Universität Bielefeld

Introduction to GPU Programming using CUDA

Olaf Kaczmarek
University of Bielefeld

STRONGnet Summerschool 2011
ZIF Bielefeld
20.06.2011

Literature: see /home/gpu/doc for CUDA documentation or http://developer.nvidia.com
Acknowledgement: most figures and infos are taken from NVIDIA documentation

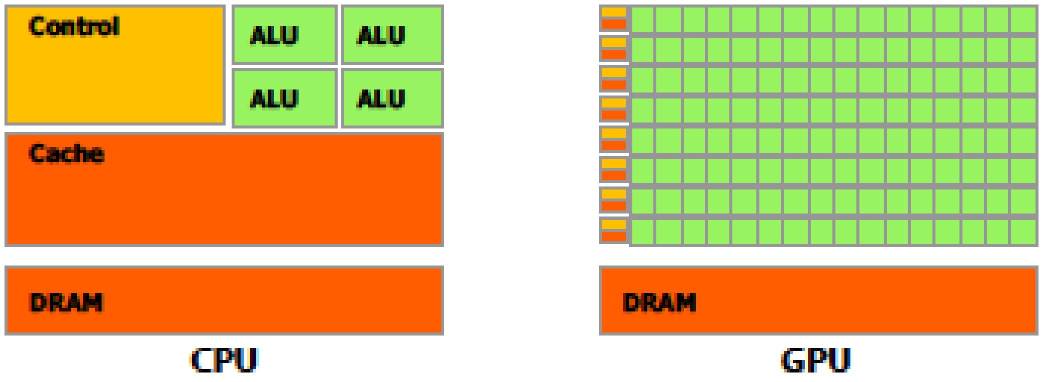
What is a GPU?

- **Graphics Processing Unit: rendering of polygons, shading, texturing**

That is fun if you like fancy games!

What is more important for us:

➢ **GPGPU: General Purpose Computing on Graphics Processing Units**



- **Lots of computing cores with a simpler architecture than cpu cores**
- **High memory bandwidth > 100GB/s on global device memory**
- **Cheap (at least the consumer cards)**

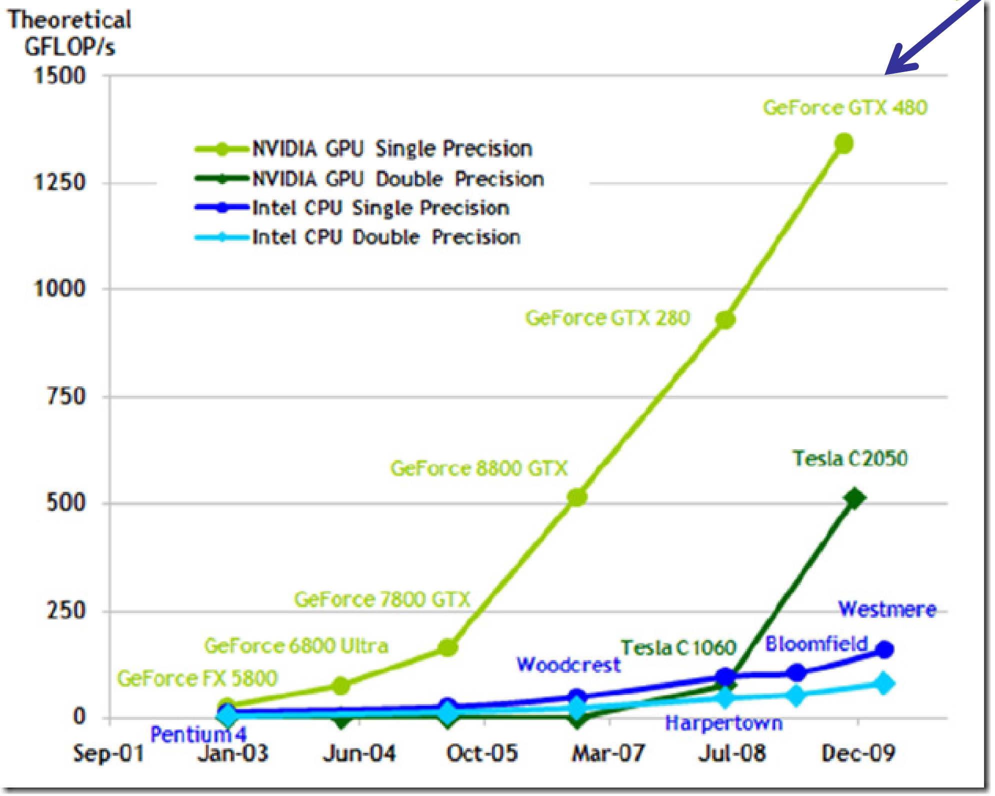
Theoretical performance and bandwidth – GPU vs CPU

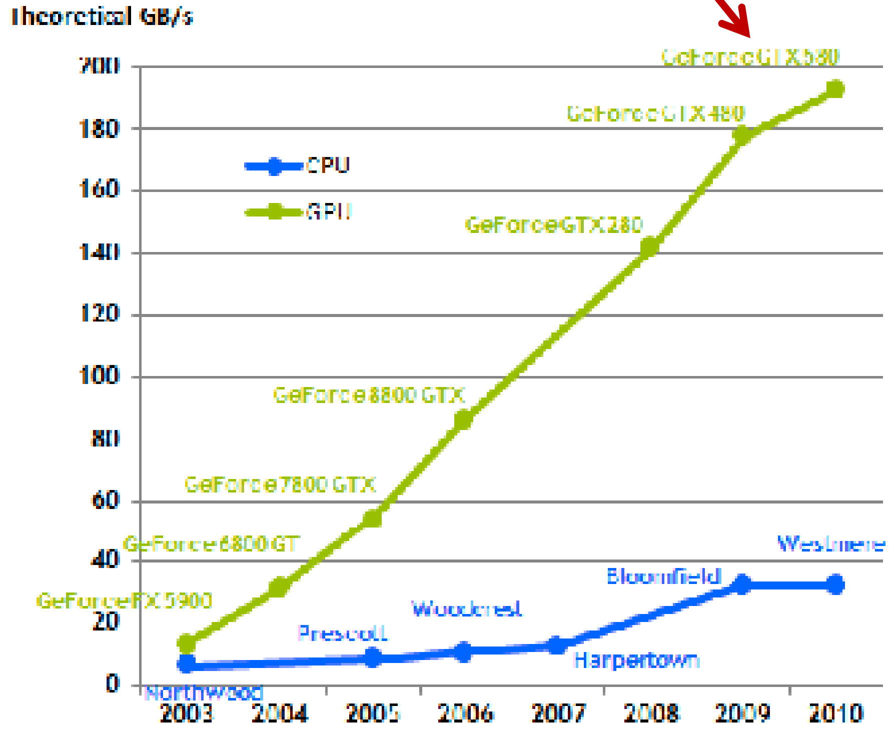
- **Limitation in Lattice QCD code is usually bandwidth not computations**

  e.g. Wilson fermion matrix*vector: 1320 flops/site, 1440 bytes/site (32bit)

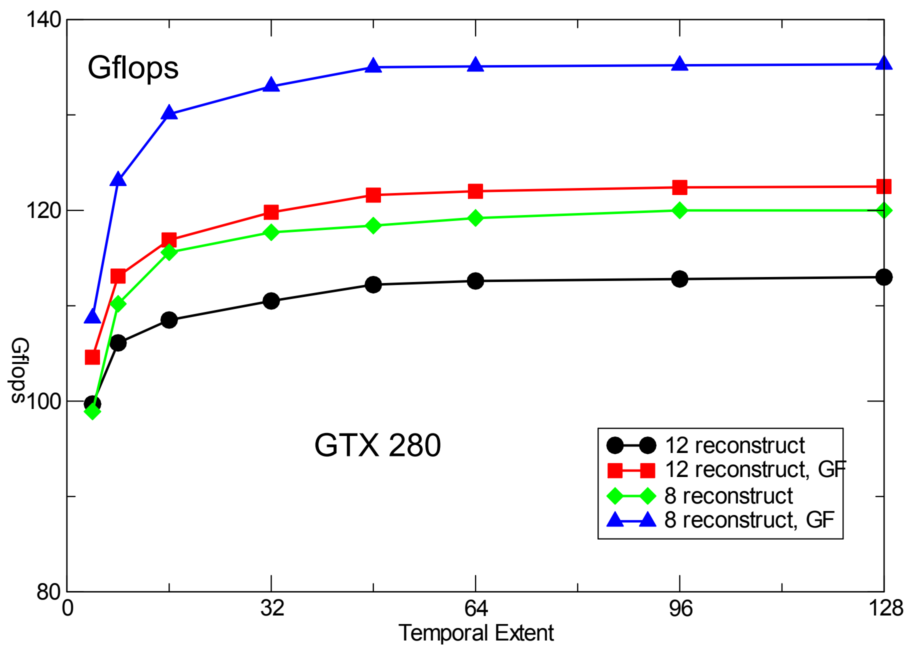
even 10% of peak is a lot

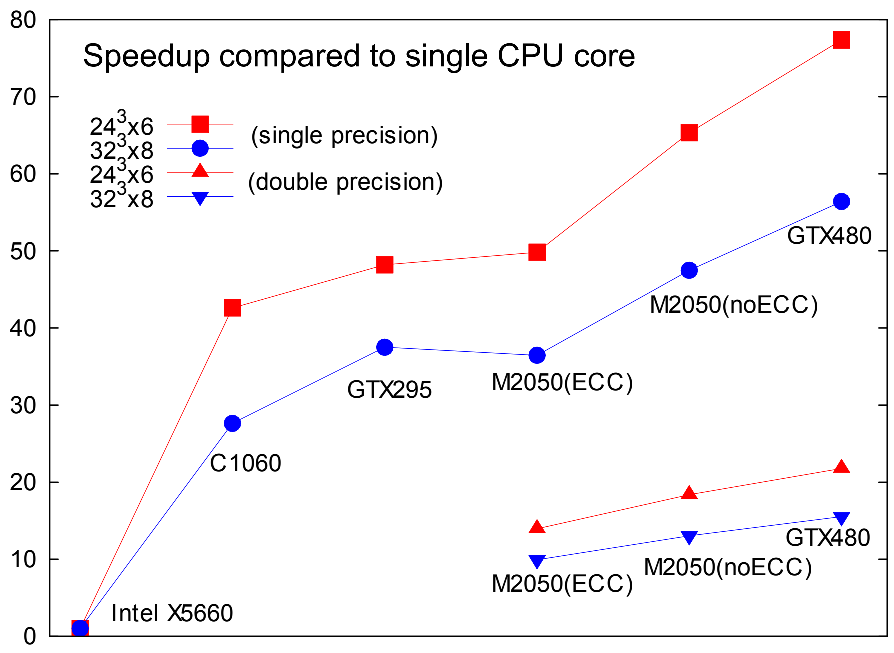
GB/s more important





Lattice QCD performance – GPU vs CPU





Wilson fermion matrix*vector $24^3 \times N_t$

SU(3) matrix stored as 8/12 floats

GF = temporal gauge fixing

[M.A.Clark et al., arXiv:0911.3191]

Staggered fermion matrix*vector

SU(3) matrix stored as 18 floats

Compared to Intel single core

[Bielefeld GPU Group]

Drawbacks of GPUs

- **GPUs are accelerator cards connected via the PCI-bus**

  all data needs to be transferred from Host to Device via the bus

- **You still need a PC and a CPU that is controlling the program**

  hybrid programming Ansatz

- **Mainly two companies on the market:**

  NVIDIA **AMD/ATI**

- **You'll have to rely on their business strategy**

- **Not a long tradition in high performance computing**

  programming extensions and tools are still being developed

- **Most of the hardware details are not known**

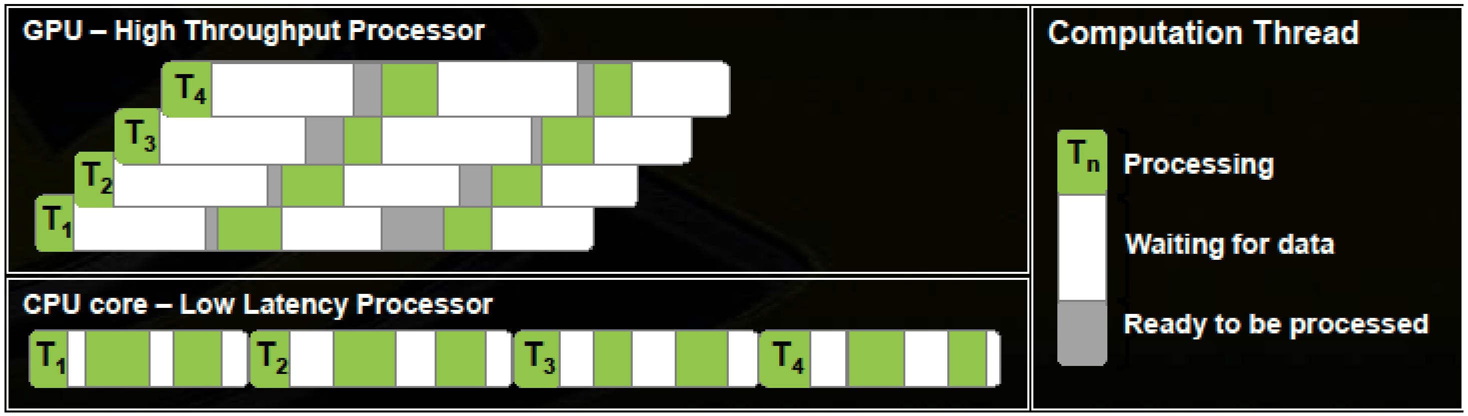
  sometimes it is a black box and one has to rely on their documentation

CUDA

I will only discuss NVIDIA GPUs and CUDA

Compute Unified Device Architecture

- **Extension to C programming language (partly also C++ functionality)**
- **Host functions to access GPU device from the host, e.g. memcpy, kernel calls**
- **Thread based model and easy multi-threading for utilizing the multiprocessors**
- **Hiding latency in memory access with overlapping computations:**



Some Terminology

- ✓ **Thread: concurrent code executed on the CUDA device** (in parallel with others)

  a thread is *the unit of parallelism* in CUDA → SIMT

- ✓ **Warp: a group of threads executed *physically* in parallel (SIMD)**

- ✓ **Thread Block: a group of threads that are executed together**

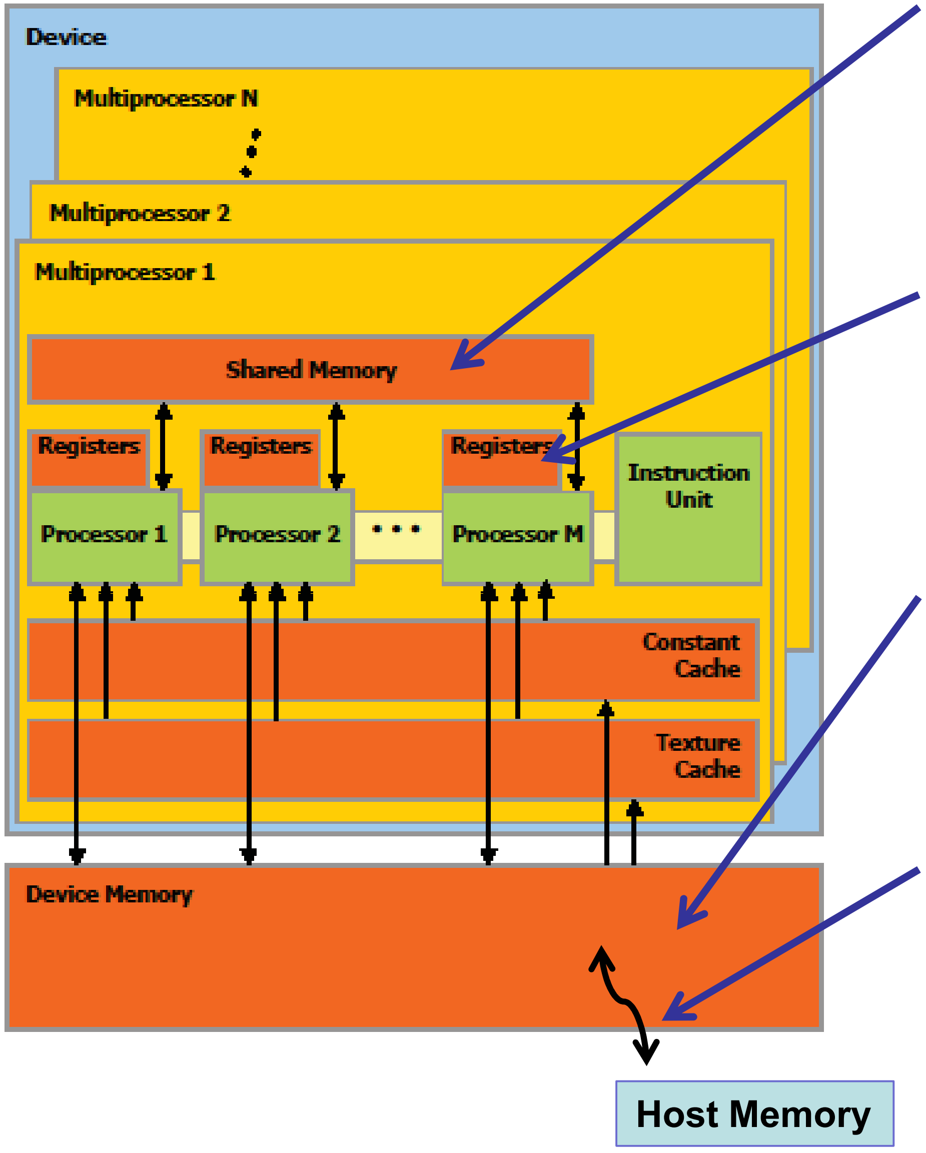
  and can share memory on the same multiprocessor

- ✓ **Grid: a group of thread blocks that are executed *logically* in parallel**

  on all available multiprocessor of the CUDA device

- ✓ **Device: GPU** **Host: CPU** **SM: Streaming Multiprocessor**

Memory Layout (GT200 architecture)



- Threads within a multiprocessor can exchange data using

  16-64 kB shared memory (+L1 cache)

- Large register file:

  8.096-32.768 registers/processor

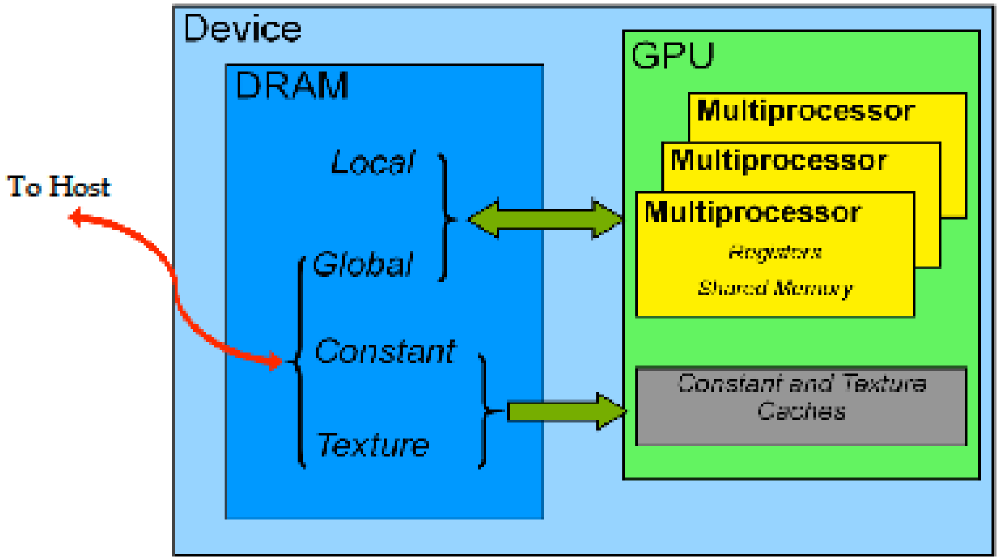
- Device Memory used by all threads

  1.5 GB consumer cards
  3/6 GB Tesla cards (with ECC)

- Memcpy Host <-> Device Memory

  with 3 GB/s a bottleneck

Memory Layout (GT200 architecture)



- Registers and shared memory

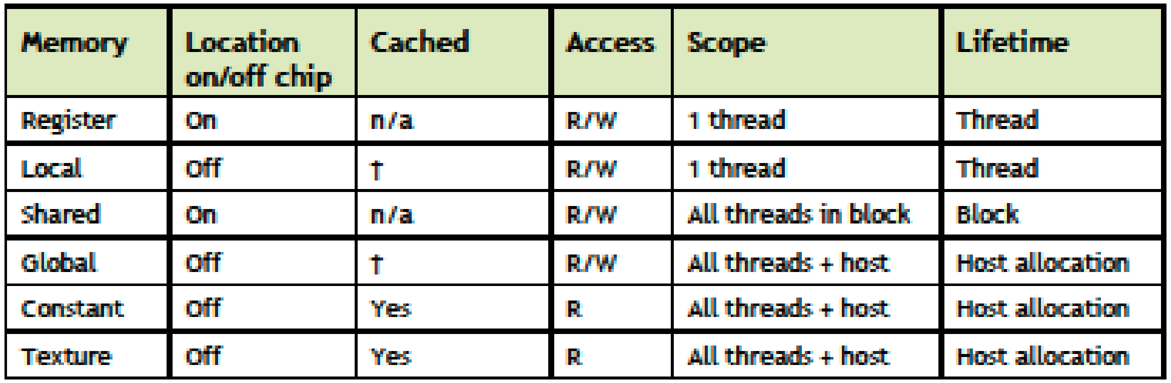
  fastest !!! (latency 1 cycle)

- Device Memory:

  150-200 GB/s but
  shared by all processors
  (latency several 100 cycles)

- Transfer from and to host

  with 3 GB/s a bottleneck



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

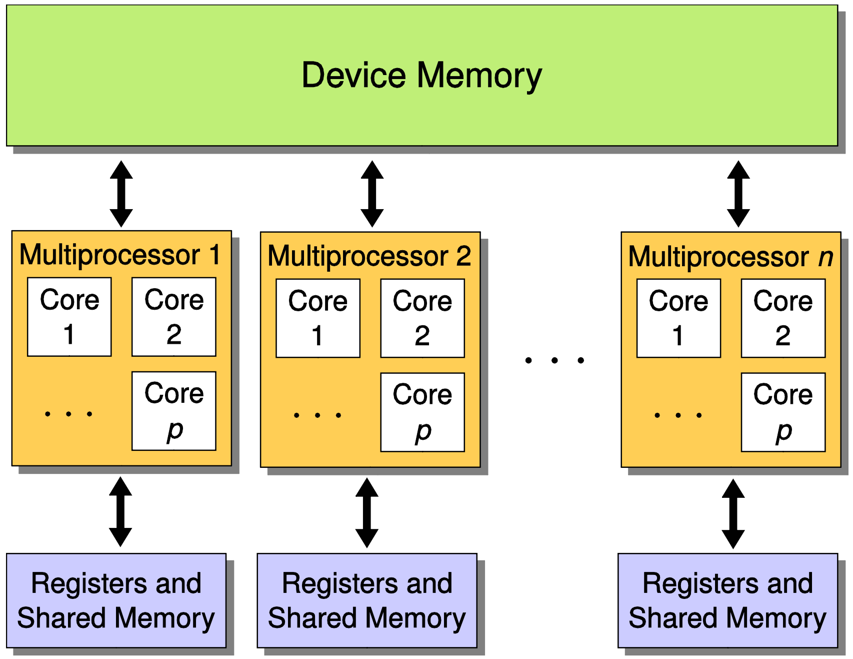
†Cached only on devices of compute capability 2.x.

- Lifetime of shared mem only within a block!

  Communication between all threads only via Global Memory!

- Additional Cache on new GPU models

Memory Layout (GT200 architecture)



- Registers and shared memory

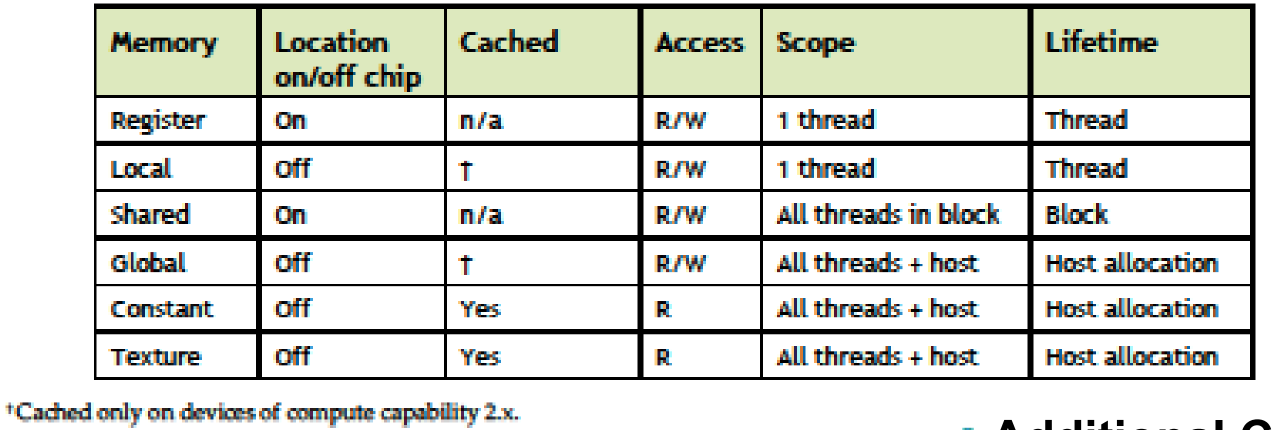
  fastest !!! (latency 1 cycle)

- Device Memory:

  150-200 GB/s but
  shared by all processors
  (latency several 100 cycles)

- Transfer from and to host

  with 3 GB/s a bottleneck



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

†Cached only on devices of compute capability 2.x.

- Lifetime of shared mem only within a block!

  Communication between all threads only via Global Memory!

- Additional Cache on new GPU models

Fermi Streaming Multiprocessor

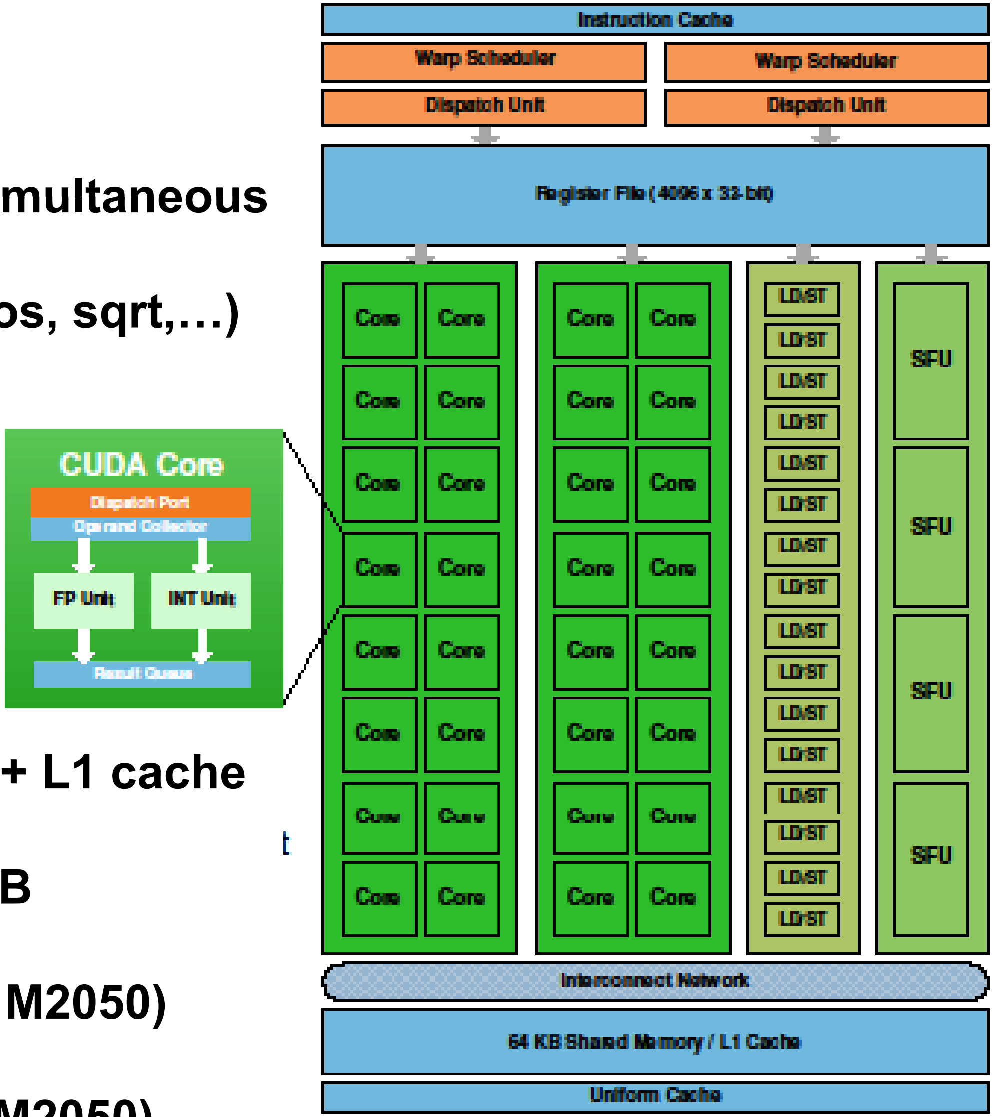
- **32 core per SM** and up to 16 SMs on one GPU
- 16 Load/Store Units

-> 16 Threads can access memory simultaneous

- Four Special Function Units (sin, cos, sqrt,…)
- Large (shared) Register File
- Dual Warp+Instruction Scheduler
- ECC Memory Support on Tesla
- 64kB configurable shared memory + L1 cache

  either 48kB +16kB or 16kB + 48kB

- 515 Gflops double precision (Tesla M2050)
- 1.03 Tflops single precision (Tesla M2050)



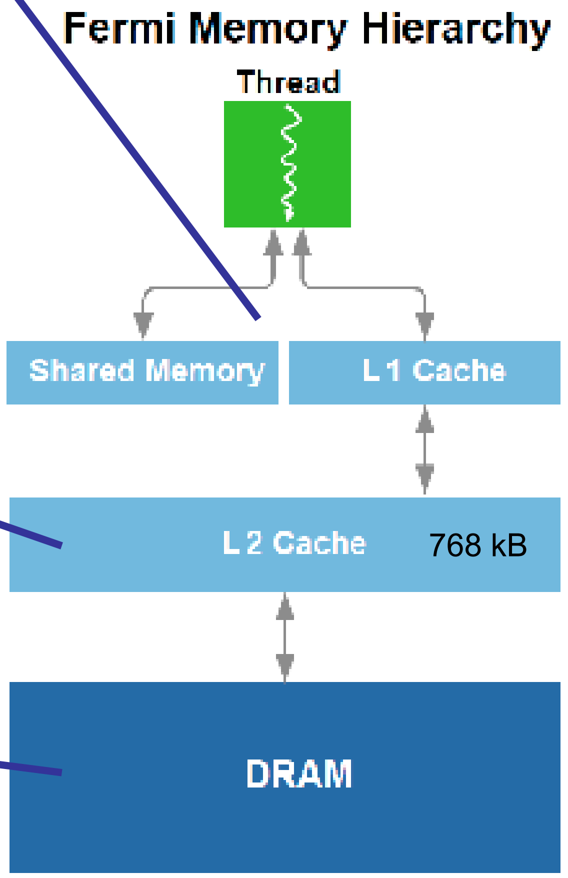
Fermi Streaming Multiprocessor (SM)

Fermi – Memory Layout

- User can decide on the size of Shared Memory vs. L1 Cache

  48 kB Shared Memory + 16 kB L1 Cache

  16 kB Shared Memory + 48 kB L1 Cache



- 768 kB of L2 Cache

- Device Memory:

  1.5 GB on Consumer Cards

  3/6 GB on Tesla Cards

CUDA basics – memory allocation

- **Data needed on the device has to be copied from host**
- **CUDA provides routines to manage the data transfer host <-> device**
- **Allocation of memory on the device (in device memory):**

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

cudaError_t cudaFree(void * devPtr)
```

Example:

```
size_t  size = N * sizeof(float);
float*  h_A, d_A;
h_A = (float *) malloc (size);
cudaMalloc (&d_A, size);
…….
free(h_A);
cudaFree(d_A);
```

h_A is an array on the host

cudaMalloc called on the host

d_A is an array on the device in the global device memory

CUDA basics – copy data to and from device

- **the host process is controlling data transfers**
- **kernels can only operate on device memory**
- **copy data between host and device:**

cudaError_t **cudaMemcpy** (void *dst, void *src, size_t count, *type of transfer*)

(cudaMemcpyHostToDevice / cudaMemcpyDeviceToHost)

Example:

```
cudaMemcpy (d_A, h_a, size, cudaMemcpyHostToDevice);
......... (do some calculations with d_A on the device) ….
cudaMemcpy (h_a, d_a, size, cudaMemcpyDeviceToHost);
```

CUDA basics – Kernel

- device kernel code is defined using __global__ declaration specifier
- device functions are defined using __device__ (host functions using __host__)
- kernels are called by the host but operate on the device
- each kernel has a unique thread ID (see next page)
- all data used by the kernel must be on the device

  i.e. A[ ] and B[ ] are arrays in the global device memory

Example:

```
// Kernel definition – Device code
__global__ vecAdd (const float* A, const float *B, float *C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i<N)
        C[i] = A[i] + B[i];
}
```

CUDA basics – Thread Hierarchy

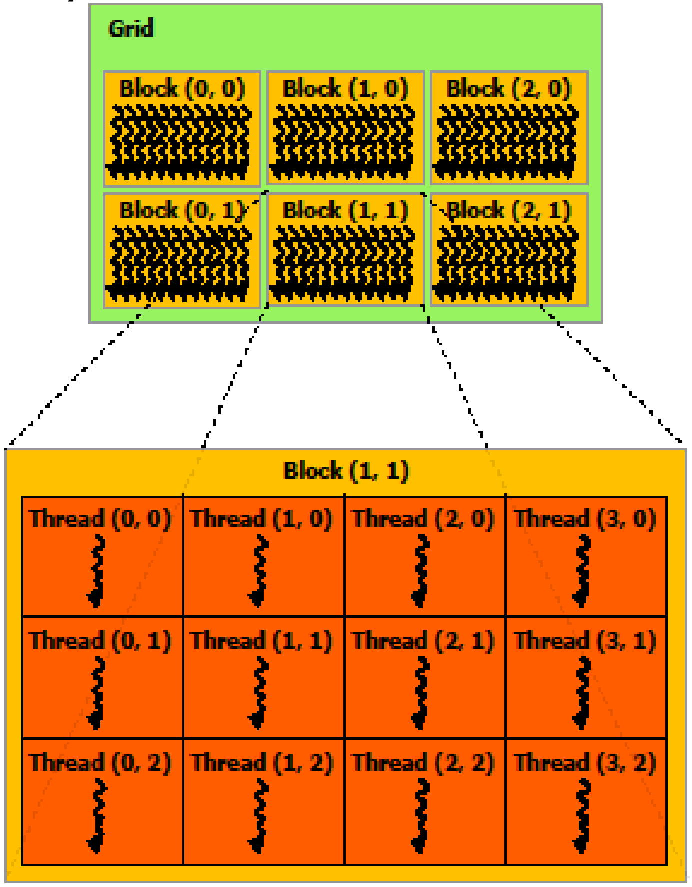
- each thread has a unique thread ID (inside a block)
- threadIdx is 3-component vector

  either one-, two-, or three-dimensional
- maximal number of threads per block: 1024
  - large number optimal for performance
  - they share registers and shared memory
  - 64 or 128 threads usually good balance
- unique ID:

```
blockID.x * blockDim.x + threadId.x
```

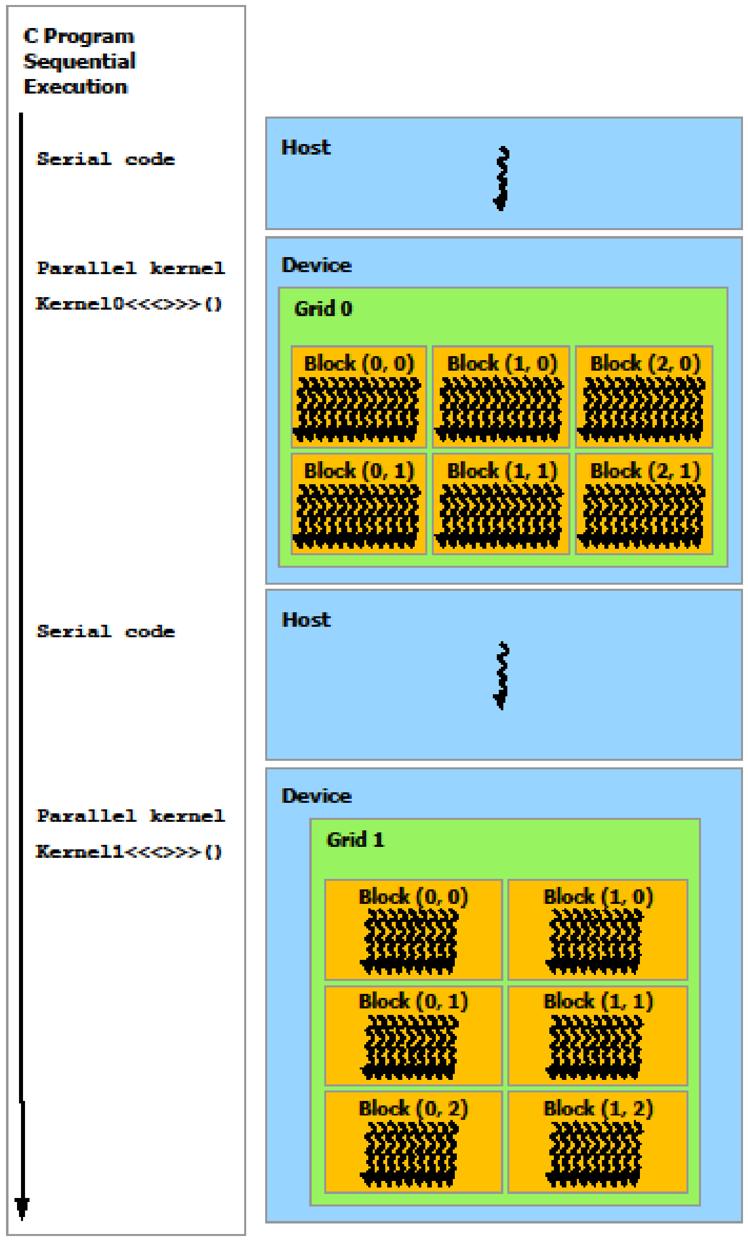
- #blocks and #threads defined in kernel call

```
vecAdd <<< numBlocks, threadsPerBlock >>> (A,B,C,N)
```



CUDA basics – Program flow Host+Device

- control flow of the program is managed by serial code on the host
- the *heavy* calculations are performed on the device
- *continuous* switching between host+device
- synchronization between kernels important



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

CUDA basics – example Vector Addition

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

int main(int argc, char** argv)
{
  float *h_A,*h_B,*h_C,*h_D;
  float *d_A,*d_B,*d_C;

  int N = 1000000;
  size_t size = N * sizeof(float);

  // Allocate input vectors h_A and h_B in host memory
  h_A = (float*)malloc(size);
          .....

  // Initialize input vectors
  RandomInit(h_A, N);
  RandomInit(h_B, N);

  // Allocate vectors in device memory
  cudaMalloc((void**)&d_A, size);
          ......

  // Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Verify result
          .......

  // Free device memory
  cudaFree(d_A);
          .......

  // Free host memory
  free(h_A);
          .......
}
```

Kernel (code running on GPU)

Main program running on CPU

Memory allocation on CPU

Memory allocation on GPU

Copy data Host→Device

Start the kernel

Copy data Host→Device

CUDA basics – example Vector Addition

```
// Variables
float* h_A;
float* h_B;
float* h_C;
float* d_A;
float* d_B;
float* d_C;

// Device code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

int main(int argc, char** argv)
{
    int N = 5000000;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

    // Initialize input vectors
    RandomInit(h_A, N);
    RandomInit(h_B, N);

    // Allocate vectors in device memory
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Verify result
    for (int i = 0; i < N; ++i)
      {
       float sum = h_A[i] + h_B[i];
        if (fabs(h_C[i] - sum) > 1e-5)
          {
           printf("Error in line: %d\n",i);
           break;
          }
      }
```

```
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);
}


// Allocates an array with random float entries.
void RandomInit(float* data, int n)
{
    for (int i = 0; i < n; ++i)
        data[i] = rand() / (float)RAND_MAX;
}
```

CUDA basics – Error handling

- most cuda functions return an error code cudaError_t
- return value on successful execution is cudaSuccess
- cudaError_t cudaGetLastError();

  returns last error from previous execution

- use simpler interface from ‘cuda_utils.h’

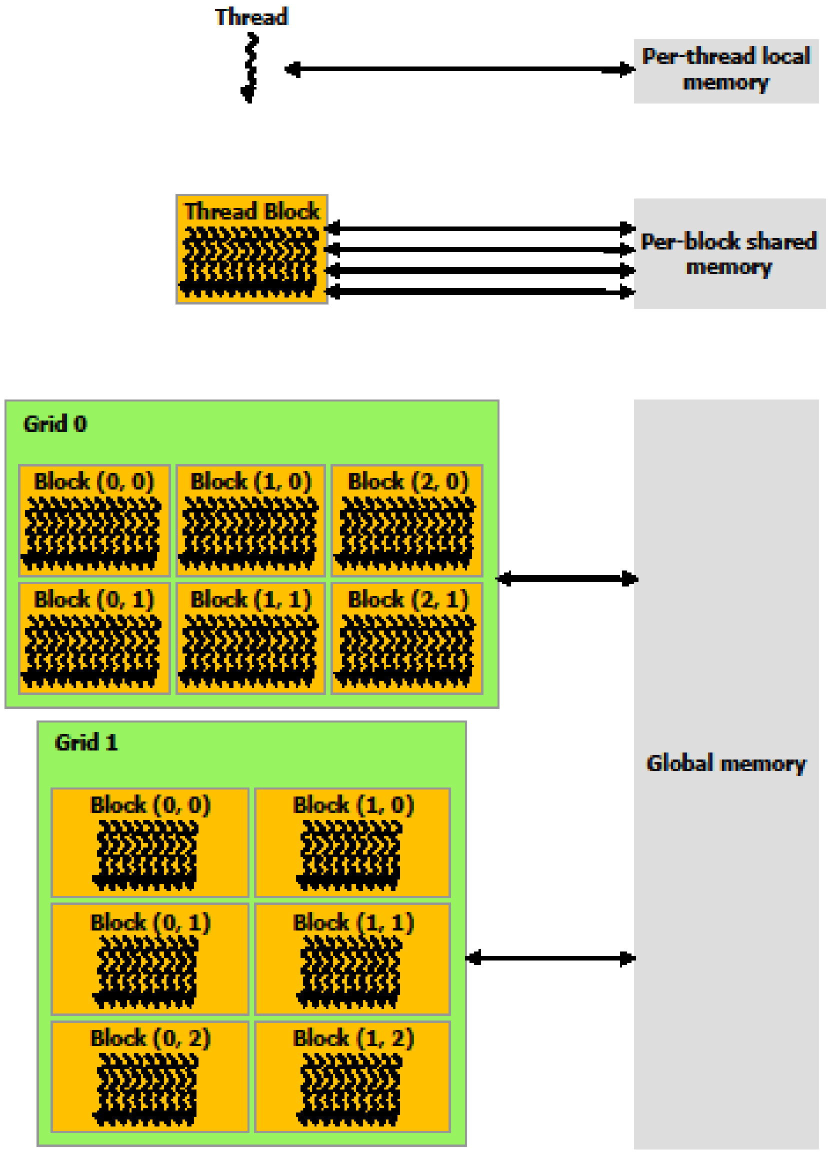
  cu_safecall(x); call x and abort program in case of an error

  e.g. cu_safecall(cudaMalloc(…..));

  cu_safecall_kernel(x); starts kernel x and check for errors

  e.g. cu_safecall_kernel((my_kernel<<<…>>>(..)));

CUDA basics – Thread/Memory Hierarchy



- Fast data access:

  register or shared memory

  and constant+texture memory

- only threads in a block can communicate

  using shared memory

- Slow data access:

  global device memory

- all threads can access all global memory
- but threads are usually not synchronized
- no control on the order of execution

➢ Tipp: reduce multiple access to global memory, use shared memory if possible

CUDA – global /constant/shared memory

- **variables declared inside device functions reside in register or global memory**

  local variables per thread

  if no more registers available moved to local memory

  (=device memory=slow!!!)

- **variables declared as __constant__ (global) are readable by all threads**
- **variables declared as __shared__ reside in shared memory (fast!!!)**

  local per block → most efficient memory access in a block of threads

  as fast as registers (only problem: bank conflicts)

  → used as buffer if data used more than once

  → fast exchange of data between threads in the same block

Example:

```
__global__  kernel  (const float* A, int N){
    __shared__ float  data [size];
    .........
}
```

CUDA – optimizing global memory access

- **coalesced access:**

  memory access of threads in a warp

  can be combined

  into one memory transaction

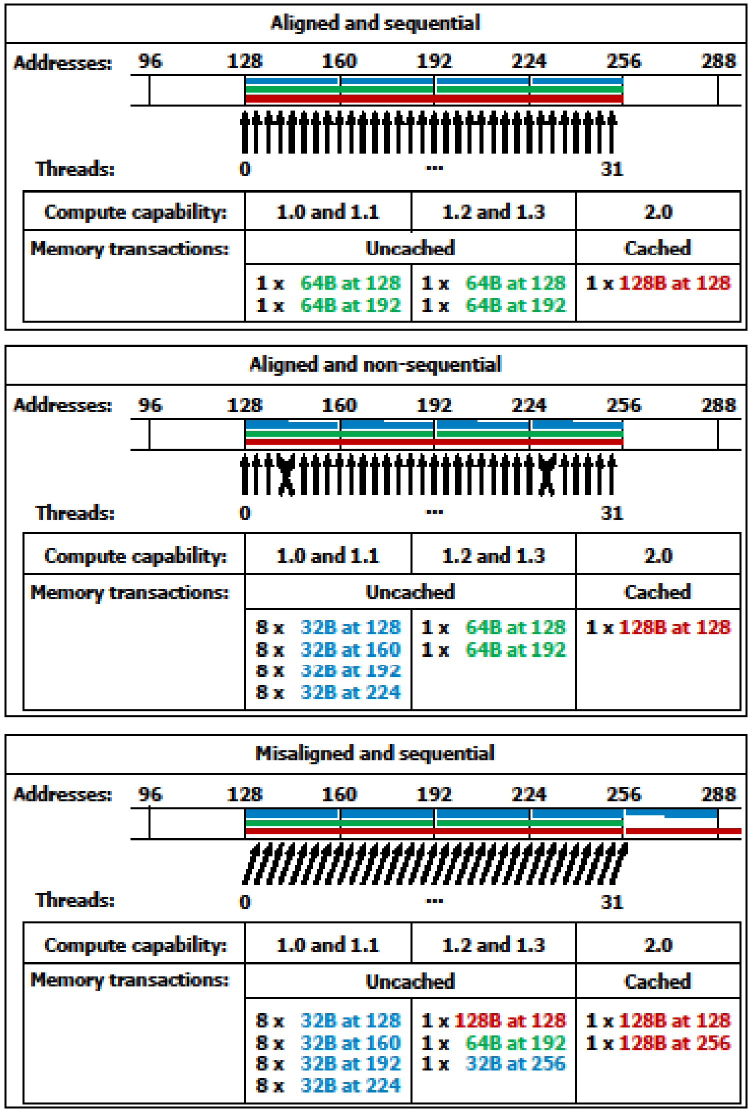
- **cache line size on fermi architecture**

  = 128 bytes (similar behavior as above)

- **data should be aligned and**

  sequentially accessed

  → minimal number of memory transactions


Aligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Aligned and non-sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Misaligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31

Aligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

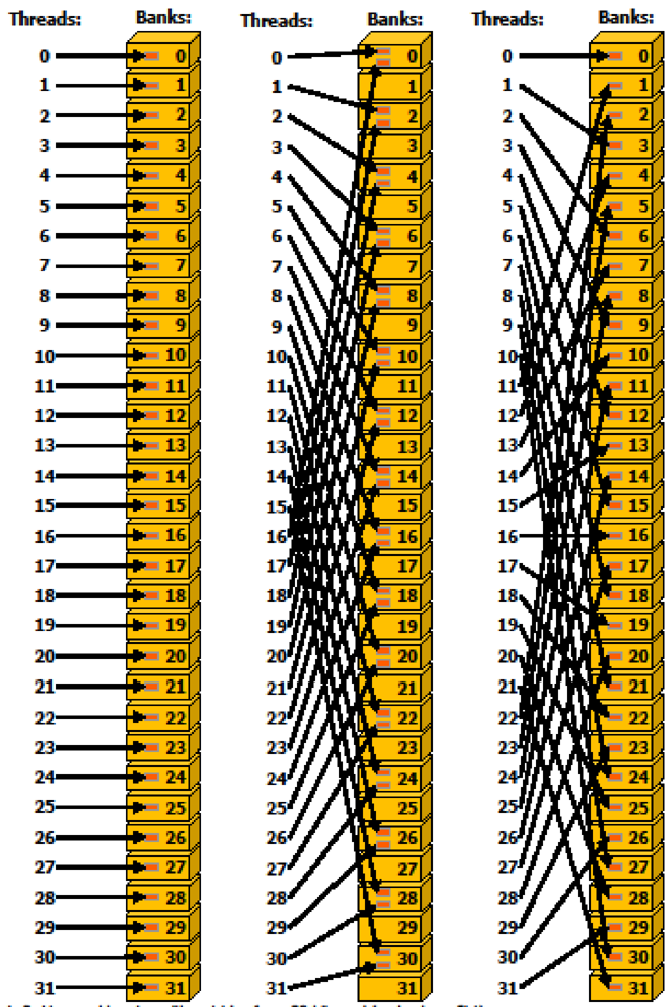
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

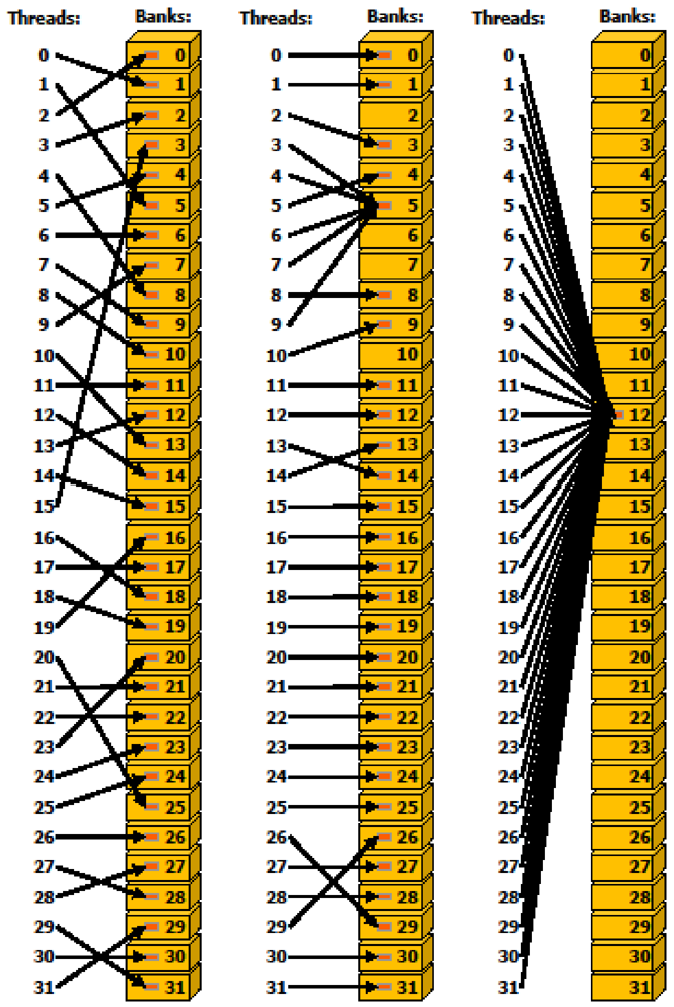
Figure G-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

CUDA – optimizing shared memory access



Left: Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Figure F-2 Examples of Strided Shared Memory Accesses for Devices of Compute Capability 2.x



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).

Figure F-3 Examples of Irregular and Colliding Shared Memory Accesses for Devices of Compute Capability 2.x

Lessons for Lattice QCD

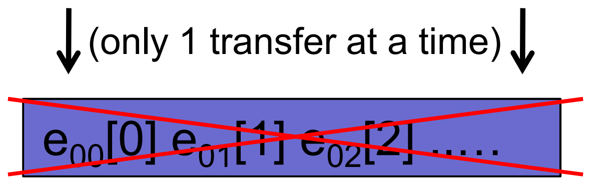
- split your code into small threads

  usually computation of one lattice site per thread

- optimize memory access: coalesced access

  ~~do not use arrays of structures~~

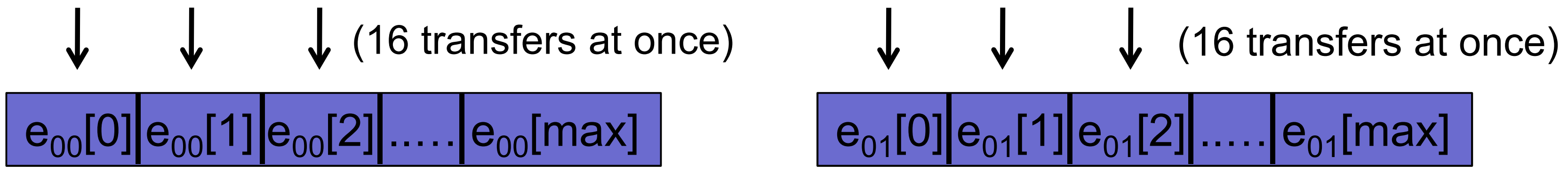
  ~~`su3   gauge_field[nr_links]`~~



(only 1 transfer at a time)
$e_{00}[0]$ $e_{01}[1]$ $e_{02}[2]$ .....

  use arrays for each element of SU(3) instead:

  `complex su3_e00[nr_links], su3_e01[nr_links] ……`



(16 transfers at once)
$e_{00}[0]$ | $e_{00}[1]$ | $e_{00}[2]$ | ..... | $e_{00}[max]$
(16 transfers at once)
$e_{01}[0]$ | $e_{01}[1]$ | $e_{01}[2]$ | ..... | $e_{01}[max]$

- reduce register pressure by using shared memory
- increase flops/bytes, i.e. reduce memory access

  use SU(3) reconstruction, e.g. just store 2 rows = 12 floats

Exercises: Bielefeld GPU System

$$\begin{tikzcd}
 & \text{Login Server: \underline{bam2.physik.uni-bielefeld.de}} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{\underline{tesla1}} & \text{\underline{gpu2}} & \text{\underline{gpu1}}
\end{tikzcd}$$

tesla1	gpu2	gpu1
2xNvidia M2050	4xNvidia GTX480	4xNvidia GTX285
448 Cores	480 Cores	240 Cores

$$\underbrace{\underbrace{\text{gpu.q@tesla1} \qquad \text{gpu.q@gpu2}}_{\text{fermi.q}} \qquad \text{gpu.q@gpu1}}_{\text{gpu.q}}$$

- submit your job using: *qsub queue_script.sh*

(example script in /home/gpu)

Exercise 1 – Vector Addition

use the program ‘*vectorAdd.cu*’

- compile: nvcc –O3 –o vector_Add vector_Add.cu
- vary the number of threads per block and plot the kernel runtimes

  what is the optimal number of threads per block?
- estimate the memory bandwidth and flops for the optimal parameters
- use an offset in the array indices,

  i.e. store data in A[offset]… A[N+offset-1] and calculate A[i+offset]+B[i+offset]

  for which values of offset do you observe coalesced access?
- reduce the number of threads by calculating more than 1 sum per tread

Exercise 2 – Scalar product

Write a program to compute the scalar product of two vectors

- **step1 : use one kernel to compute the product A[i]*B[i] for all i**
- **step2 : use one kernel for partial sums inside thread blocks**

  (optimization: use shared memory for the partial sums)
- **step3 : reduce the partial sums for the final result**

Note:

- **separate kernels required as steps depend on previous results**
- **synchronization important between the steps**
  - **cudaThreadSynchronize(); between kernels**
  - **__syncthreads(); inside kernel to sync in a block**
- **intermediate results must be written to global memory**
- **see /home/gpu/reduction.pdf**
- **see ‘*cuda_reduce.hpp*’ (taken from CUDA SDK)**

(Advanced) Exercise 3 – Matrix multiplication

Write a program to compute C=A*B with A,B being NxN matrices

- step1 : each thread computes $C_{ij} = \sum_{\kappa} A_{ik} \times B_{kj}$

  note: each element is read N times from global memory

- step2 : each block calculates a MxM subregion of C
- step3 : optimization: read MxM subregion of A and B to shared memory

  and do the computation only using shared memory

- compare the kernel runtime of the 3 versions
- compare step2 and 3 on different architecture

  Is shared memory still faster than the cache

  architecture of Fermi?

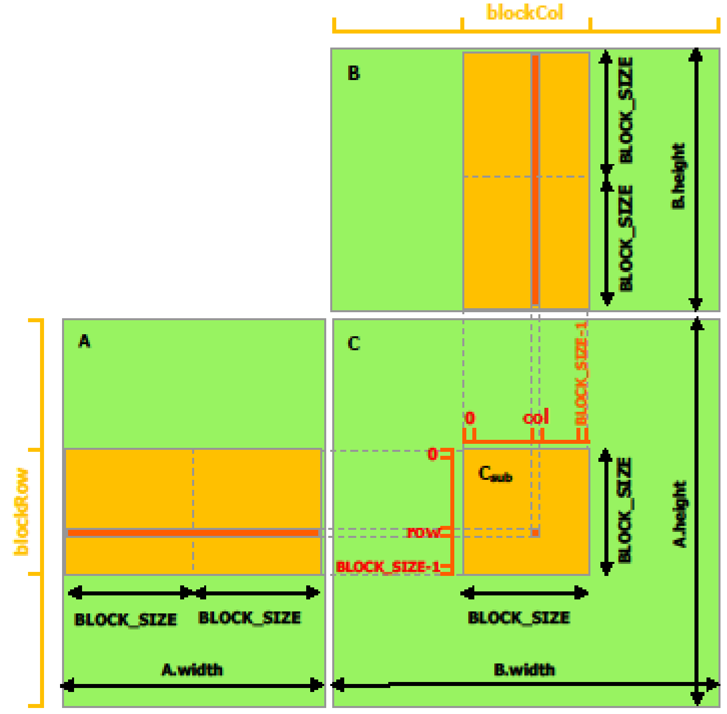


Figure 3-2. Matrix Multiplication with Shared Memory

Photos from the Canoe Excursion online:

http://www2.physik.uni-bielefeld.de/strongnet2011.html