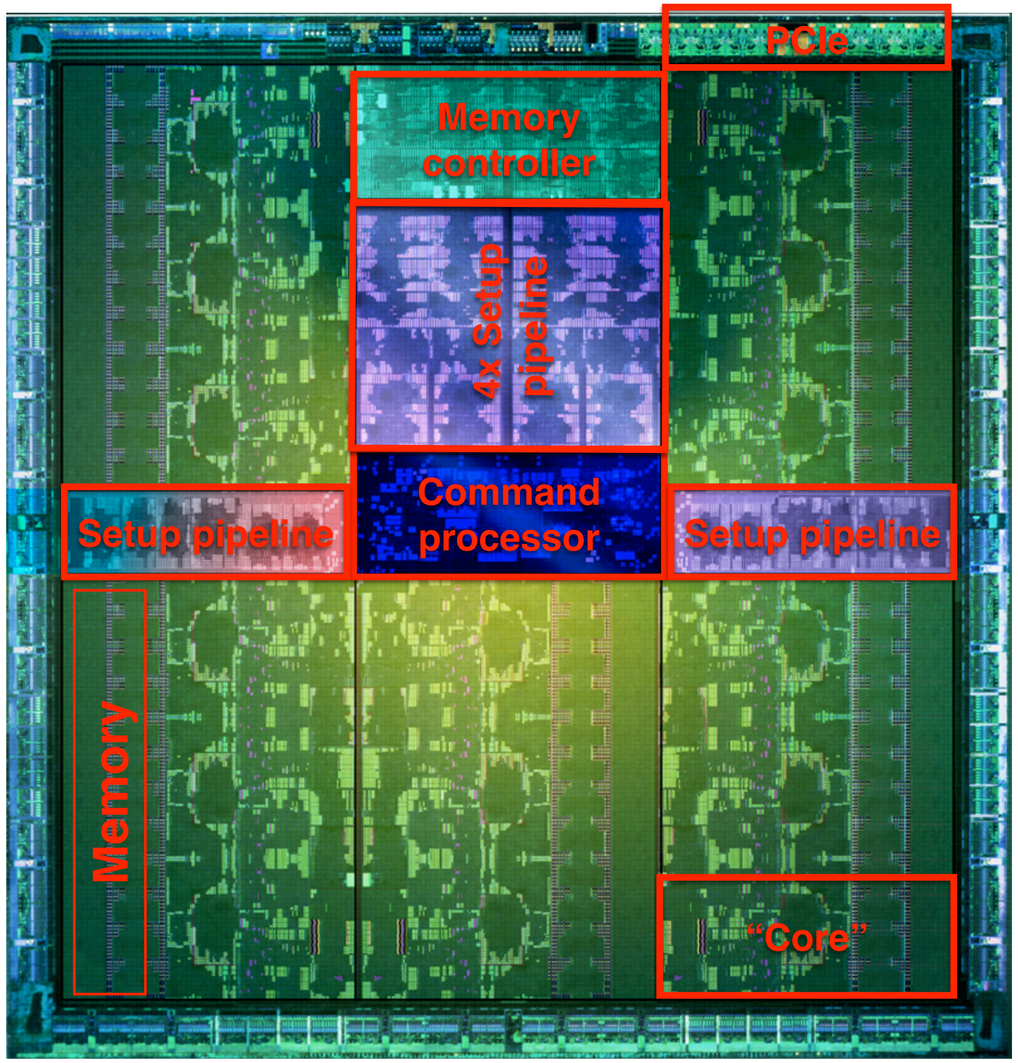
GPU COMPUTING
LECTURE 02 - CUDA PROGRAMMING

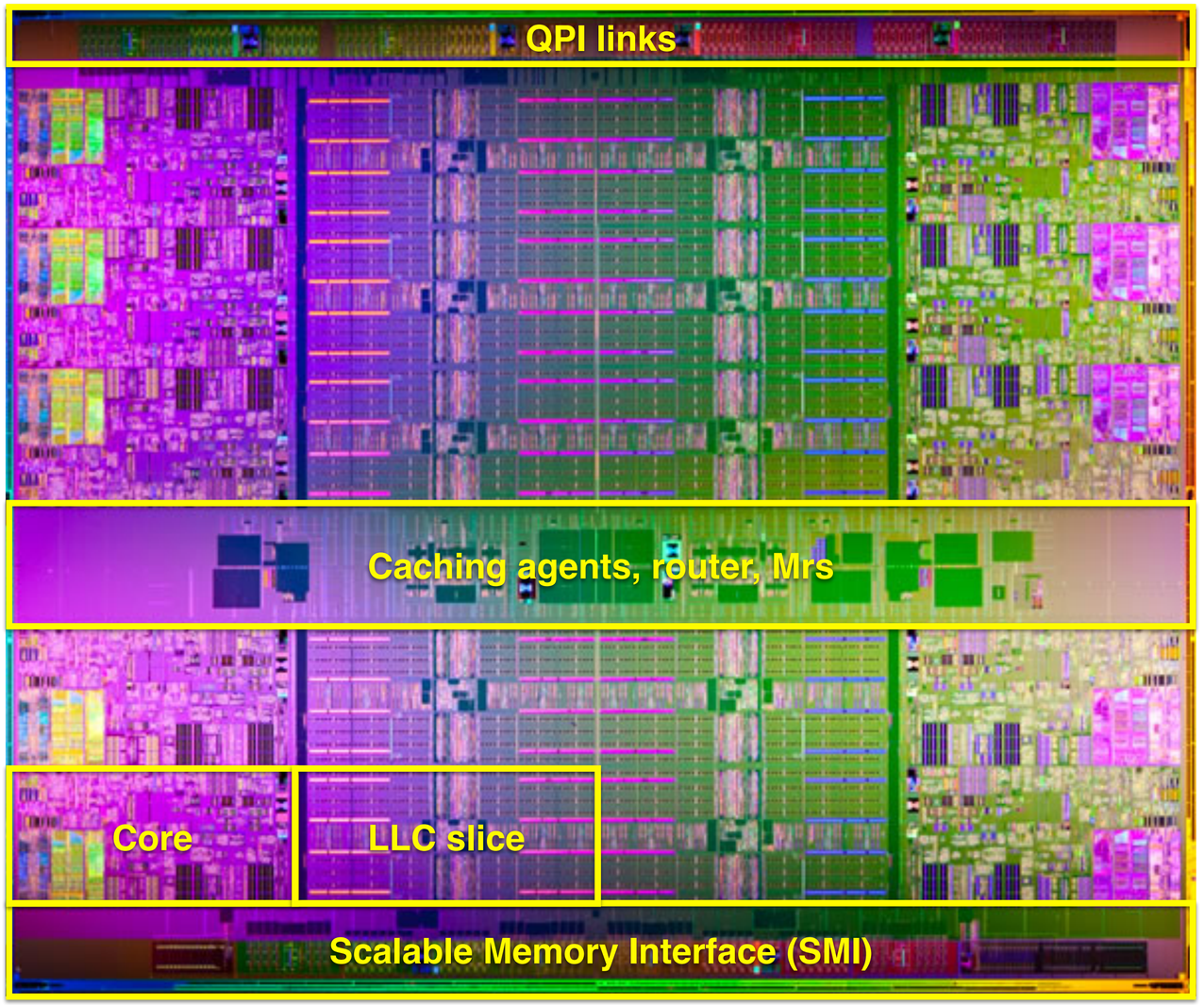
Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

With material from D. Kirk, W. Hwu ("Programming Massively Parallel Processors")

DIE SHOTS - CPU OR GPU?



NVIDIA Kepler- GK110



Intel Xeon E7 - Westmere-EX

MAIN DIFFERENCES BETWEEN GPU AND CPU

Parallelism: SISD, SIMT, SIMD

Usage: latency vs. throughput

Accessible memory capacity

Performance (specially per watt)

CUDA & GPU - OVERVIEW

NVIDIA CUDA

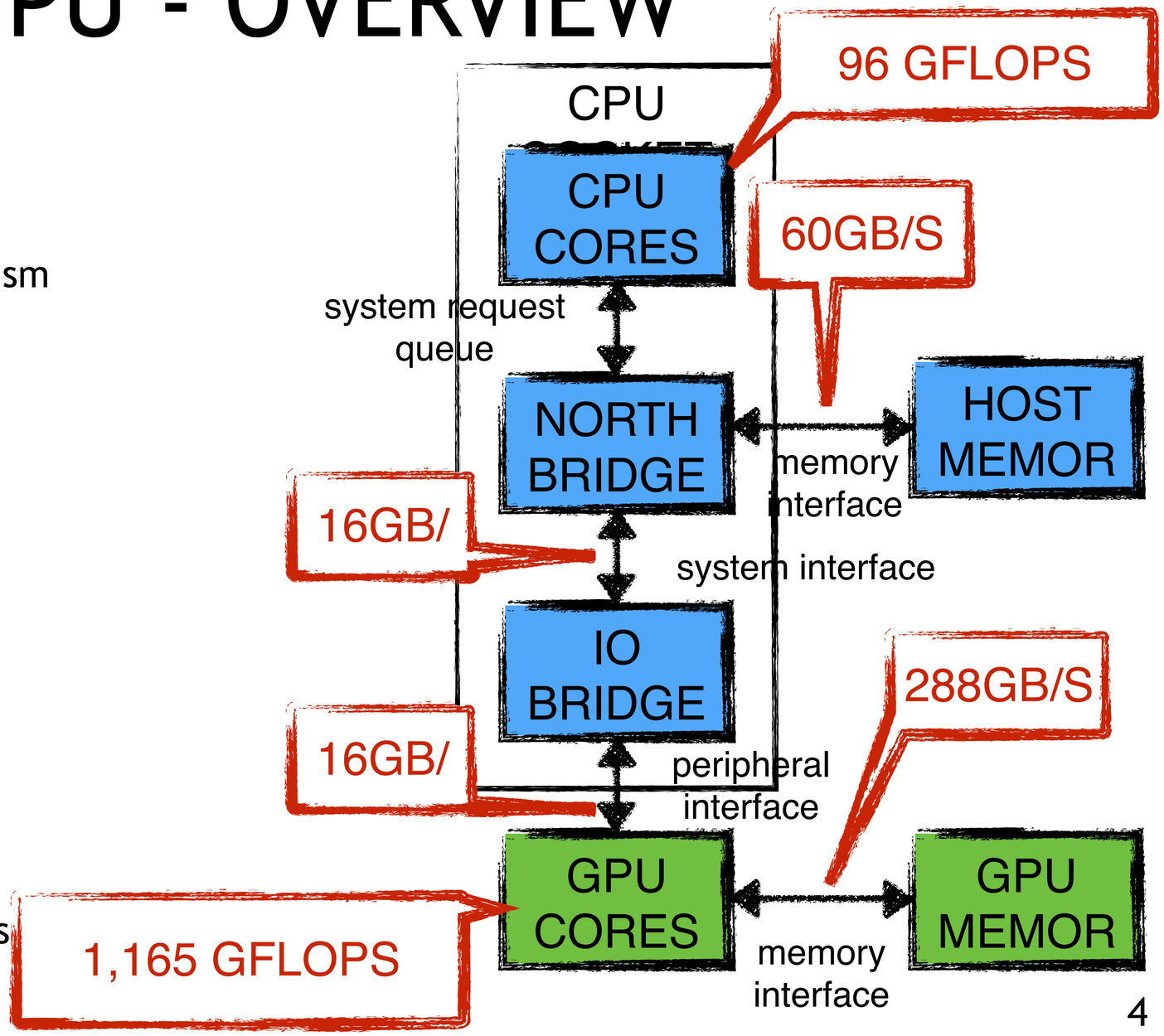
- Compute kernel as C program
- Explicit data- and thread-level parallelism
- Computing, not graphics processing
- Host communication

Memory hierarchy

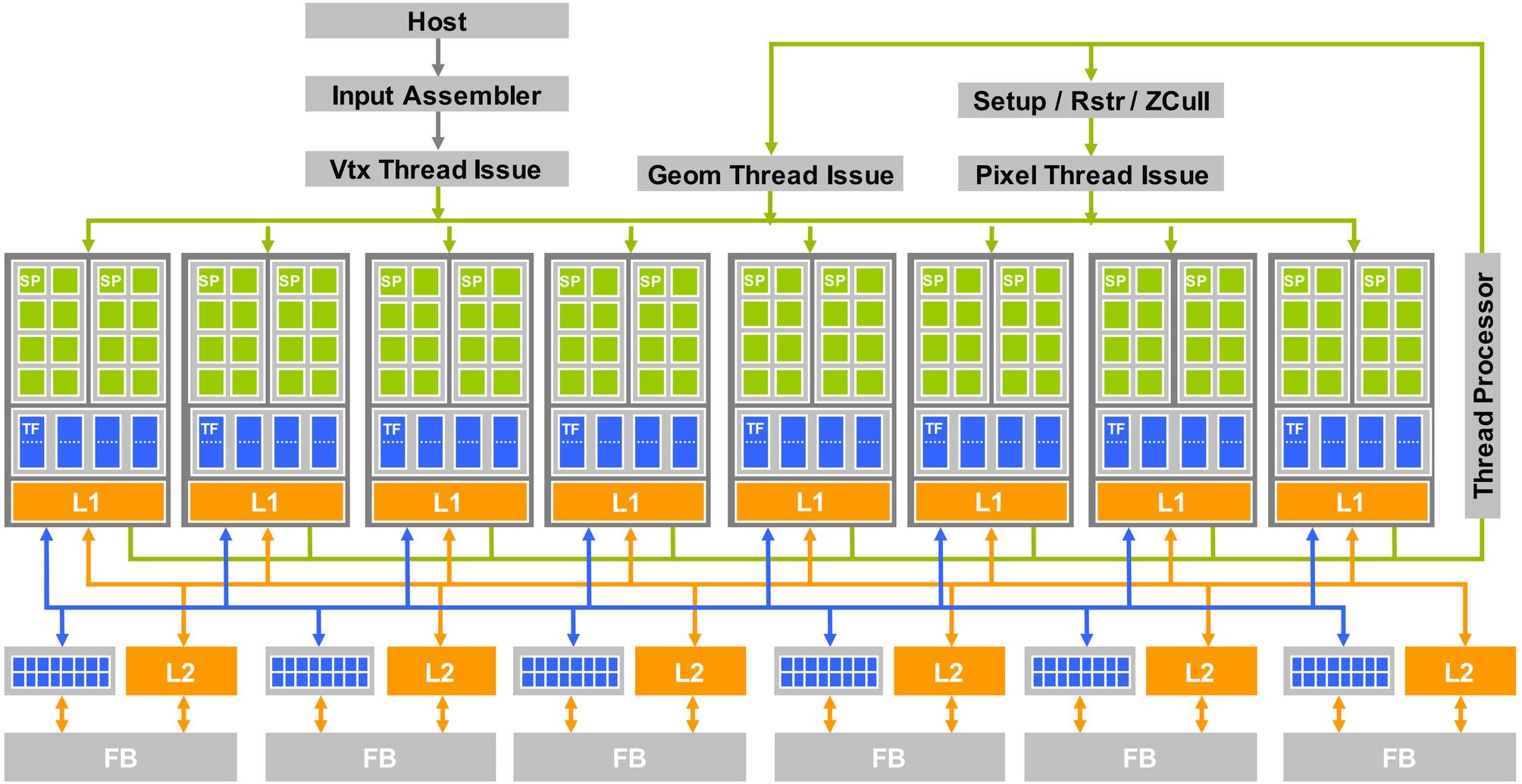
- Host memory
- GPU (device) memory
- GPU on-chip memory (later)

More HW details exposed

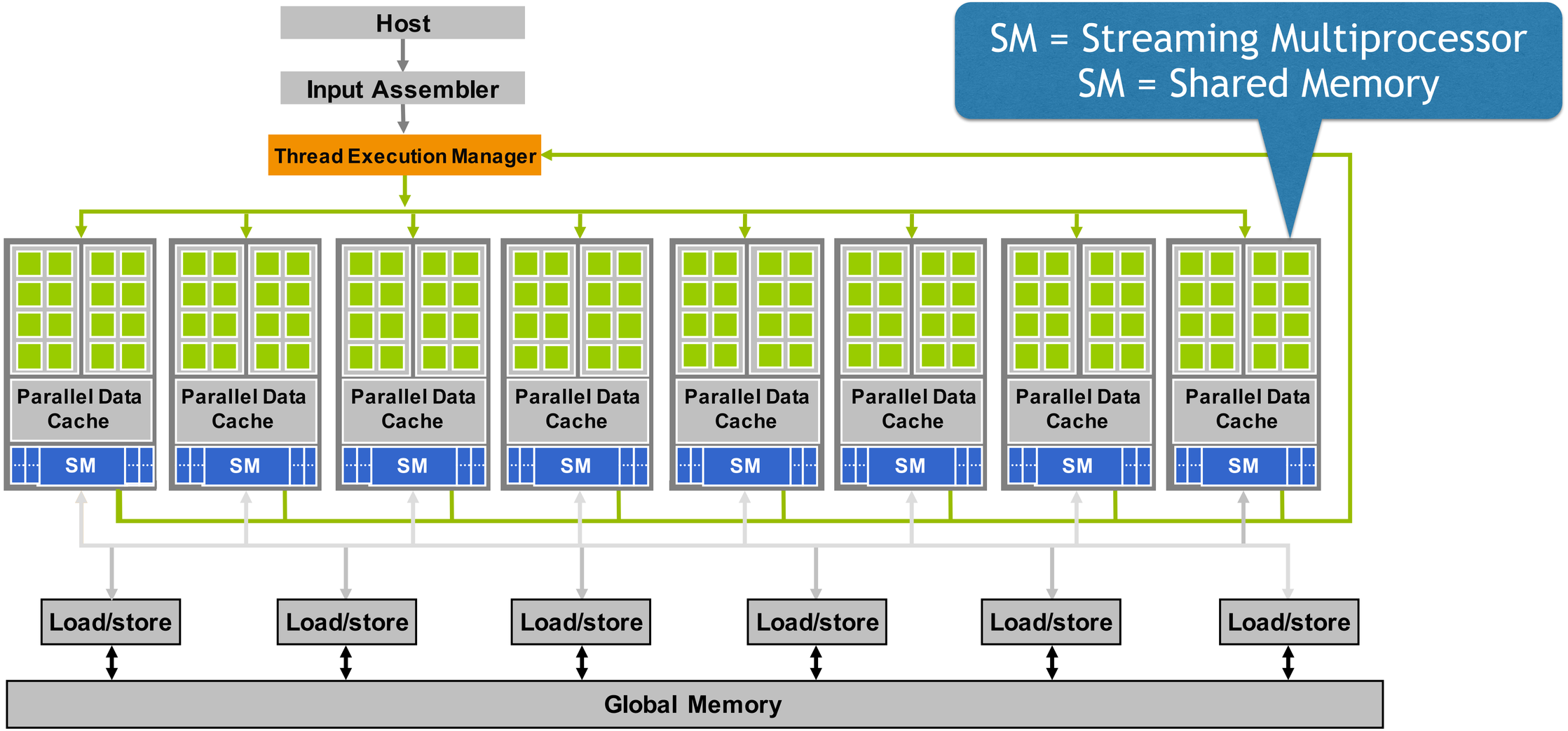
- Use of pointers
- Load/store architecture
- Barrier synchronization of thread blocks



G80 ARCHITECTURE FOR GRAPHICS PROCESSING



G80 ARCHITECTURE FOR GENERAL-PURPOSE PROCESSING



CUDA PROGRAMMING MODEL

PROGRAMMING MODEL

CUDA program consists of CPU & GPU part

- CPU part: part of the program with no or little parallelism
- GPU part: high parallel part, SPMD-style

Concurrent execution

- Non-blocking thread execution
- Explicit synchronization

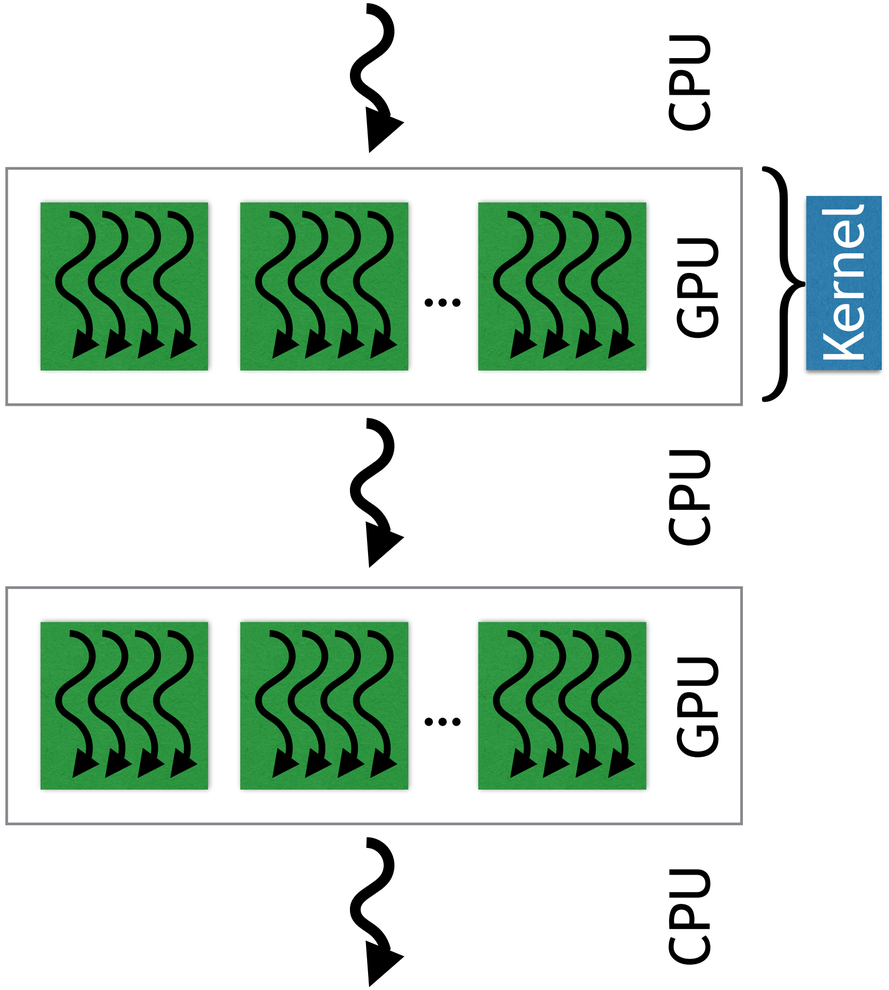
C Extension with three main abstractions

1. Hierarchy of threads
2. Shared memory
3. Barrier synchronization

Exploiting parallelism

- Fine-grain data-level parallelism (DLP)
- Thread-level parallelism (TLP)

} Inner loops, Threads, Kernels



KERNEL LAUNCH

Kernels: N-fold execution by N threads

```
__global__
```

Execution:

```
kernel <<< numBlocks,
           threadsPerBlock >>>
           (args)
```

Unique ID

```
threadIdx.{x,y,z}
```

Control flow for SPMD programs

Memory access orchestration

```
__global__ void matAdd ( float A[N][N],
                         float B[N][N],
                         float C[N][N] )
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  // Kernel invocation
  dim3 dimBlock ( N, N );
  matAdd <<< 1, dimBlock >>> ( A, B, C );
}
```

KERNEL LAUNCH

Each thread block has up to 3 dimensions

- “Block” in the following

Number of blocks is limited

- 512 x 512 x 64 -> 1024 x 1024 x 64
- GPU dep.

Additional hierarchy level: grid = multiple blocks

- Grid = kernel in execution
- Unique ID `blockIdx`, up to 3 dimensions
- Blocks are executed independently and implementation-dependent

```
__global__ void matAdd ( float A[N][N],
                         float B[N][N],
                         float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  // Kernel invocation
  dim3 dimBlock ( N, N );
  matAdd <<< 1, dimBlock >>> ( A, B, C );
}
```

up to 3D

KERNEL LAUNCH

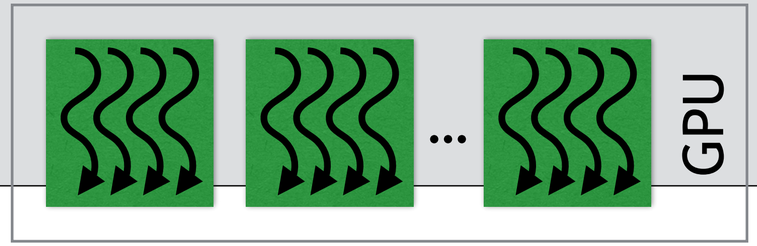
```
__global__ void matAdd ( float A[N][N],
                         float B[N][N],
                         float C[N][N] )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if ( i < N && j < N )
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  // Kernel invocation
  dim3 dimBlock ( 16, 16 );
  dim3 dimGrid  ( ( N + dimBlock.x - 1 ) / dimBlock.x,
                  ( N + dimBlock.y - 1 ) / dimBlock.y );
  matAdd <<< dimGrid, dimBlock >>> ( A, B, C );
}
```

Super-fine grained: one thread computes one element

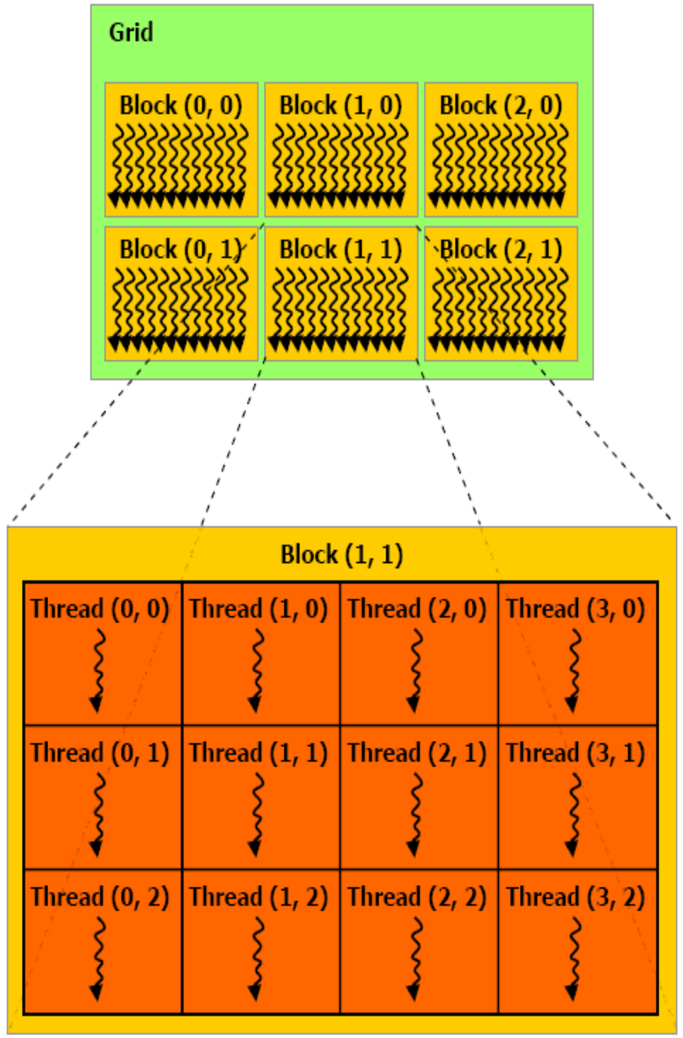
Operator “/” rounds down, so add block size to round up!
E.g. N=50
grid size = (50+16-1)/16=4.0625 => 4

grid size

block size



THREAD HIERARCHY



- Thread hierarchy
  - Grid of thread blocks
  - Blocks of equal size
- Given problem size N, how to choose the parameter threads per block respectively blocks per grid?
- Recommendations wrt block count
  - >2x number of SMs
  - Optimal: 100 – 1000 (max. 64k-1)
- Recommendations wrt threads/block
  - Required concurrency for latency toleration vs resources per thread

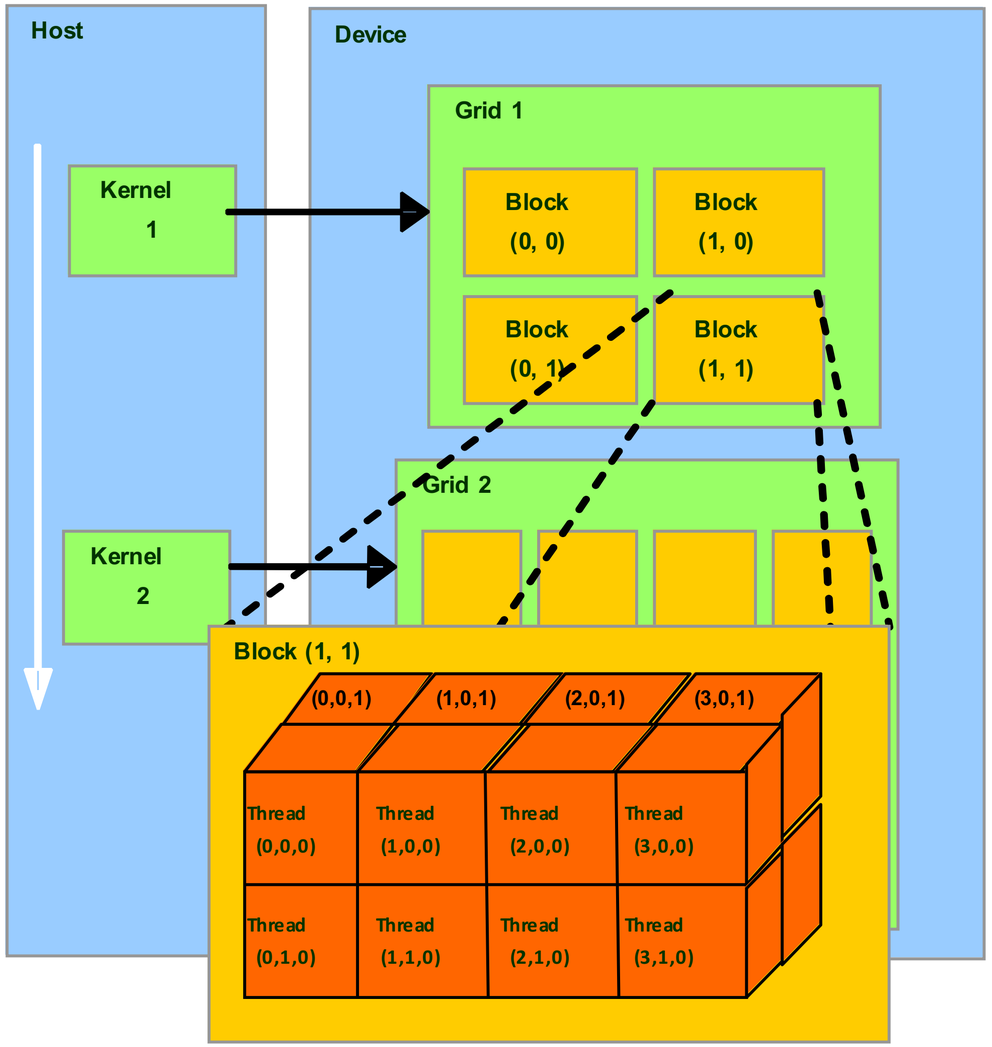
THREAD COMMUNICATION

Communication and synchronization only within one thread block

- Shared memory
- Atomic operations
- Barrier synchronization

Threads from different blocks cannot interact

- Exception: global memory
- Very weak coherence & consistency guarantees



MEMORY HIERARCHY – GLOBAL MEMORY

Global memory

- Communication between host and device
- Accessible from all threads (R/W)
- High latency
- Lifetime exceeds thread lifetime
- Sensitive to fine-grained accesses

Allocation

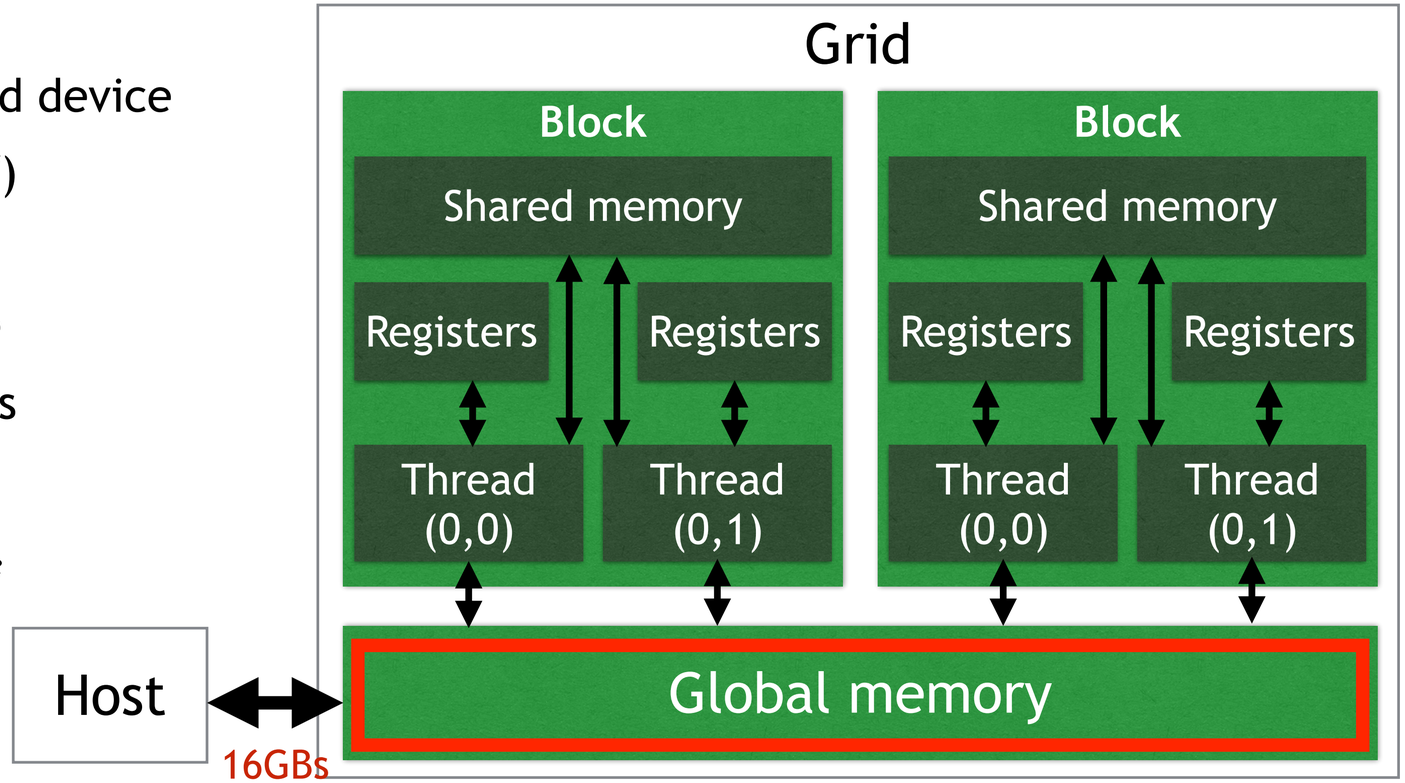
```
cudaMalloc (&dmem, size);
```

Deallocation

```
cudaFree (dmem);
```

Data transfer (blocking)

```
cudaMemcpy (*dst, *src, size, transfer_type);
cudaMemcpyAsync ( … )
```



MEMORY HIERARCHY – GLOBAL MEMORY

Annotate variable scope!

Only references to device memory

```
void *dmem = cudaMalloc ( N*sizeof ( float ) ); // Allocate GPU memory

void *hmem = malloc ( N*sizeof ( float ) );     // Allocate CPU memory

// Transfer data from host to device
cudaMemcpy ( dmem, hmem, N*sizeof ( float ), cudaMemcpyHostToDevice );

// Do calculations
kernel1 <<< numBlocks, numThreadsPerBlock >>> ( dmem, N );
...
kernel2 <<< numBlocks, numThreadsPerBlock >>> ( dmem, N );

// Transfer data from device to host
cudaMemcpy ( hmem, dmem, N*sizeof ( float ), cudaMemcpyDeviceToHost );

cudaFree ( dmem );  // Free device buffer
free ( hmem );      // Free host buffer
```

MEMORY HIERARCHY – SHARED MEMORY

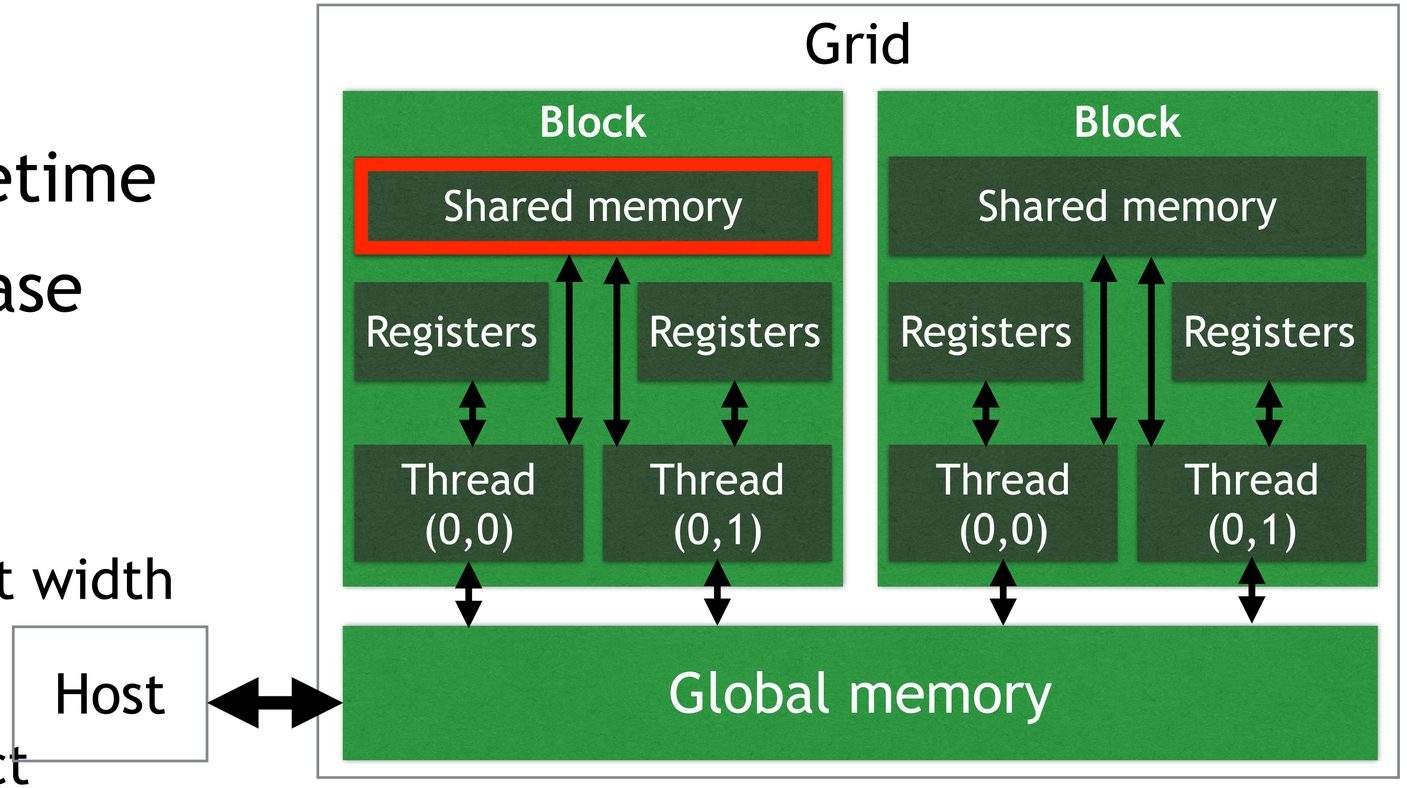
On-chip memory

Lifetime: thread block lifetime

Access costs in the best case equal register access

Organized in n banks

- Typ. 16-32 banks with 32bit width
- Low-order interleaving
- Parallel access if no conflict
- Conflicts result in access serialization



VARIABLE DECLARATION

	Location	Access from	Lifetime
`__device__ float var;`	global memory (device memory)	device/host	program
`__constant__ float var;`	constant memory (device memory)	device/host	program
`__shared__ float var;`	shared memory	threads	thread block
`texture <float> ref;`	texture memory (device memory)	device/host	program

`__device__` can be combined with others (e.g., `__constant__`)

Shared memory & consistency model

`__syncthreads` to wait for completion of outstanding write operations

Unconstrained completion of read/write operations (exception: `volatile`)

TYPE SPECIFIERS

Vector types

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, **float4**, double2

Derived from basic types (int, float, …)

Dimension type: dim3

Based on uint3

Unspecified components are initialized with 1

FUNCTION DECLARATION

	Executed on	Callable from
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

`__global__` defines a kernel (return type: void)

`__host__` is optional

`__host__` and `__device__` can be combined

No pointers to `__device__` functions (exception: `__global__` functions)

For functions that are executed on the GPU:

- No recursions
- Only static variable declarations
- No variable parameter count

JUST-IN-TIME COMPILATION

Device code only supports C-subset of C++ (getting better)

Compile with nvcc

- Compiler Driver
- Calls other tools as required
- cudacc, g++, clang, …

Output

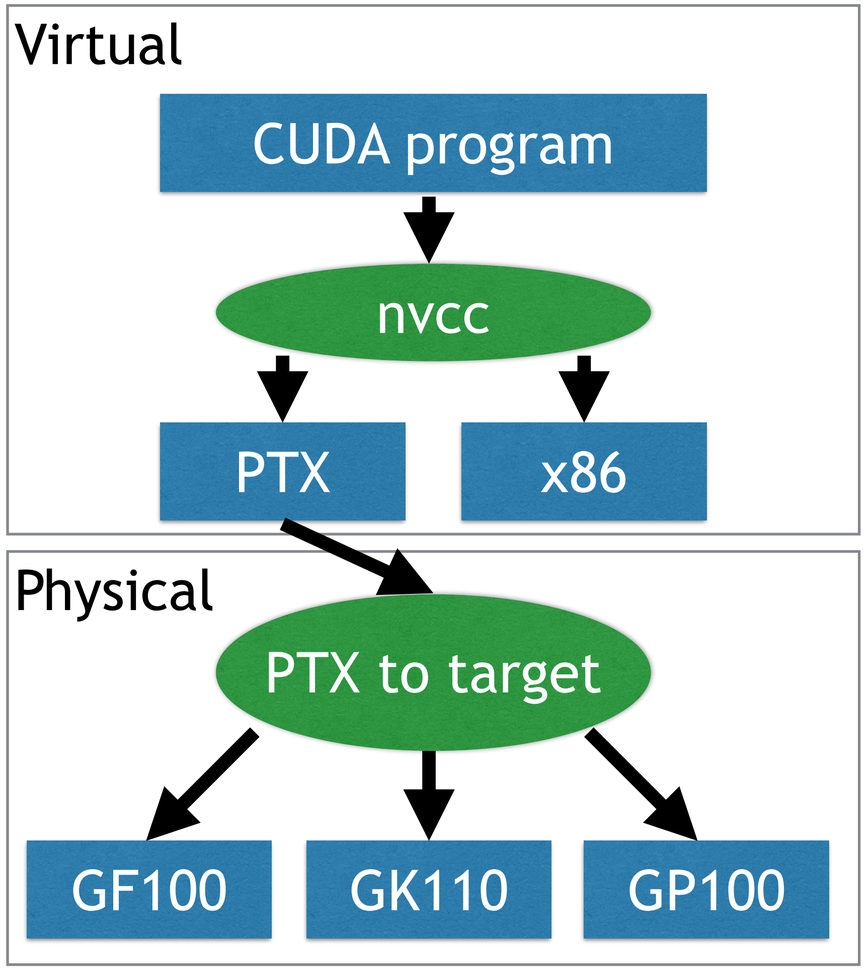
- C code (host CPU Code)
- Either PTX object code, or source code for run-time interpretation

PTX (Parallel Thread Execution)

- Virtual Machine and ISA
- Execution resources and state

Linking

- CUDA runtime library cudart
- CUDA core library cuda



BRIEF PROPERTY SURVEY (DEVICEQUERY)

Model	CC Revi-sion	Total global memory [bytes]	Multi-proces sors	Cores	Total constant memory [bytes]	Shared memory per block [bytes]	Regis-ters per block	Warp size	Threads per block	Max dimen-sion of a block	Max. dimen-sion of a grid	Max. memory pitch [bytes]	Clock rate [GHz]	Concurrent copy and execution
GeForce GTX 480	2,0	1.5G	15	480	64k	48k	32k	32	1k	1k x 1k x 64	65535 x 65535 x 65535	2G	1,4	Y 1
Tesla K20c	3,5	5G	13	2496	64k	48k	64k	32	1k	1k x 1k x 64	2G x 65535 x 65535	2G	0,7	Y 2
RTX 2080Ti	7,5	11G	68	4352	64k	48k	64k	32	1k	1k x 1k x 64	2G x 65535 x 65535	2G	1,54	Y 3

CUDA EXAMPLE: SAXPY

SAXPY EXAMPLE

$$y[i] = alpha \cdot x[i] + y[i]$$

SAXPY: Scalar Alpha X Plus Y

Simple test to compare GPU and CPU performance

- Objective: runtime reduction
- Max. gridSize * threadsPerBlock elements
- 65535*1k -> ~ 64M elements
- Memory requirement = 32M elements * 2 arrays * 4 Byte/element = 256MB

Source code contains kernels for the GPU and the CPU

SAXPY EXAMPLE

```
// kernel function (CPU)
void saxpy_serial(int n, float alpha, float *x, float *y)
{
  int i;
  for (i=0; i<n; i++) {
    y[i] = alpha*x[i] + y[i];
  }
}
```

CPU version

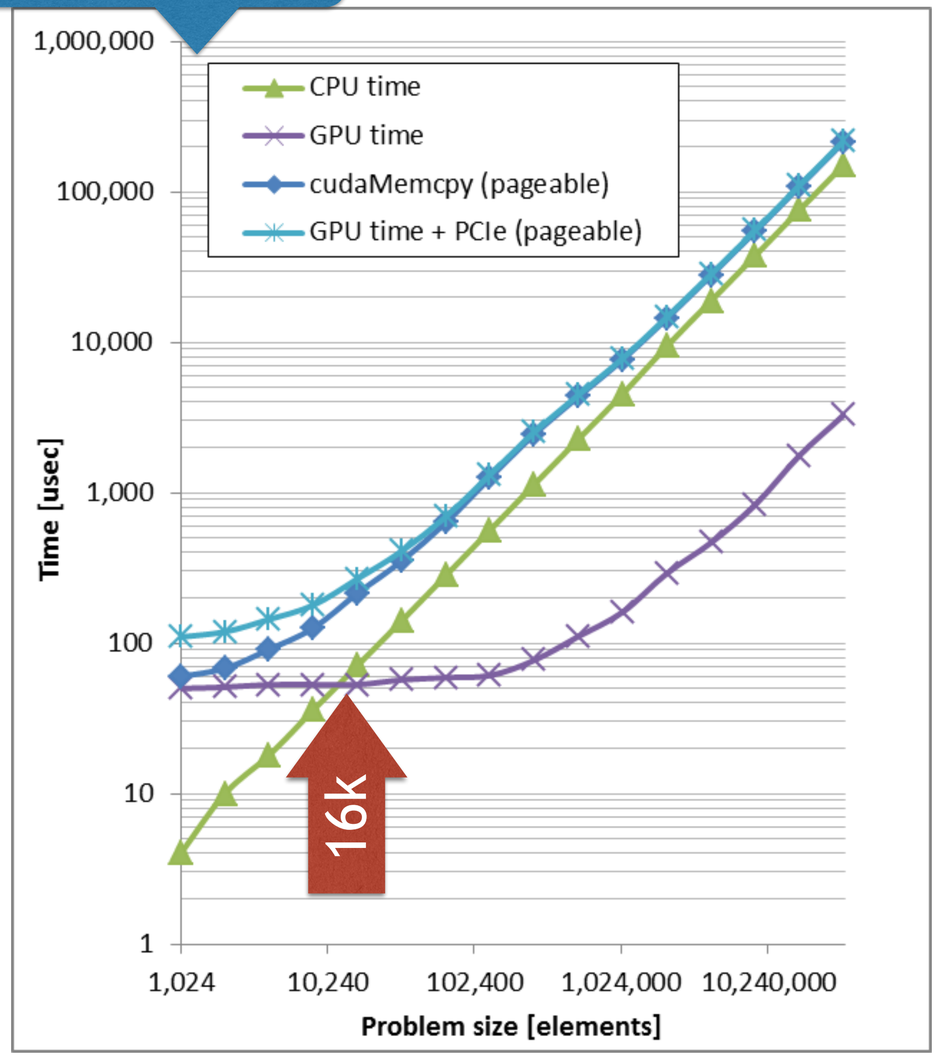
```
// kernel function (CUDA device)
__global__ void saxpy_parallel(int n, float alpha, float *x, float *y)
{
  // compute the global index in vector from
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  // avoid writing past the allocated memory
  if (i<n) {
   y[i] = alpha*x[i] + y[i];
  }
}
```

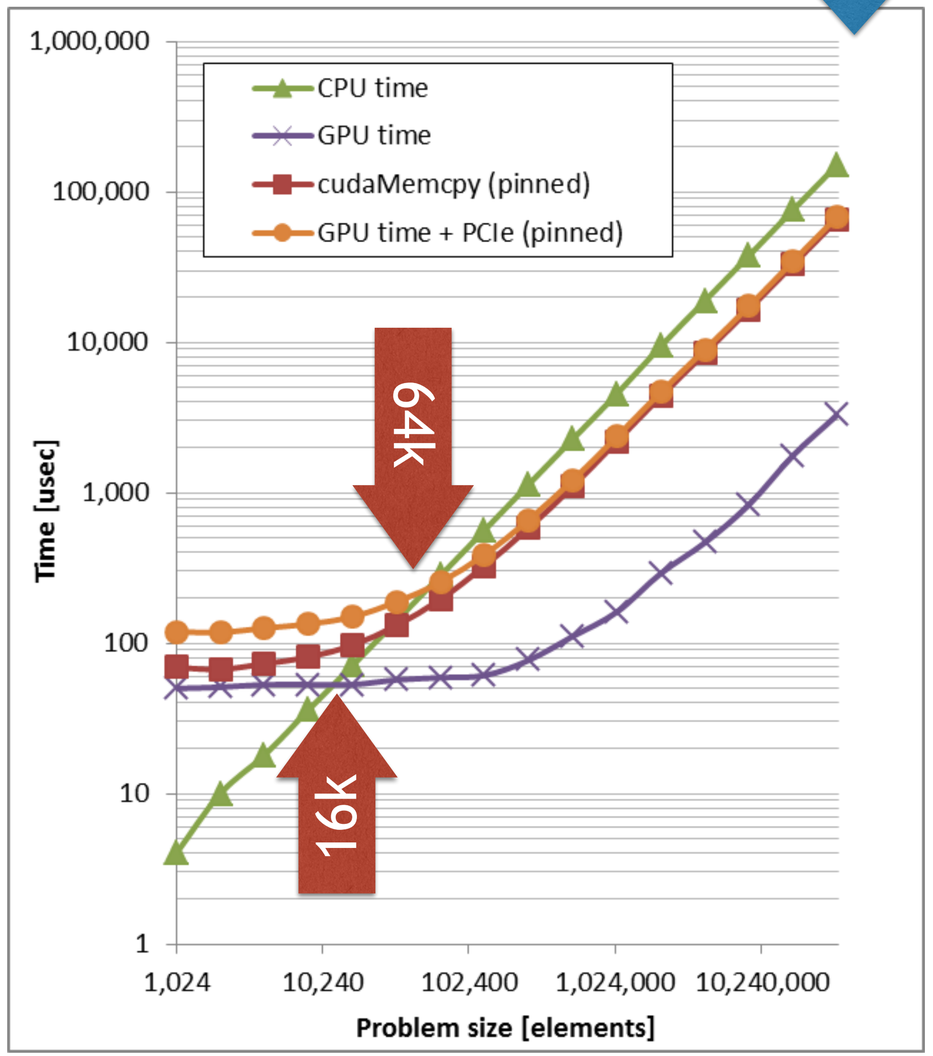
GPU version

INITIAL PERFORMANCE

Huge advantage for GPU when no data movements

Pageable memory vs. pinned memory





Tesla K10, PCIe 2.0 x16, Intel Xeon E5 (single-threaded)

PINNED MEMORY

Replace `malloc` with `cudaMallocHost`

Significant reduction of data movement costs

Pinned memory is a scarce resource!

```
float *h_x;
float *h_y;
float *d_x;
float *d_y;

if (USE_PINNED_MEMORY) {
  cudaMallocHost ( (void**) &h_x, N*sizeof(float) );
  cudaMallocHost ( (void**) &h_y, N*sizeof(float) );
} else {
  h_x = (float*) malloc ( N*sizeof(float) );
  h_y = (float*) malloc ( N*sizeof(float) );
}
cudaMalloc ( (void**)&d_x, N*sizeof(float) );
cudaMalloc ( (void**)&d_y, N*sizeof(float) );
```

COMMON ERRORS

CUDA Error: the launch timed out and was terminated

-> Stop X11

CUDA Error: unspecified launch failure

-> Typically a segmentation fault

CUDA Error: invalid configuration argument

-> Too many threads per block, too many resources per thread (shared memory, register count)

Compile problem:

```
mmult.cu(171): error: identifier "__eh_curr_region" is undefined
```

-> Non-static shared memory, use static allocation of shared memory

SUMMARY

- Introduction to CUDA
  - Pretty unusual concept compared to CPU programming
  - Once understood: pretty easy programming model
  - Did you see any vector instructions today?
- Direct control over hardware
  - Plenty of opportunities for the (experienced) user
  - Increases the burden
- Main differences to CPU programming
  - Sophisticated resource planning
  - Many manual data movements
  - Limited memory capacity