RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to High Performance Computing

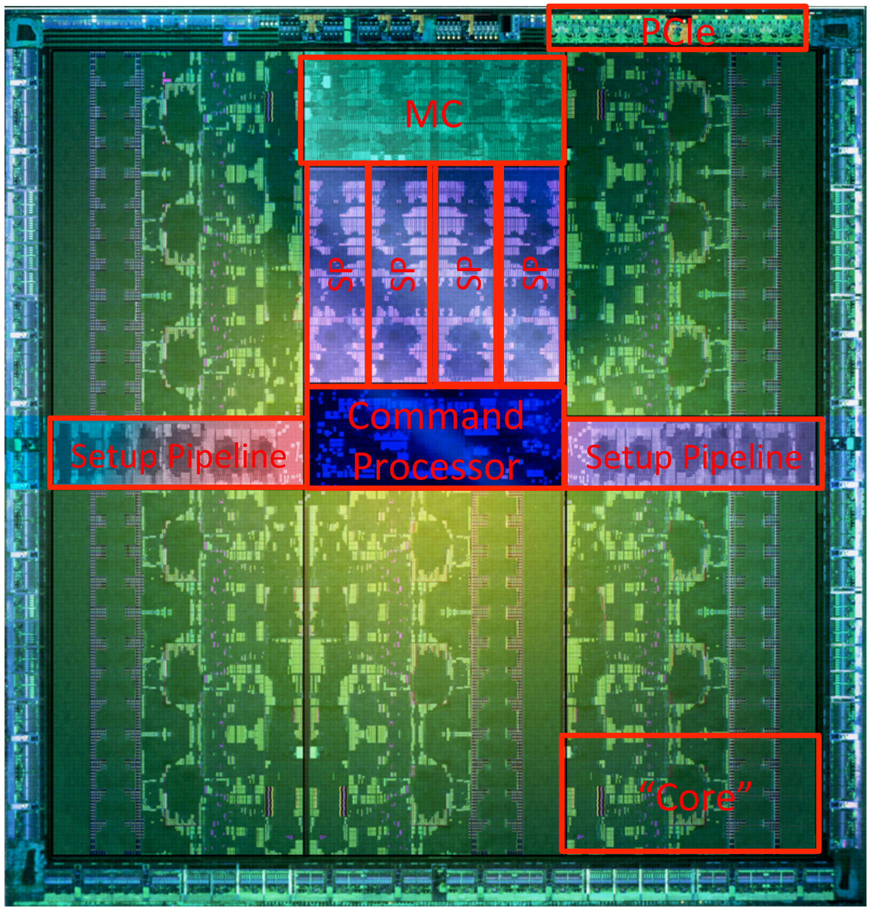
Lecture 02 – CUDA Programming

Holger Fröning
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

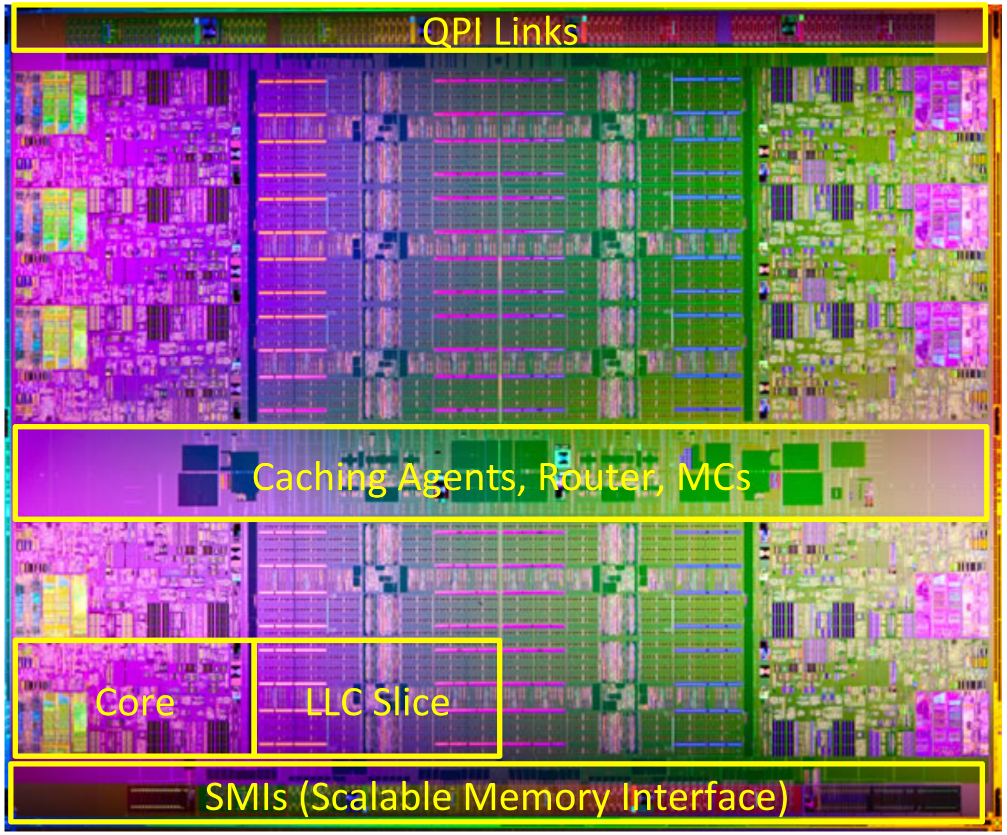
With material from D. Kirk, W. Hwu („Programming Massively Parallel Processors“)

Introduction

GK110



XEON-E7



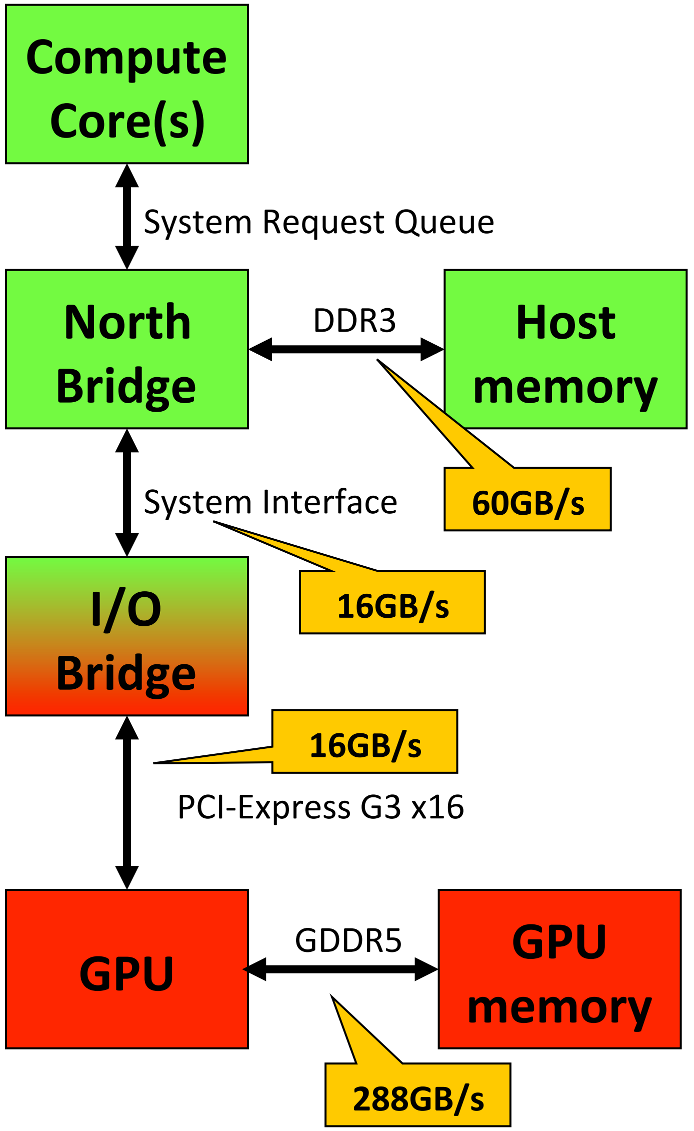
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

GPUs vs. CPUs

	Tesla K20	Xeon E7-4800 4P
Core count	13 SMs 64/832 (DP), 192/2,496 (SP)	10 Cores 2 FP-ALUs/core
Frequency	0.7GHz	2.4GHz
Peak Compute Performance	1,165 GFLOPS (DP) 3,494 GFLOPS (SP)	96 GFLOPS (DP)
Use model	throughput-oriented	latency-oriented
Latency treatment	toleration	minimization
Programming	1000s-10,000s of threads	10s of threads
Memory bandwidth	250 GBytes/sec	34 GByte/s (per P)
Memory capacity	<= 8 GB	up to 2TB
Die size	550mm^2	684 mm^2
Transistor count	7.1 billion	2.3 billion
Technology	28nm	32nm

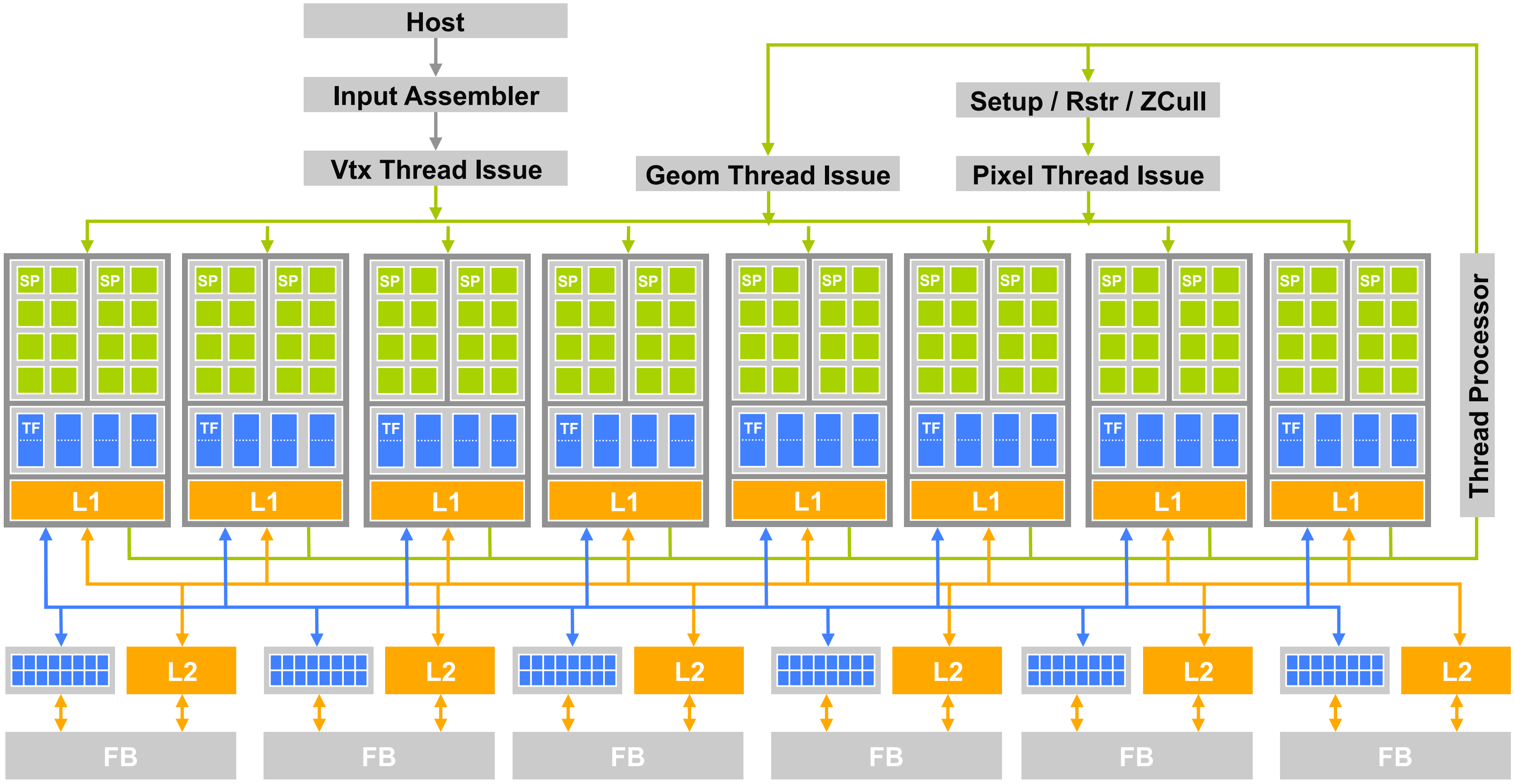
Introduction

- NVidia CUDA
  - **Compute kernel** as C program
  - Explicit data- and thread-level parallelism
  - Computing, not graphics processing
  - Host communication
- Memory hierarchy
  - Registers at thread level
  - Shared memory at thread block level
  - GPU memory
  - Host memory
- More HW details exposed
  - Use of pointers
  - Load/store architecture
  - Barrier synchronization of thread blocks
- Requires more detailed understanding of underlying HW



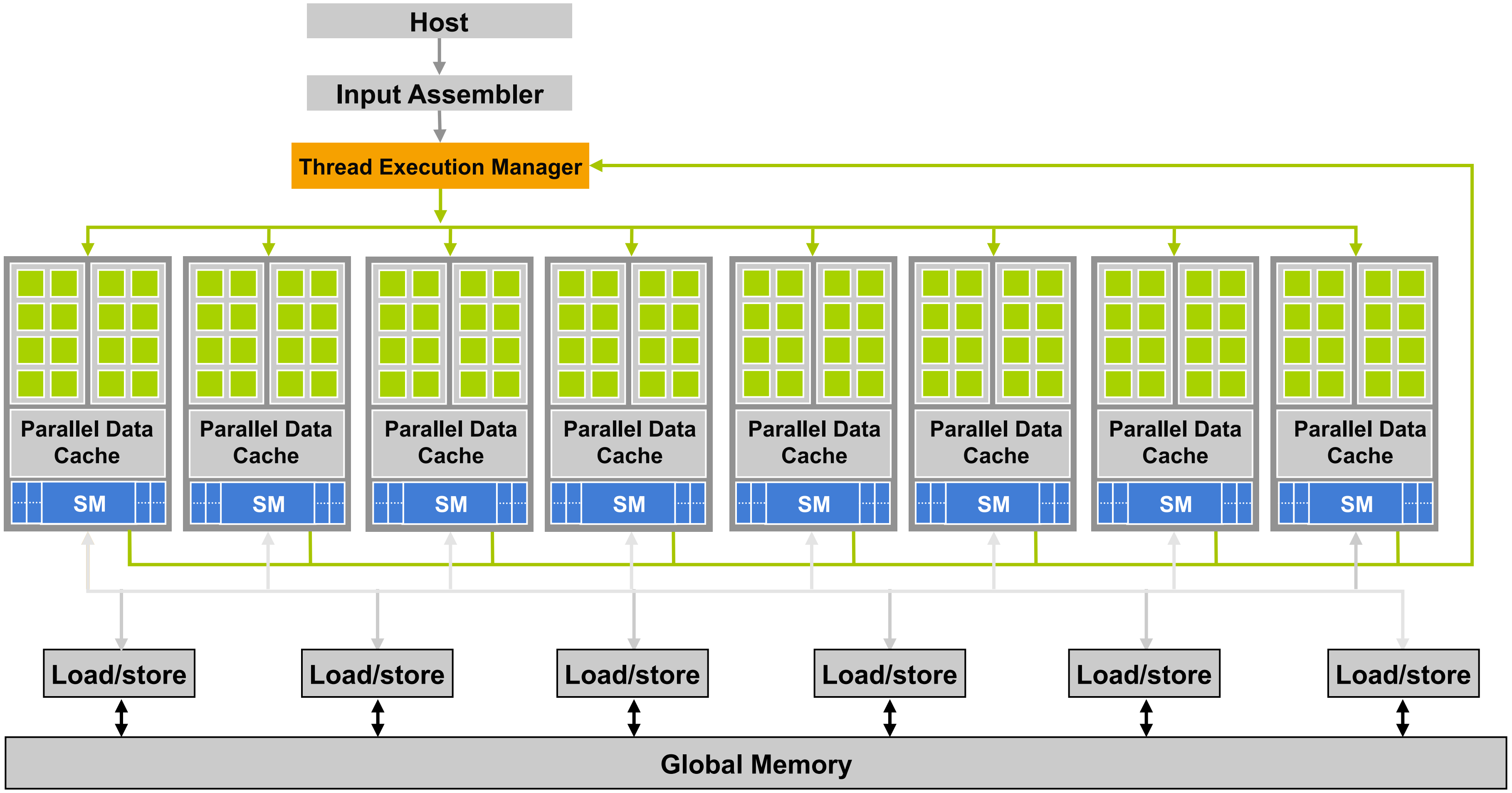
Introduction

- G80 architecture for graphic processing



Introduction

- G80 architecture for general-purpose computing

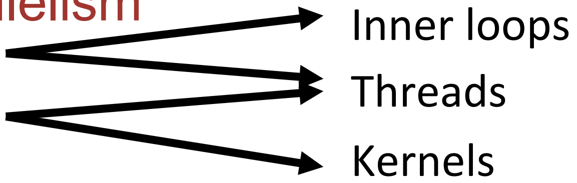


RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

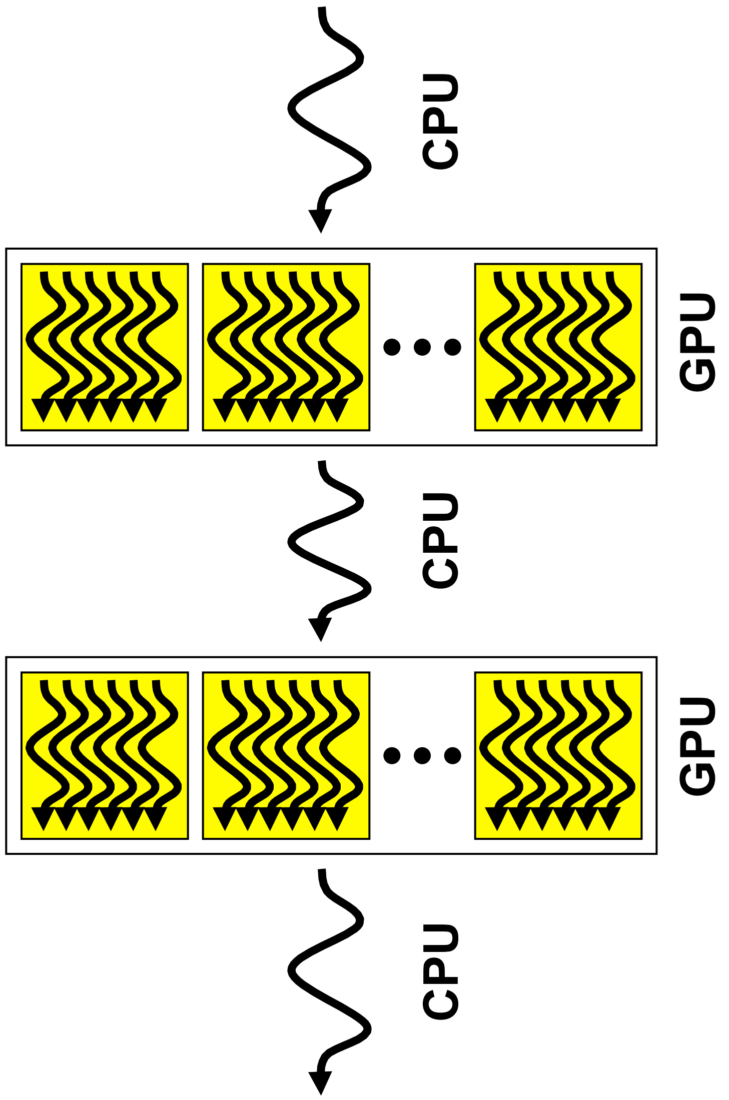
CUDA Programming Model

Programming Model

- C Extension with three main abstractions
  1. Hierarchy of threads
  2. Shared memory
  3. Barrier synchronization
- Exploiting parallelism
  - Fine-grain DLP
  - TLP



- CUDA program consists of CPU & GPU part
  - CPU part: part of the programm with no or little parallelism
  - GPU part: high parallel part, SPMD-style
- Concurrent execution
  - Non-blocking thread execution
  - Explicit synchronization



Programming Model

```
__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 );
}
```

- **Kernels: n**-fold execution by N threads
  - Definition: keyword `__global__`
- Execution: `kernel <<<NumBlocks, threadsPerBlock>>> (args)`
- (Unique) ID: `threadIdx`
  - Control flow for SPMD programs
  - Memory access orchestration

Programming Model

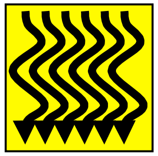
```
__global__ void matAdd (float A[N][N], float B[N][N], float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

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

- `threadIdx` has up to 3 dimensions: `threadIdx.{x,y,z}`

➔ Each thread block has up to 3 dimensions

- Number of threads per block is limited
  - 512 x 512 x 64 ➔ 1024 x 1024 x 64 (implementation dep.)

➔ Additional hierarchy level: grid = blocks of threads
  - Unique ID `blockIdx`, up to 3 dimensions
  - Blocks are executed independently and implementation-dependent
  - Number of blocks limited (typ. 64k-1 per dimension)


up to 3D

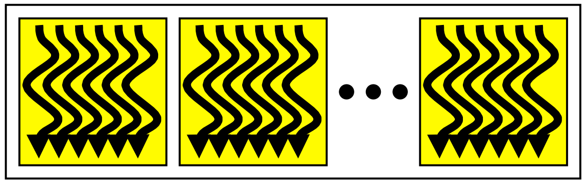
Programming Model

```
__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 );
}
```

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

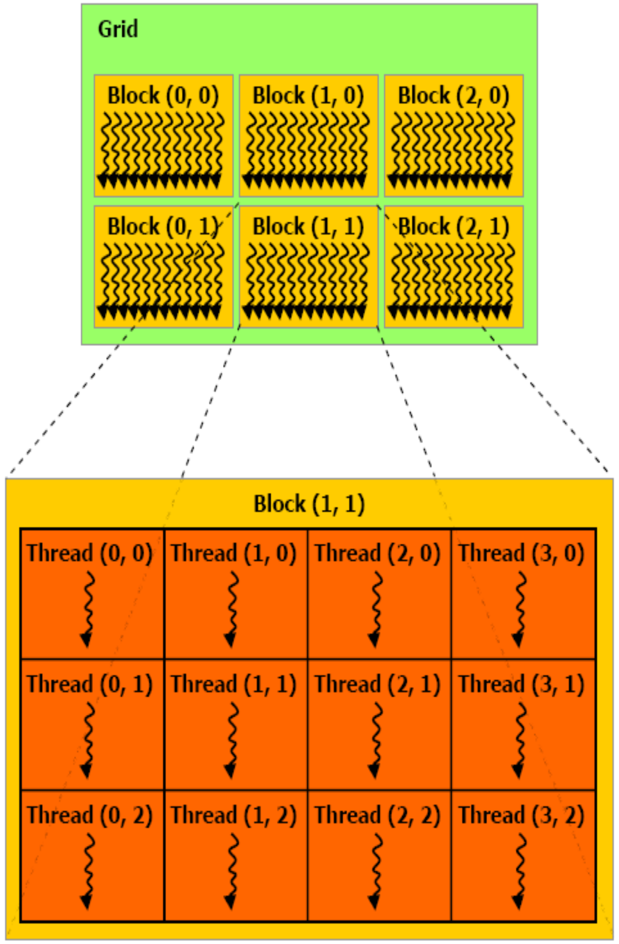
Block size

Fine grain PCAM
One thread computes one element



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Programming Model

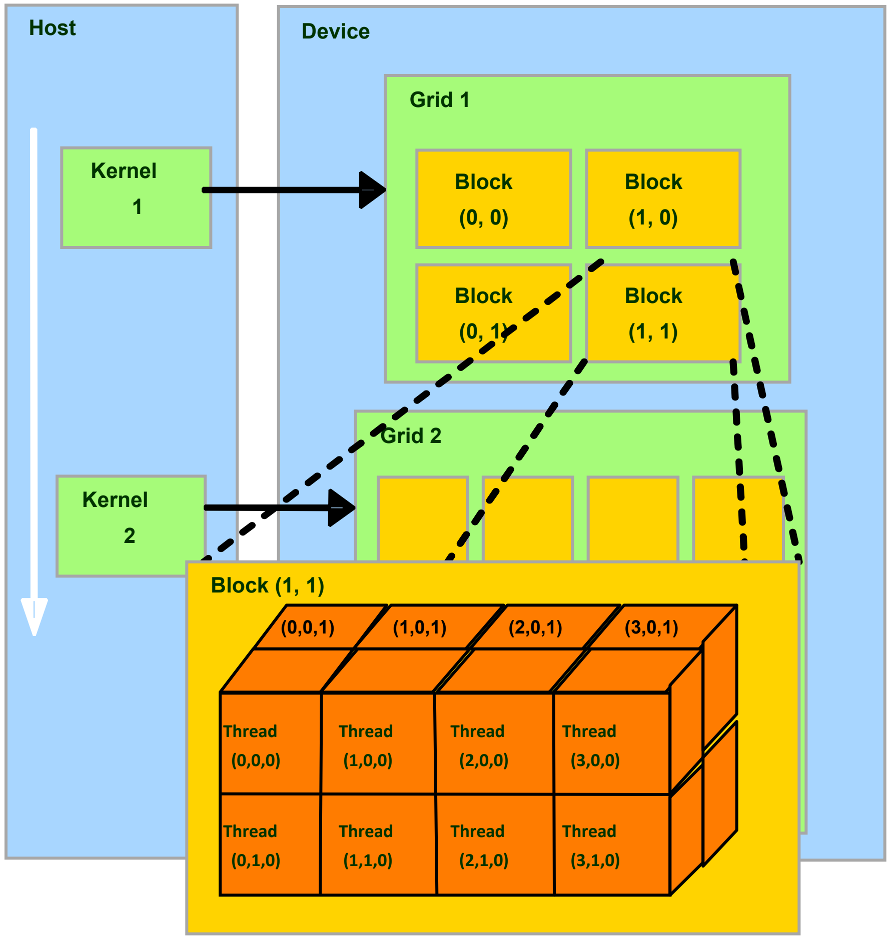


- Thread hierarchy
  - Grid of thread blocks
  - Blocks of equal size
- Given problem size N, how to choose parameters threads per block respectively blocks per grid?
- Recommendations wrt block count
  - >2x number of SMs
  - Optimal: 100 – 1000 (max. 64k-1)
- Recommendations wrt threads/block
  - Parallel slackness vs. number of registers per thread
    - R / (B * ceil (T,32))

R = total registers, B = active blocks / SM, T = threads per block, ceil = round up to next multiple of 32
R = 8k/SM .. 32k/SM (implementation dependent)

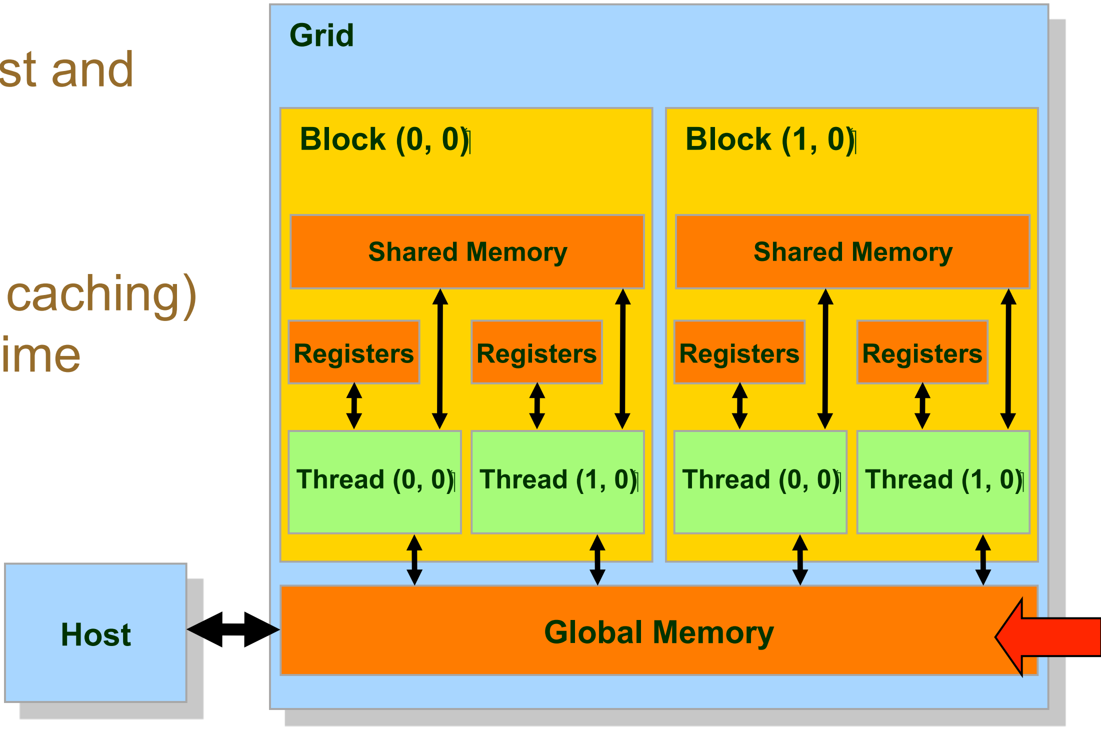
Programming Model

- 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
- Iterative kernel invokations



Memory Hierarchy – Global Memory

- Global memory
  - Communication between host and device
  - Accessible from all threads
    - Read/write
  - High associated latency (no caching)
  - Lifetime exceeds thread lifetime
- Sensitive to coalescing
- Allocation
  - cudaMalloc (&dmem, size);
- De-allocation
  - cudaFree (dmem);
- Data transfer (blocking)
  - cudaMemcpy (*dst, *src, size, transfer_type);
  - cudaMemcpyAsync (…)



Memory Hierarchy – Global Memory

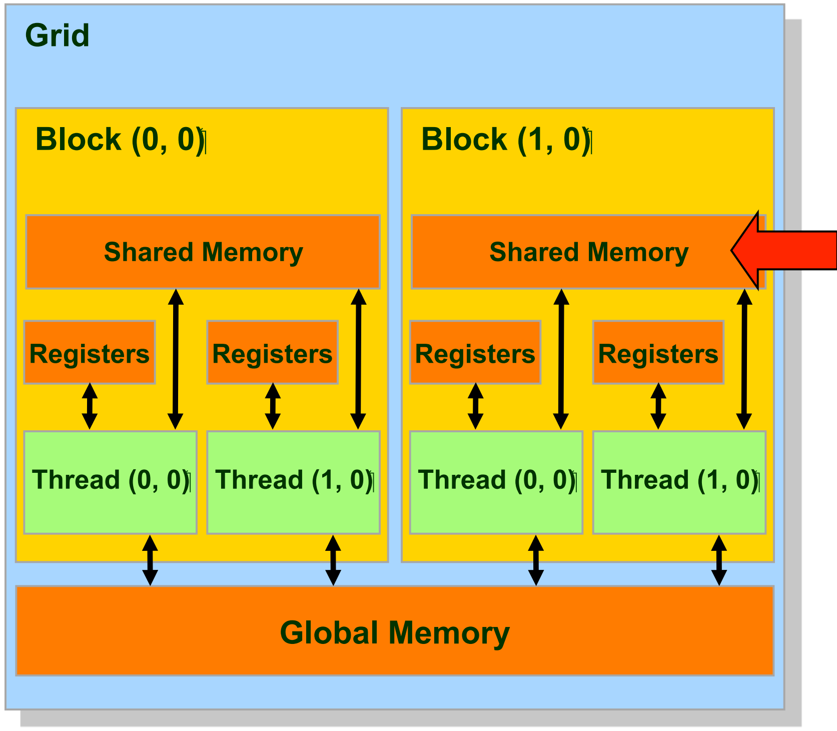
Variable scope annotation

```
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
```

Only references to device memory

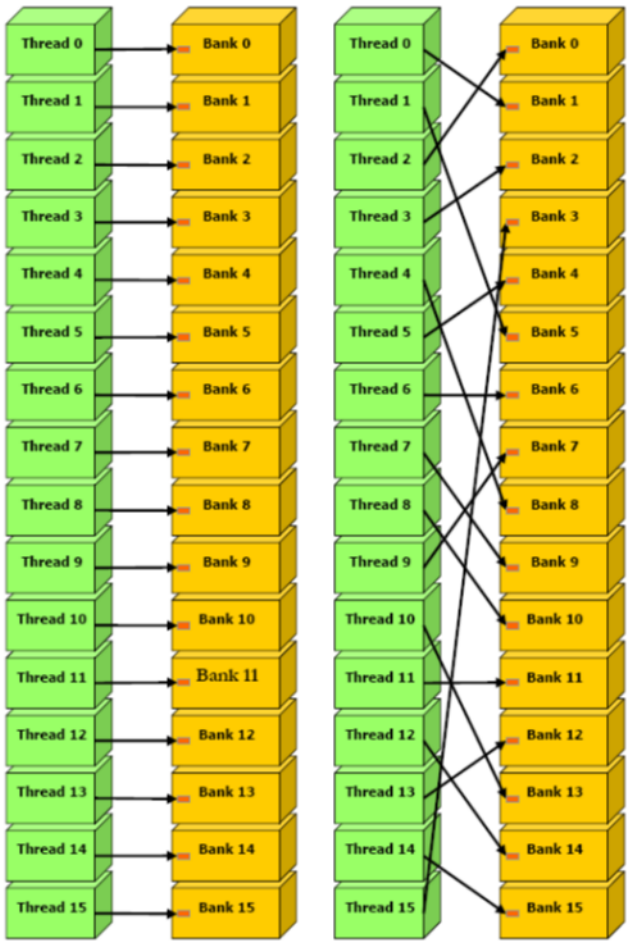
Memory Hierarchy – Shared Memory

- Shared Memory
  - On-chip memory
  - Lifetime: thread 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

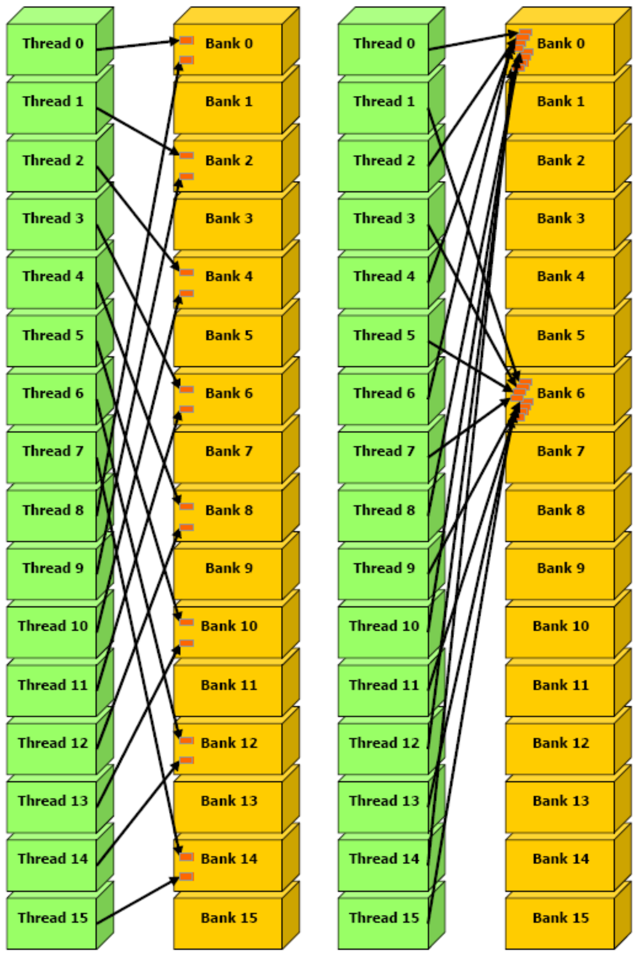


Memory Hierarchy – Shared Memory

Shared Memory
Bank Access without Blocking



Shared Memory
Bank Access with Blocking



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Memory Hierarchy - Declaration

	Location	Access from
`__device__ float var;`	global mem	device/host
`__constant__ float var⏎;`	constant mem	device/host
`__shared__ float var;⏎`	shared mem	block

- `__device__` can be combined with one additional type
  - Otherwise placed in global memory, application lifetime
- `__constant__` implies `__device__`, see above
- `__shared__` implies `__device__`
  - Lifetime of a block, only available within this block
  - `__syncthreads` to wait for outstanding write operations
  - Unconstrained completion of read/write operations (exception: `volatile`)

Variable Types

- 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 Declarations

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

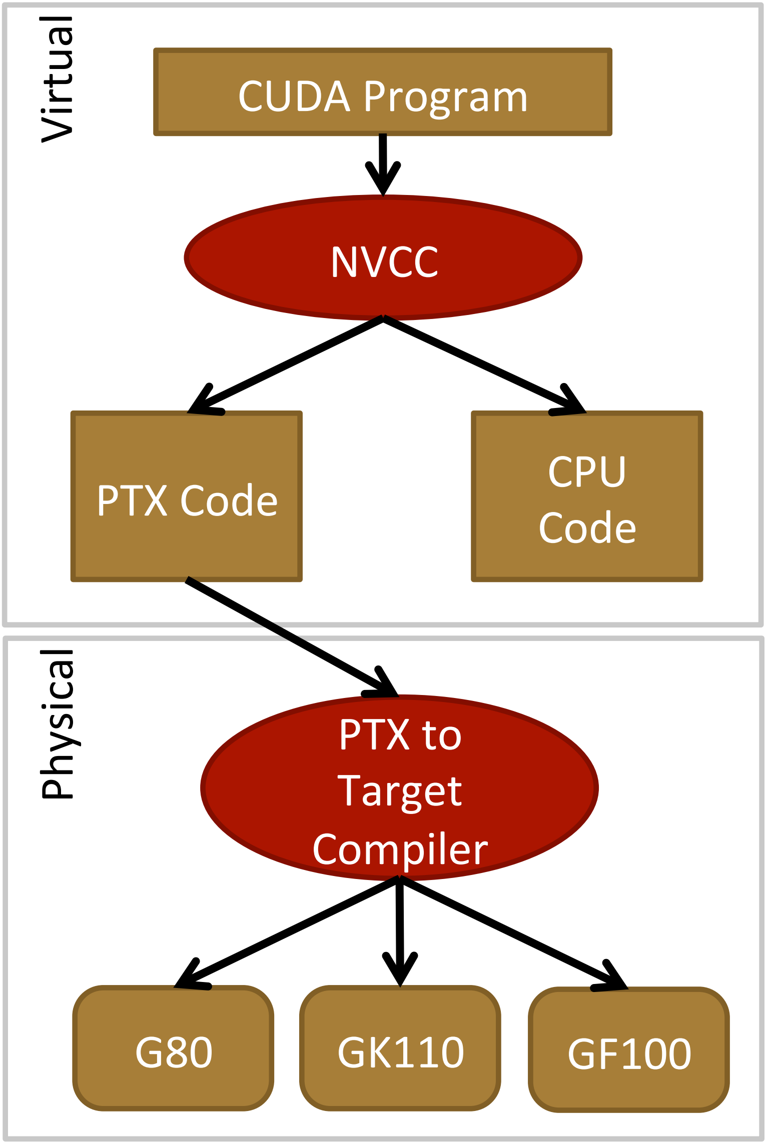
- **`__global__`** defines a kernel (return type: void)
  - Rückgabewert vom Typ void
- **`__host__`** is optional
- **`__host__`** and **`__device__`** can be combined
- No pointers to **`__device__`** functions
  - Exception: **`__global__`** functions
- For functions that are executed on the GPU:
  - No recursions, only static variable declarations, no variable parameter count

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Just-in-time Compilation

- Device code only supports C-subset of C++
- Compile with nvcc
  - Compiler Driver
  - Calls other required tools
    - cudacc, g++, cl, …
  - Debugging:
    command line parameter -deviceemu
- 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

```
export LD_LIBRARY_PATH=/usr/local/cuda/lib
export PATH=$PATH:/usr/local/cuda/bin
nvcc –help
nvcc foo.cu -o foo
```



Brief Property Survey (deviceQuery)

Model	Revision number	Total global memory [bytes]	Multiprocessors	Cores	Total constant memory [bytes]	Shared memory per block [bytes]	Registers per block	Warp size	Threads per block	Max dimension of a block	Max. dimension of a grid	Max. memory pitch [bytes]	Clock rate [GHz]	Concurrent copy and execution
GeForce GTX 280	1.3	1G	30	240	64k	16k	16k	32	512	512 x 512 x 64	65535 x 65535 x 1	256k	1.3	Y 1
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

CUDA Example: saxpy

SAXPY Example

Scalar Alpha X Plus Y
y[i] = alpha*x[i] + y[i]

- SAXPY: Scalar Alpha X Plus Y
- Simple test to compare the GPU and the CPU
  - 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

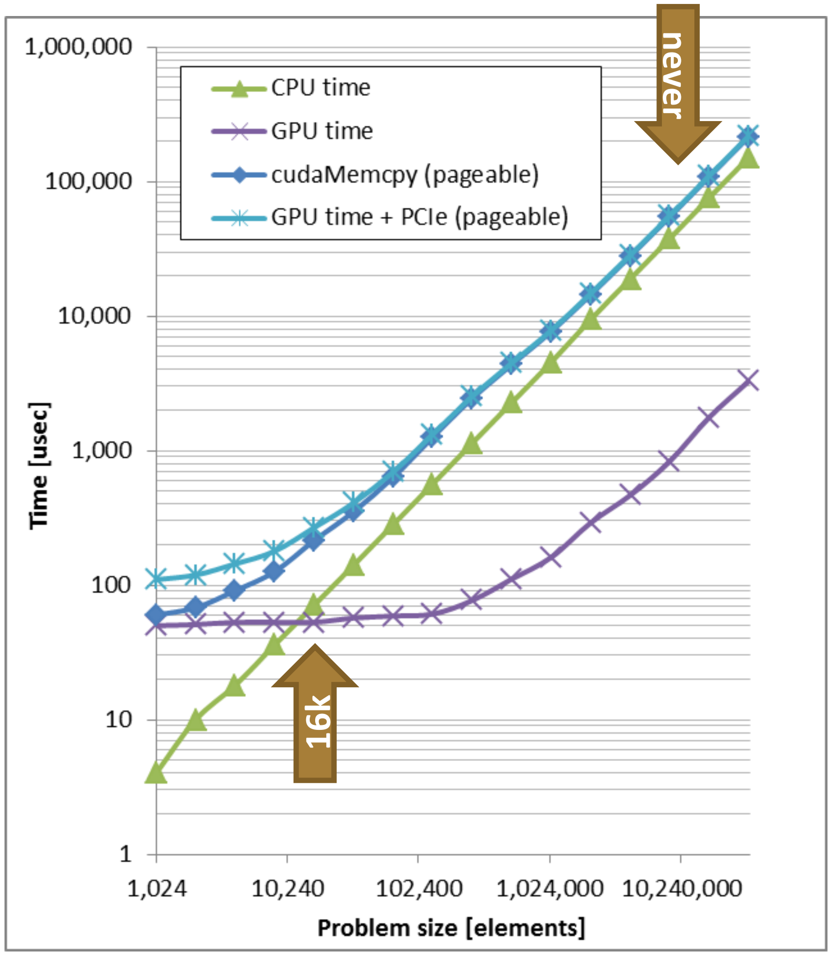
SAXPY Example

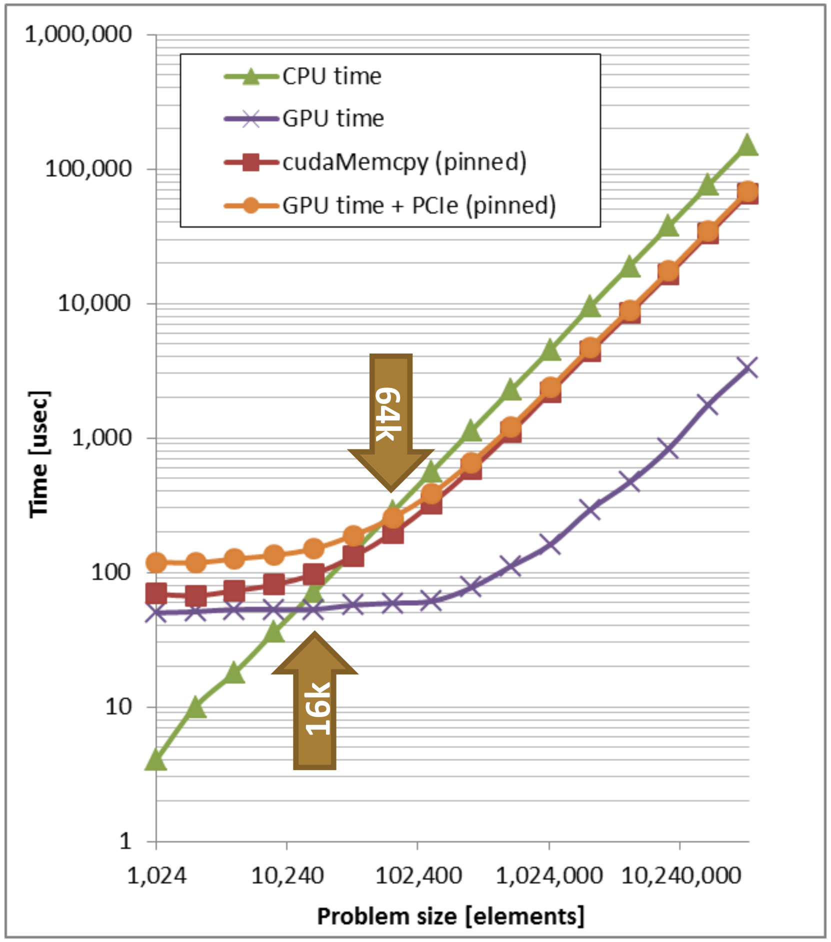
```
/////////////////////////////////////
// kernel function (CPU)
/////////////////////////////////////
void saxpy_serial(int n, float alpha, float *x, float *y)
{
  int i;
  for (i=0; i<n; i++) {
    y[i] = alpha*x[i] + y[i];
  }
}
```

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

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

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

SAXPY Example (2)





- Tesla K10, PCIe 2.0 x16 connection, Intel Xeon E5, single-threaded
- Break-even at 16k
- Huge advantage for GPU when no data movements
- Pageable memory vs. pinned memory

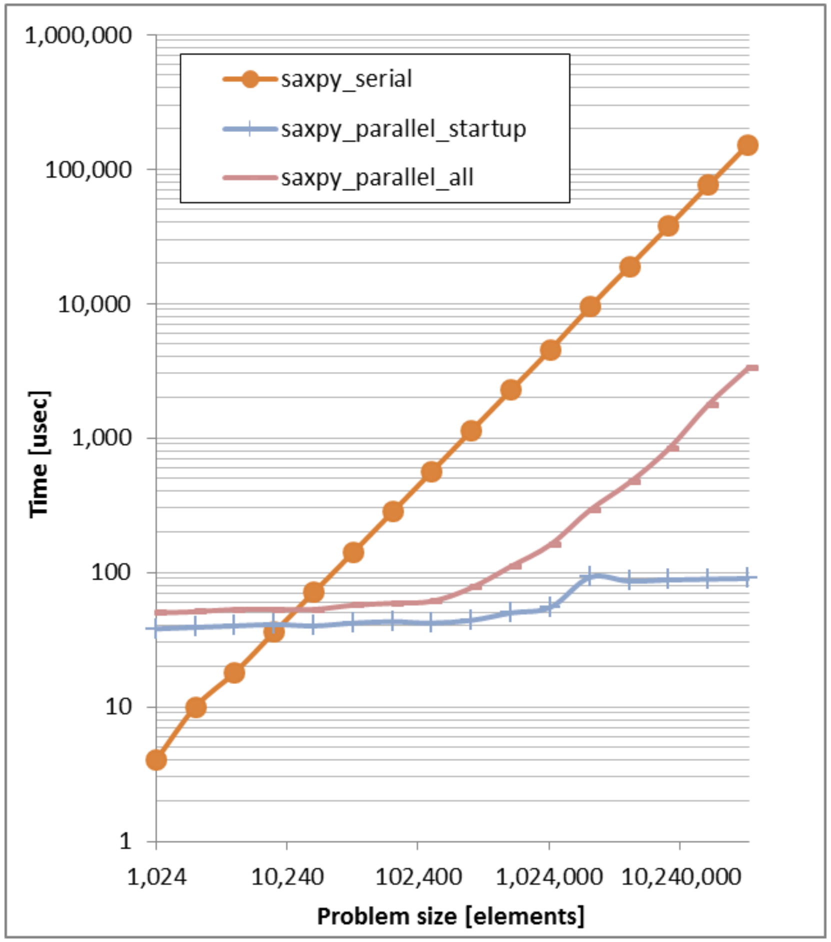
SAXPY Example (3)

- Replace malloc with cudaMallocHost
  - Significant reduce data movement costs
- Pinned memory is a scarce resource!

```
//////////////////////////////////////
// (2) allocate memory on host (main CPU memory) and device,
//     h_ denotes data residing on the host, d_ on device
//////////////////////////////////////
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));
checkErrors("memory allocation");
```

SAXPY Example

- Successful measures ☺
  - Use of pinned memory
  - (Increase computational intensity)
- Unsuccessful ☹
  - Precompilation: 40usec kernel startup time is maintained, independent of pre-compiled code or not
- SAXPY is a trivial example
  - See dependency analysis



Summary – CUDA & GPU Computing

- Introduction to CUDA
  - Pretty unusual concept compared to CPU programming
  - Pretty easy programming model
  - Pretty challenging for upmost speed-ups
- Direct control over hardware
  - Plenty of opportunities for the (experienced) user
  - Increases the burden

- Main differences to CPU programming
  - Sophisticated resource planning and usage
  - Scratch pad: use shared memory as explicitly controlled cache
  - Data movement over PCIe
  - Limited memory

- Key characteristics for good GPU applications
  - High degree of DLP
  - Low data reuse
  - High computational intensity
  - High bandwidth

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Common Errors

- Runtime problems:

```
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