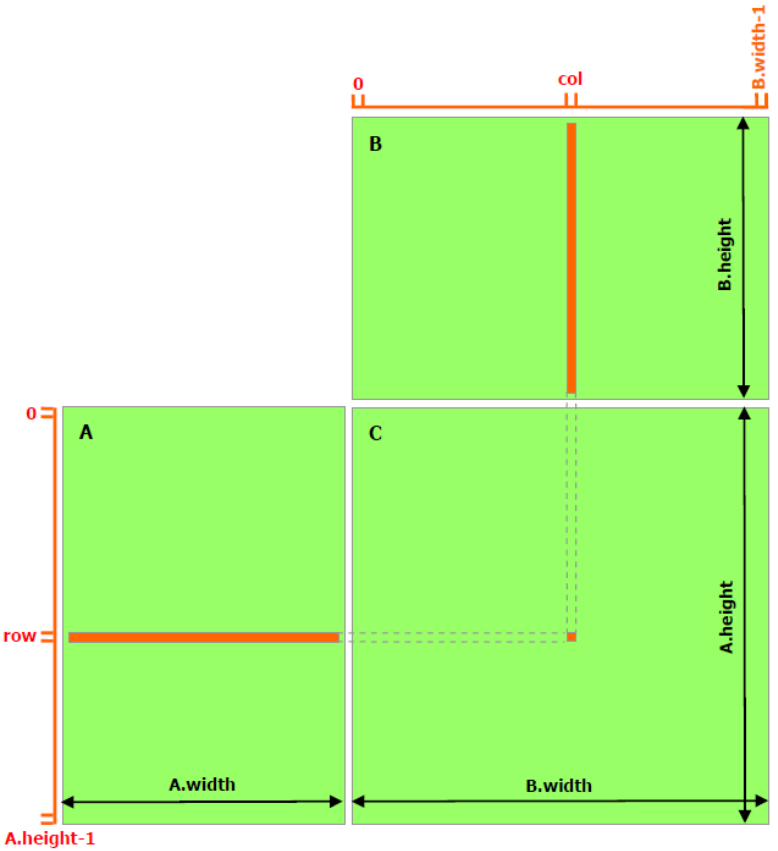
Path: NVIDIA_CUDA-5.0_Samples/6_Advanced/transpose

Matrix Multiplication Example

For each element of C, one row of A and one column of B should be read from memory. Therefore, A matrix is accessed B.width times and B matrix is accessed A.height times.

No shared memory:

```
__global__ void matmul(double *a, double* b, double *c, int aw, int bw) {

        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        double sum = 0.0;

        for (int i = 0; i < aw; i++) {
                sum += a[row*aw+i] * b[i*bw+col];
        }

        c[row*bw+col] = sum;
}
```



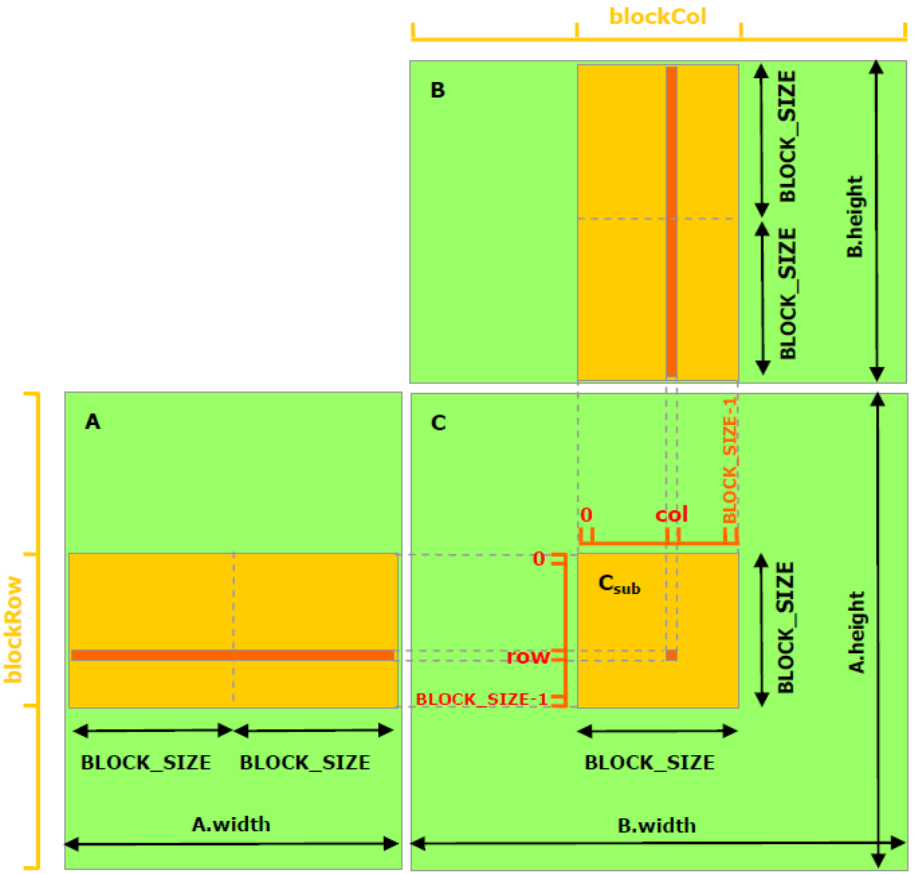
Matrix Multiplication Example

Shared memory version with blocking:
Each block computes one sub-matrix C_{sub} using rectangular sub-matrices of A and B.

In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension block_size as necessary and C_{sub} is computed as the sum of the products of these square matrices.

Sub-matrix multiplications are done in shared memory. Therefore, A is read (B.width / block_size) times from global memory and B is read (A.height / block_size) times.

See *NVIDIA SDK code samples* or *CUDA C Programming Guide* for source code.



exercise 10 (15 pts)

- Convert the matrix multiplication kernel in to a shared memory version using the strategy of the previous slide.

- Compare the performances of no shared memory and shared memory versions for matrix sizes of (2560 x 2560).

- Compare the performance of shared memory version with cpu codes. (**matmul_cpu.cpp, matmul_cpu_tiles.cpp, dgemm_mkl.cpp**)

- Convert your code in to single precision and compare the performances of different precisions. Also compare single precision GPU code with single precision CPU codes (**matmul_cpu_float.cpp, matmul_cpu_tiles_float.cpp, sgemm_mkl.cpp**)

(vi command to replace **double** with **float** : " :%s/double/float/g ")