MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

The Basics

Klaus Mueller and Sungsoo Ha

Stony Brook University

Computer Science

Stony Brook, NY

Parallel Computing Explained

CPU vs. GPU

Parallel Computing Explained

Any questions?

Parallelism

What you just saw was an embarrassingly parallel task

- no or very little communication among parallel tasks

Most computational problems are not like that

Parallelism

On the other hand, some processes are not parallel at all

- are they embarrassingly sequential?

Need to find and gauge where the parallelism is

Types of Parallelism

Task based parallelism

- unrelated processes are executed in parallel
- slowest process determines the speed
- also known as *coarse grained parallelism*
- MIMD model = Multiple Instructions Multiple Data

Data based parallelism

- decompose a specific task into *threads*
- each thread executes the same statement at the same time
- also known as *fine grained parallelism*
- SIMD model = Single Instructions Multiple Data

Patterns of Parallelism

Loops

- *for* and *while* statements
- Fork and Join

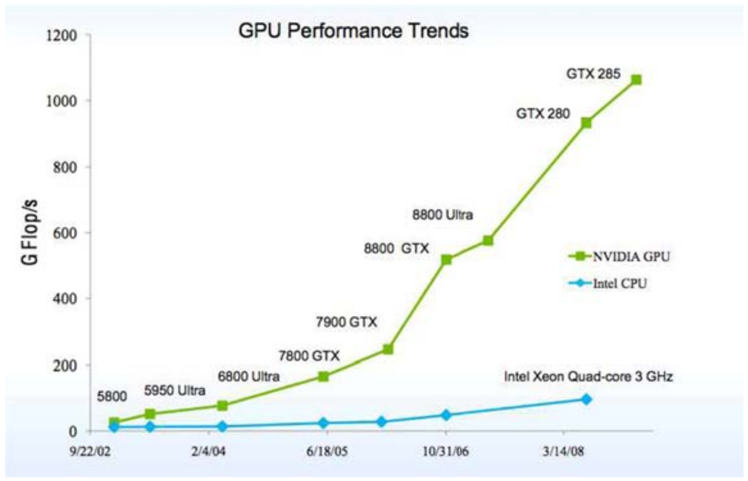
Tiling and grids

- break the domain into sub-problems that map well to the hardware
- 2D tiles/grid for images, 3D tiles/grid for volumes

Divide and Conquer

- recursion: can present problems for parallelism when too deep
- better use an iterative approach that solves a level in parallel

Speedup Curves



Amdahl's Law

Governs theoretical speedup

$$S = \frac{1}{(1-P) + \frac{P}{S_{parallel}}} = \frac{1}{(1-P) + \frac{P}{N}}$$

P: parallelizable portion of the program
S: speedup
N: number of parallel processors

Amdahl's Law

Governs theoretical speedup

$$S = \frac{1}{(1-P) + \frac{P}{S_{parallel}}} = \frac{1}{(1-P) + \frac{P}{N}}$$

P: parallelizable portion of the program
S: speedup
N: number of parallel processors

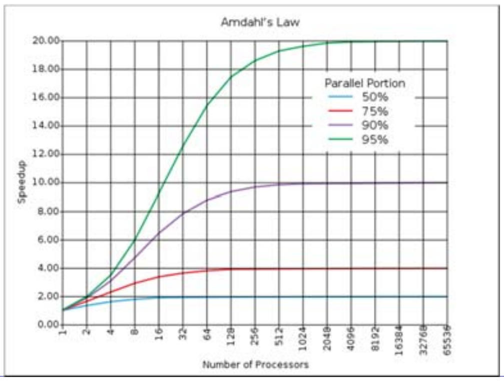
P determines theoretically achievable speedup

- example (assuming infinite N): P=90% → S=10
  P=99% → S=100

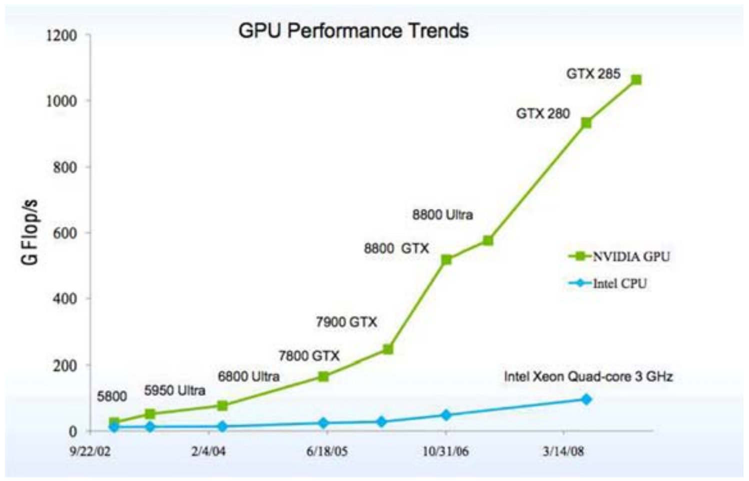
Amdahl's Law

How many processors to use

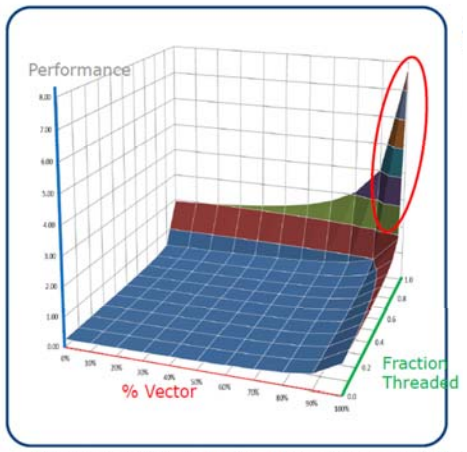
- when P is small → a small number of processors will do
- when P is large (embarrassingly parallel) → high N is useful



Speedup Curves



Decision Support

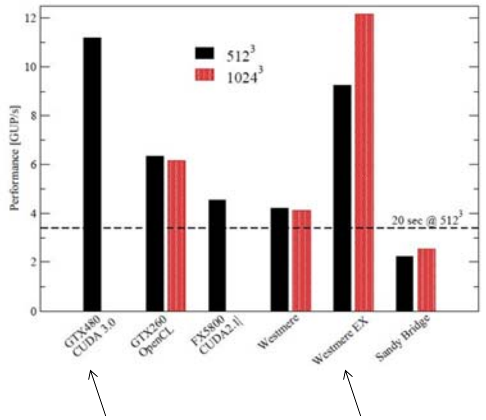


* Theoretical acceleration of a highly parallel processor over a Intel® Xeon®

Comparison with CPUs

Backprojection task

- 496 projections
- size 1,248×960 each



$500

$4,500

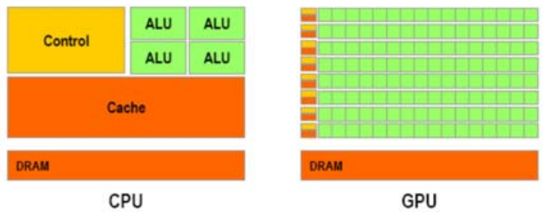
from Treibig et al. “Pushing the limits for medical image reconstruction on recent standard multicore processors,” *International Journal of High Performance Computing Applications*, 2012

Beyond Theory….

GPUs are more than parallel computing

There are certain features that provide a turbo boost

- special ASIC circuits for frequent operations
- latency hiding by rapid thread switching
- special memory organization for 2D data
- schedulers
- managers
- APIs, drivers
- caches
- dedication to computing



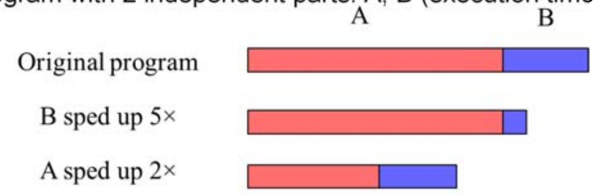
Focus Efforts on Most Beneficial

Optimize program portion with most ‘bang for the buck’

- look at each program component
- don’t be ambitious in the wrong place

Example:

- program with 2 independent parts: A, B (execution time shown)



- sometimes one gains more with less

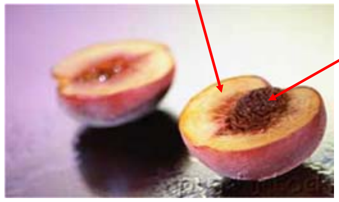
Programming Strategy

Use GPU to complement CPU execution

- recognize parallel program segments and only parallelize these
- leave the sequential (serial) portions on the CPU

parallel portions (enjoy)

sequential portions (do not bite)

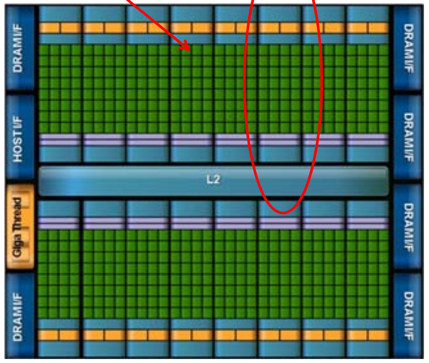


PPP (Peach of Parallel Programming – Kirk/Hwu)

The Hardware …. NVIDIA Fermi

CUDA Core

SM (Streaming Multiprocessor) → has 32 Streaming Processors (SP) = CUDA core

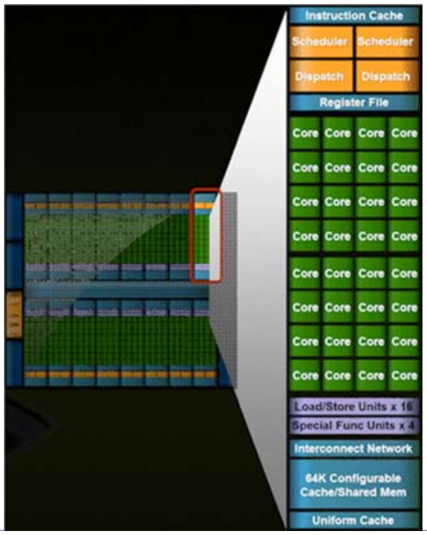


On chip:

SMs: up 16

CUDA cores: 32/SM → up to 512/chip

The Hardware …. NVIDIA Fermi



full cross-bar interface

32 CUDA Cores

4 special function units (sin, cosine, reciprocal, and square root)

Host and Device

Host → CPU

- controls program flow
- manages threads
- loads GPU programs (kernels)
- has host memory

Device → GPU

- loads data
- performs computations
- has device memory

Heterogeneous programming model

Cost of Data Transfer

Amortizing the cost for data transfer is important

- computational benefit of a transfer plays a large role
- transfer costs are (or can be) significant

Adding two ($N \times N$) matrices:

- transfer back and from device: 3 N^2 elements
- number of additions: N^2

→ operations-transfer ratio = 1/3 or O(1)

Multiplying two ($N \times N$) matrices:

- transfer back and from device: 3 N^2 elements
- number of multiplications and additions: N^3

→ operations-transfer ratio = O(N) grows with N

Parallelism Exposed as Threads

Thread management:

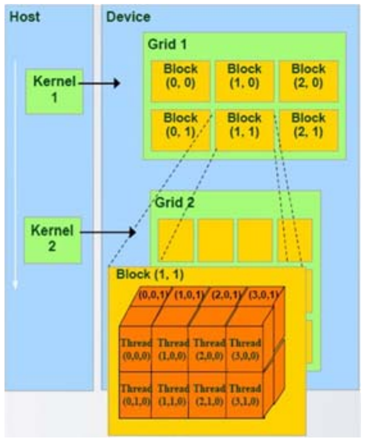
- all threads run the same code
- a thread runs on one core

The threads divide into *blocks*

- each block has a unique ID → *block ID*
- each thread has a unique ID within a block → *thread ID*
- block ID and thread ID can be used to compute a *global ID*

The blocks form a *grid*

Block/grid size can be set in program



Threads Organization: Fine Grain

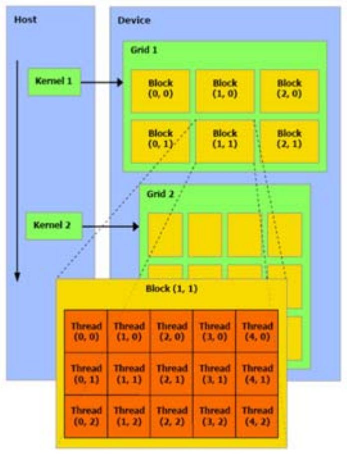
Threads within a block are organized into *warps*

- execute the same instruction simultaneously with different data

A warp is 32 threads (fixed)

One SM can maintain 48 warps simultaneously

- keep one warp active while 47 wait for memory → latency hiding
- 32 threads × 48 warps ×16 SMs → 24,576 threads !



Block and Thread Management

Upon invoking a CUDA program from the host:

Block-level

- blocks are serially distributed to SMs
- threads of a block execute on one SM
- as thread blocks terminate, new blocks are launched on vacated SMs

Thread-level

- each SM launches warps of threads
- SM schedules and executes warps that are ready to run
- as warps and thread blocks complete, resources are freed

Choose grid dimensions according to task dimensions (1D, 2D, now 3D)

Block Scheduling: Example

Threads are assigned to SMs in block granularity

- up to 8 blocks to each SM as resource allows
- choose number of blocks per SM based on overall task size
- big blocks and small task will leave many SMs idle

An SM can take up to 1,536 threads

- could be 512 (threads/block) * 3 blocks
- or 256 (threads/block) * 6 blocks, etc.

The optimal block size depends on:

- how much latency needs to be hidden (larger blocks)
- how much memory is needed per thread (smaller blocks)
- task size (see above)

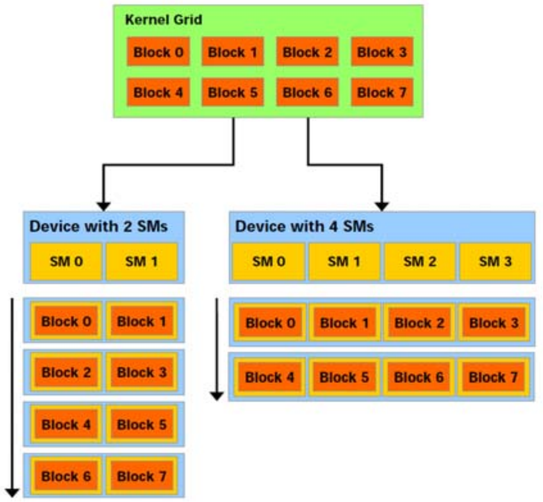
Mapping the Architecture to Parallel Programs

Mapping of blocks to SMs

- depends on device hardware
- *transparent scalability*

Thread management

- very lightweight thread creation, scheduling
- in contrast, on the CPU thread management is very heavy



An Important Player: Memory

CUDA threads may access data from multiple memory spaces:

Thread-level

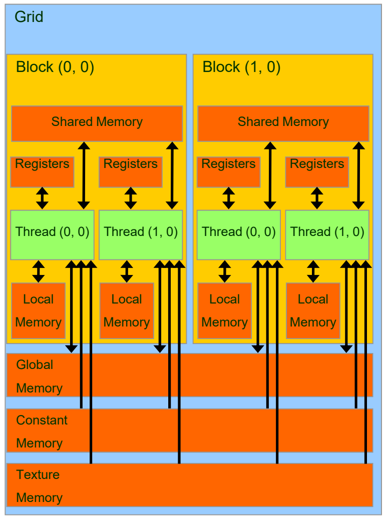
- registers (fast)
- local memory to handle register spills (slow)

Block-level

- shared memory

Grid-level

- global memory (slowest)
- constant memory (read-only)
- texture memory (cached, read-only)
- surface memory (writable texture)



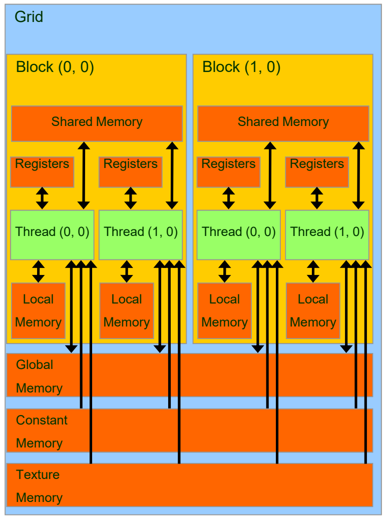
An Important Player: Memory

Memory	On-chip	Cached	Access
Local	N	Y	RW
Shared	Y	Y	RW
Global	N	1D	RW
Constant	N	Y	R
Texture	N	1-3D	R

Caches are on-chip

Code development strategy

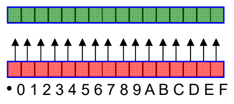
- start by using just global memory
- then optimize
- more about this later



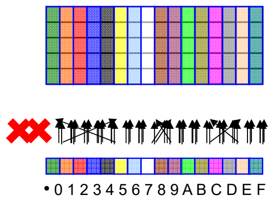
Global vs. Shared Memory

SPIE Medical Imaging

Global memory
- partitioned into segments divisible by 32
- ensure coalesced access
- most efficient when data access is contiguous



Shared memory
- organized into 32 banks
- ensure conflict-free access



If data is not aligned well in global memory
- align it in shared memory
- use collaborative load operation

Latency Hiding -- Revisited

SPIE Medical Imaging

Latency hiding is a form of hardware multi-threading

Major source of the speedup of GPUs
- a new warp is switched to within one clock cycle

But….hardware multi-threading requires memory
- contexts of all these threads must be maintained in memory
- this typically limits the amount of threads that can be simultaneously maintained for latency hiding
- so there is a tradeoff

Avoid Latency – Exploit Locality

SPIE Medical Imaging

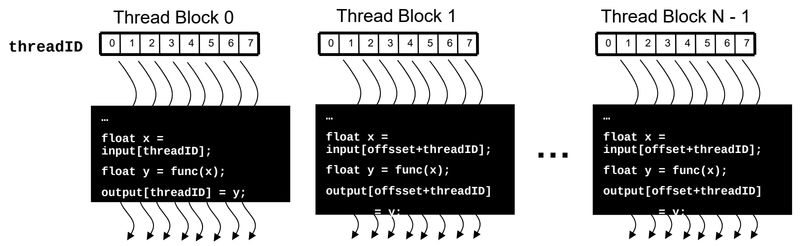
Temporal locality
- data that was accessed before will be likely accessed again
- use cache to reduce access latencies

Spatial locality
- data close to the data accessed last will likely be accessed soon
- fetch entire cache lines when accessing one element

Exploit locality by
- storing data in shared memory
- configure hardware caches (L2, CUDA vs. self-managed shared memory)
- e.g., split 64 KB/block into 48 KB CUDA cache and 16 KB self-managed (Fermi and higher)

Thread Communication Across Blocks

SPIE Medical Imaging

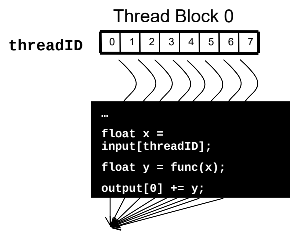


```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
```

```
…
float x =
input[offsset+threadID];
float y = func(x);
output[offsset+threadID]
        = y;
```

```
…
float x =
input[offset+threadID];
float y = func(x);
output[offset+threadID]
        = y;
```

Thread Communication



```
…
float x =
input[threadID];
float y = func(x);
output[0] += y;
```

Thread communication

- threads within a block cooperate via
  - atomic operations on global memory or shared memory,
  - shared memory + barrier synchronization

Recent Architectures

Kepler

- emphasis on better programmability
- Hyper-Q (block scheduling occurs in parallel – not in a queue)
- Dynamic Parallelism (threads can launch other threads)

Maxwell

- emphasis on energy efficiency
- independent warp schedulers eliminating crossbar
- increased L2 cache allowing reduction in memory bus
- reduction in number of cores

Pascal (announced)

- 3D memory
- unified memory (CPU and GPU) with NVLink fast bus

Next – Small Example

Programmed in CUDA

CUDA = Compute Unified Device Architecture

- C-like language
- language and API created by NVIDIA
- libraries available (cuBLAS, cuFFT, Thrust, …)

Vector Add – CPU

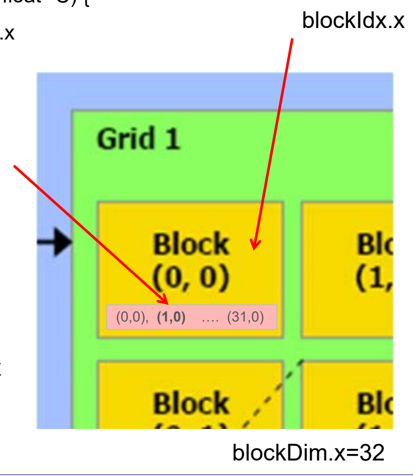
```
void vectorAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i]; }


int main() {
    int N = 4096;
            // allocate and initialize memory
    float *A = (float *) malloc(sizeof(float)*N);
    float *B = (float *) malloc(sizeof(float)*N);
    float *C = (float *) malloc(sizeof(float)*N);
    init(A); init(B);

    vectorAdd(A, B, C, N);          // run kernel
    free(A); free(B); free(C);}     // free memory
```

Vector Add – GPU

```
__global__ void gpuVecAdd(float *A, float *B, float *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x
    C[tid] = A[tid] + B[tid]; }
```



tid = blockId.x * blockDim.x + threadIdx.x

Vector Add – GPU

```
int main() {
    int N = 4096;          // allocate and initialize memory on the CPU
    float *A = (float *) malloc(sizeof(float)*N);
          float *B = (float *) malloc(sizeof(float)*N); *C = (float*)malloc(sizeof(float)*N)
    init(A); init(B);
          // allocate and initialize memory on the GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N);
    cudaMalloc(&d_B, sizeof(float)*N);    cudaMalloc(&d_C, sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
          // configure threads
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);
          // run kernel on GPU
    gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
          // copy result back to CPU
    cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);
          // free memory on CPU and GPU
    cudaFree(d_A);  cudaFree(d_B);  cudaFree(d_C);  free(A);  free(B);  free(C); }
```

Subdivide Problem into 3 Stages

Handle each data subset with one thread block by:

- **load** the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
- **compute** on the subset in shared memory; each thread can efficiently multi-pass over any data element
- **copy** results from shared memory to global memory

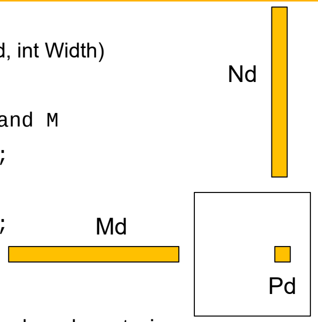
Let's see how this works using a matrix multiplication example

Example: Matrix Multiplication

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```



Using Multiple Blocks

Break-up Pd into tiles

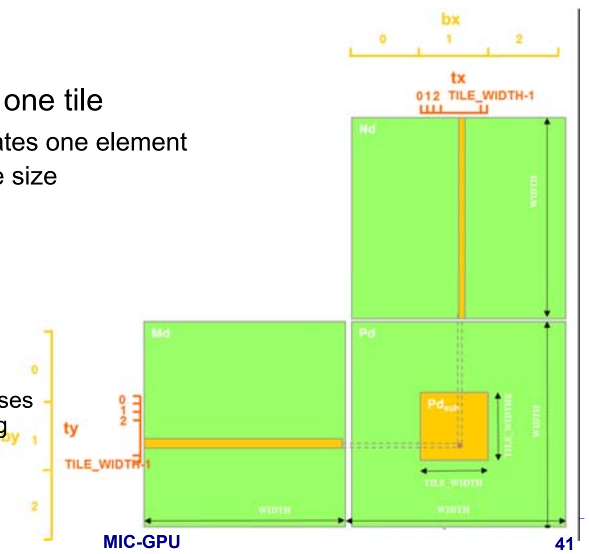
Each block calculates one tile

- Each thread calculates one element
- Block size equal tile size

Problem

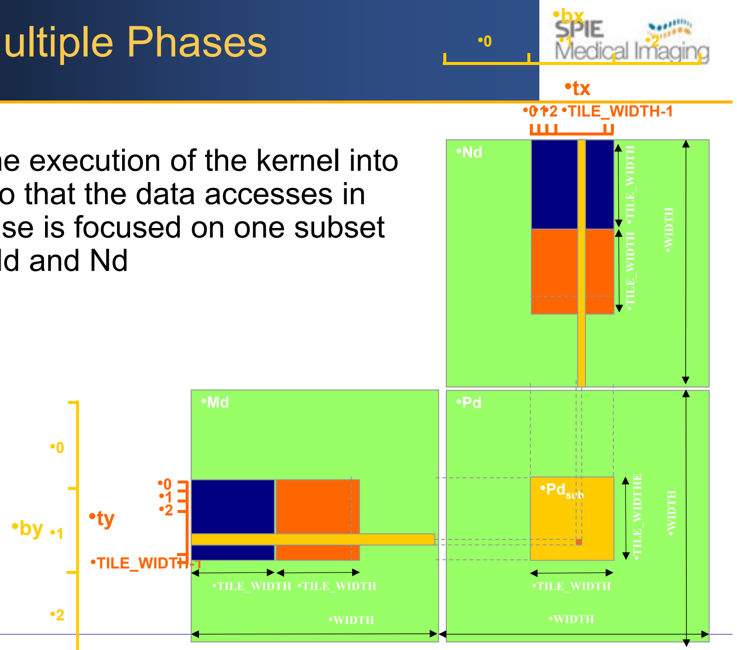
All threads access global memory for their input matrix elements

- Two memory accesses (8 bytes) per floating point multiply-add



Using Multiple Phases

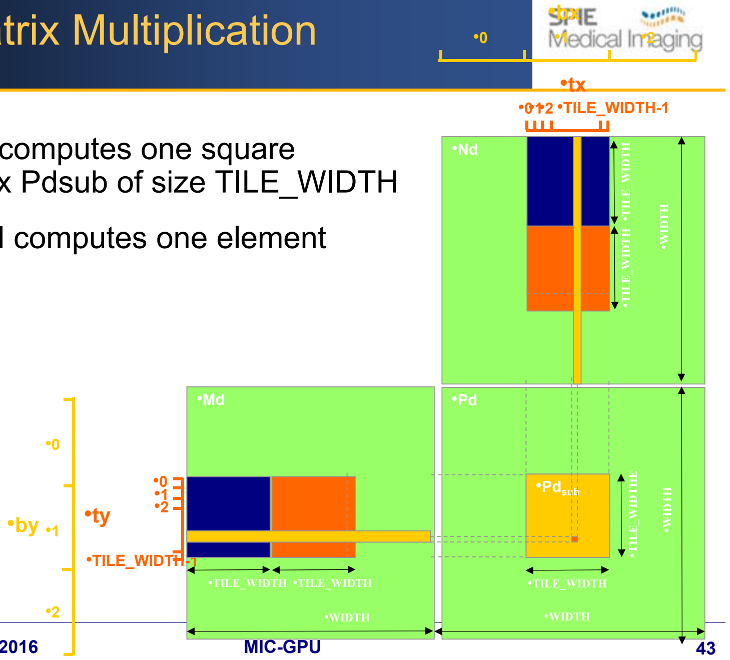
Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



Tiled Matrix Multiplication

Each block computes one square sub-matrix Pdsub of size TILE_WIDTH

Each thread computes one element of Pdsub



Kernel

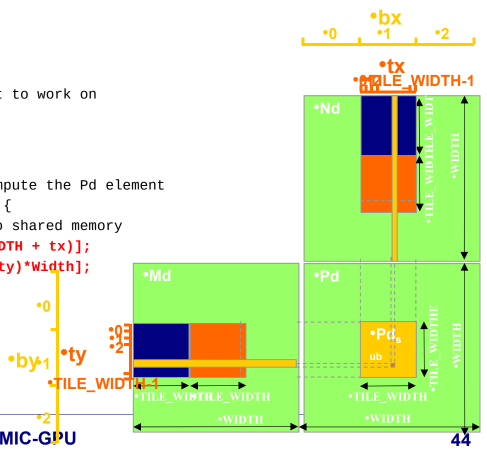
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

     for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
      Synchthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
```

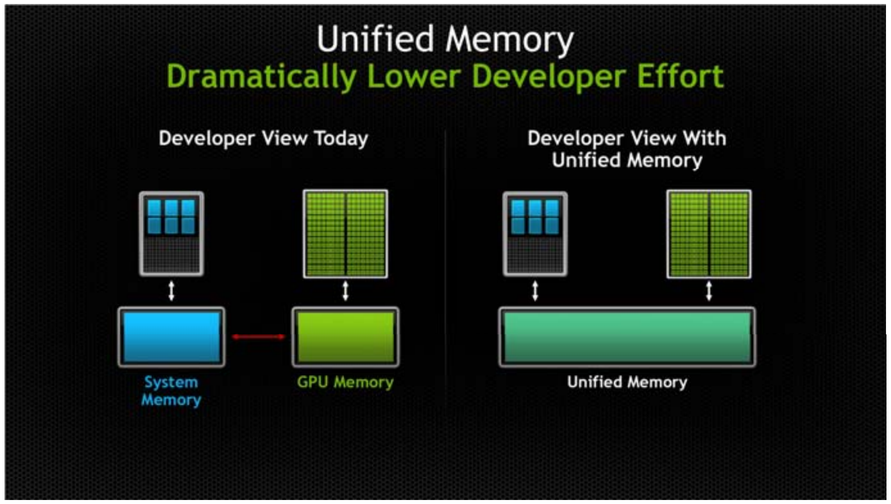


Locality

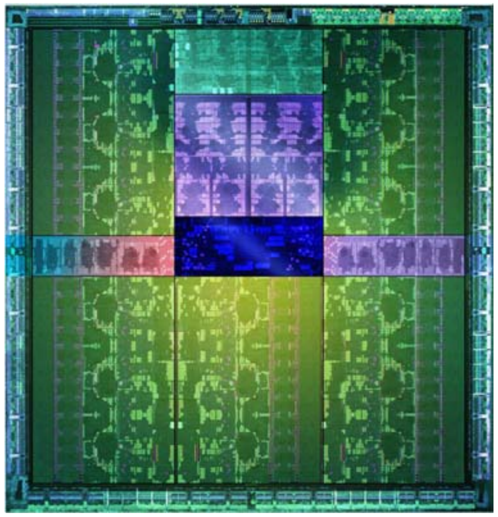
This scheme enforces *locality*

- focus of computation on a subset of data elements
- allows one to use small but high-speed memory for fast computation
- this exploit matches fast processors with high memory bandwidth and so maximizes the performance
- locality useful in any multi-core configurations

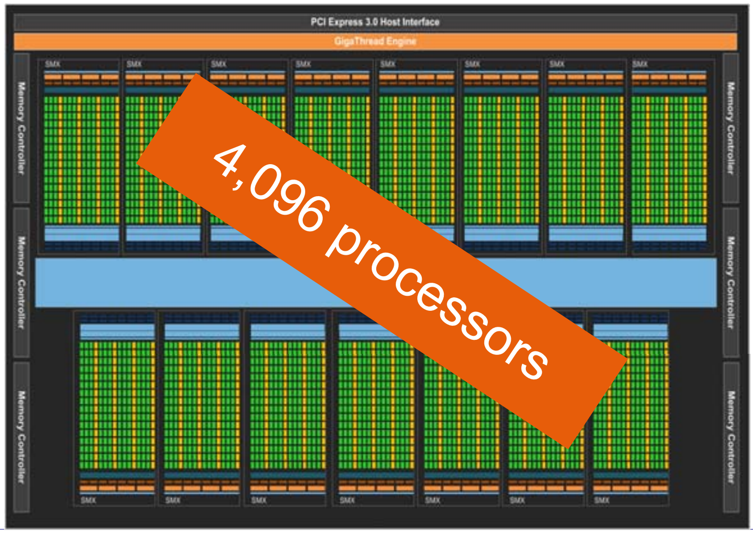
New Developments – CUDA 6



NVIDIA Kepler Architecture

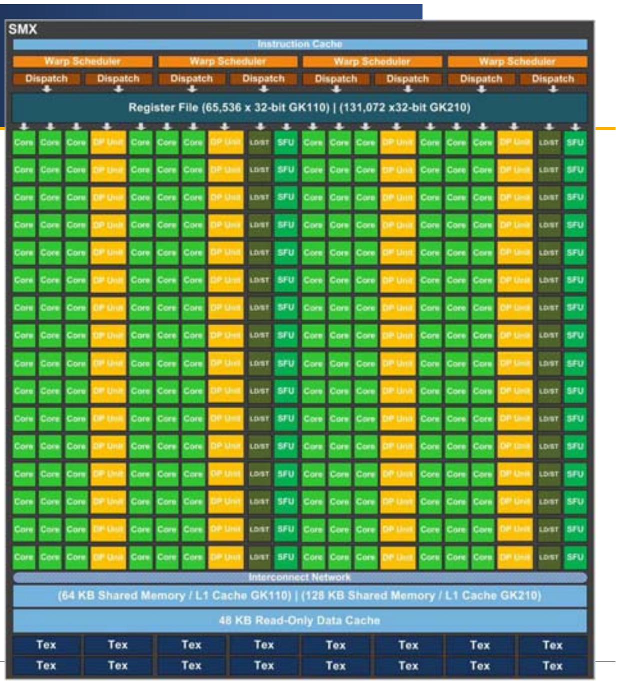

Kepler GK110 Die Photo

16 Streaming Multiprocessors (SMX)



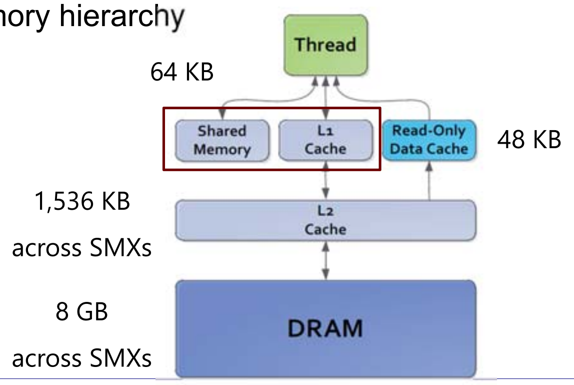
One SMX

- 192 single-precision CUDA cores
- 64 double-precision units
- 32 special function units (SFU)
- 32 load/store units (LD/ST)

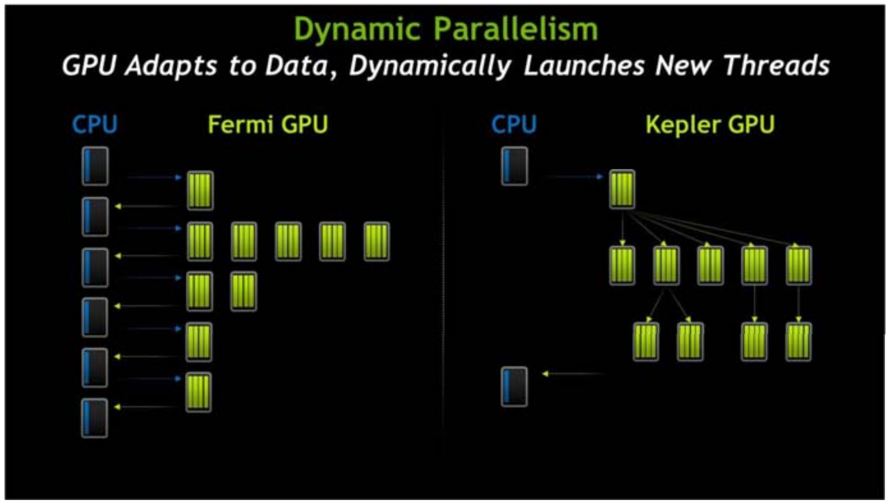

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit GK110) | (131,072 x32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconnect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

Features

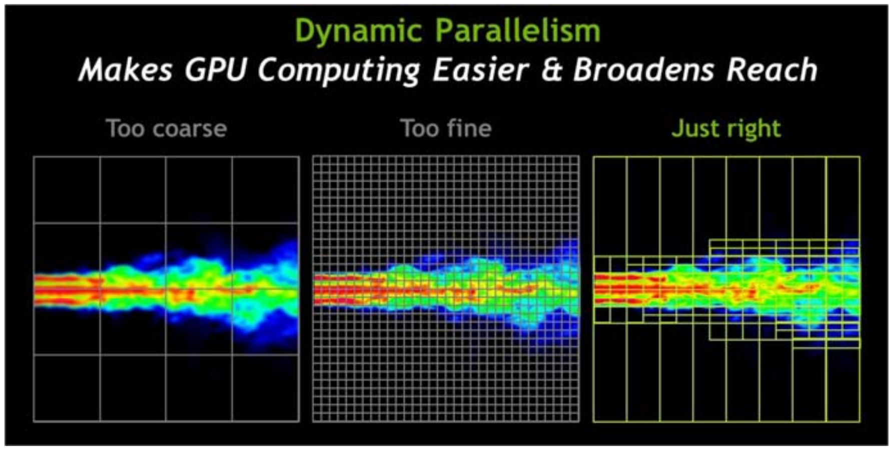
- Full IEEE 754-2008 compliant
- Atomic operations (Add, Max, Min, AND, OR. …)
- Sophisticated memory hierarchy
- ECC protection



Dynamic Parallelism



Dynamic Parallelism



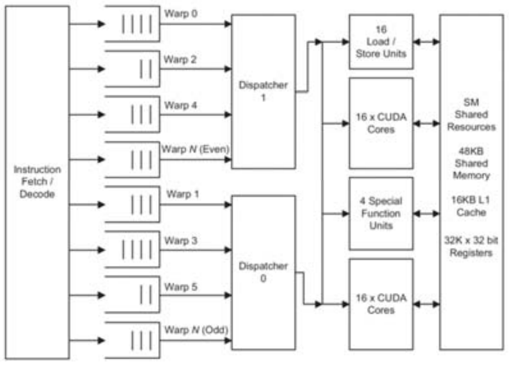
Application: Dynamic load balancing and grid refinement

Instruction Level Parallelism

SPIE Medical Imaging

Each Kepler SMX contains

- 4 Warp Schedulers
- each with dual Instruction Dispatch Units



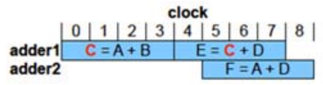
2 warp schedulers and single instruction dispatcher shown here

Instruction Level Parallelism (ILP)

SPIE Medical Imaging

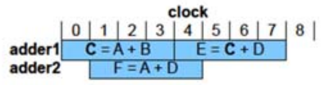
Dependencies not permitting ILP (9 clock cycles)

C = A + B
E = C + D
F = A + D



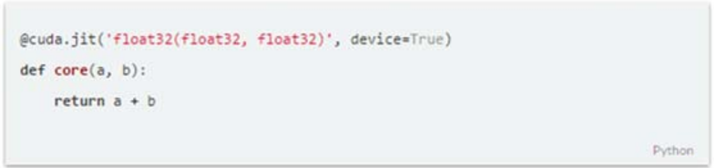
Instruction reordering for better ILP (8 clock cycles)

C = A + B
F = A + D
E = C + D



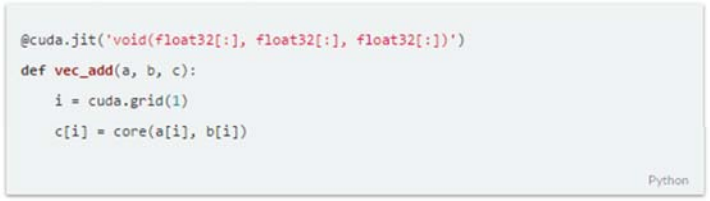
Computation and Load/Store: No ILP

SPIE Medical Imaging



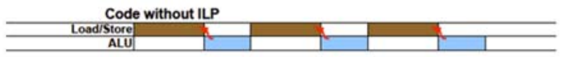
```python
@cuda.jit('float32(float32, float32)', device=True)
def core(a, b):
    return a + b
```

Python



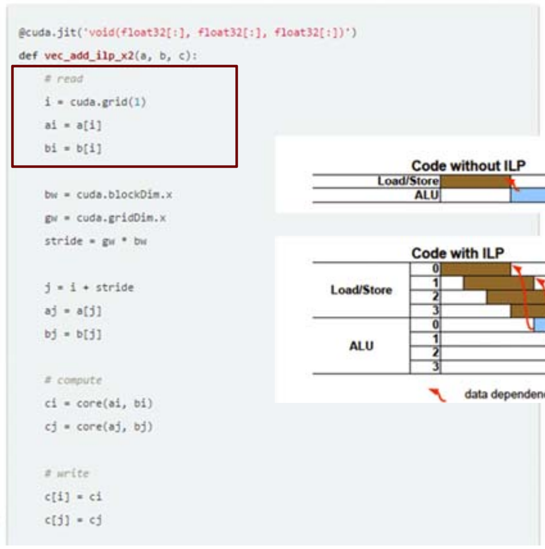
```python
@cuda.jit('void(float32[:], float32[:], float32[:])')
def vec_add(a, b, c):
    i = cuda.grid(1)
    c[i] = core(a[i], b[i])
```

Python



….With ILP

SPIE Medical Imaging



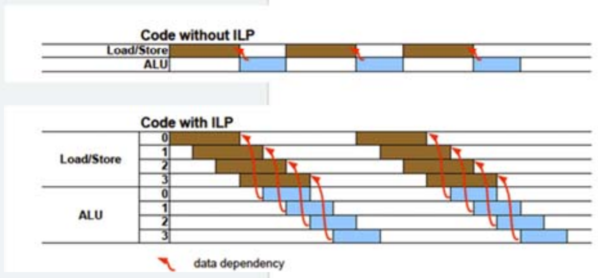
```python
@cuda.jit('void(float32[:], float32[:], float32[:])')
def vec_add_ilp_x2(a, b, c):
    # read
    i = cuda.grid(1)
    ai = a[i]
    bi = b[i]

    bw = cuda.blockDim.x
    gw = cuda.gridDim.x
    stride = gw * bw

    j = i + stride
    aj = a[j]
    bj = b[j]

    # compute
    ci = core(ai, bi)
    cj = core(aj, bj)

    # write
    c[i] = ci
    c[j] = cj
```



Common Optimizations

Loop unrolling

- reduces arithmetic and creates better vectorization

Loop fusion

- but check for dependencies

Thread fusion

- increases workload for threads

Kernel fusion

- encourages data reuse

Collaborative load into shared memory

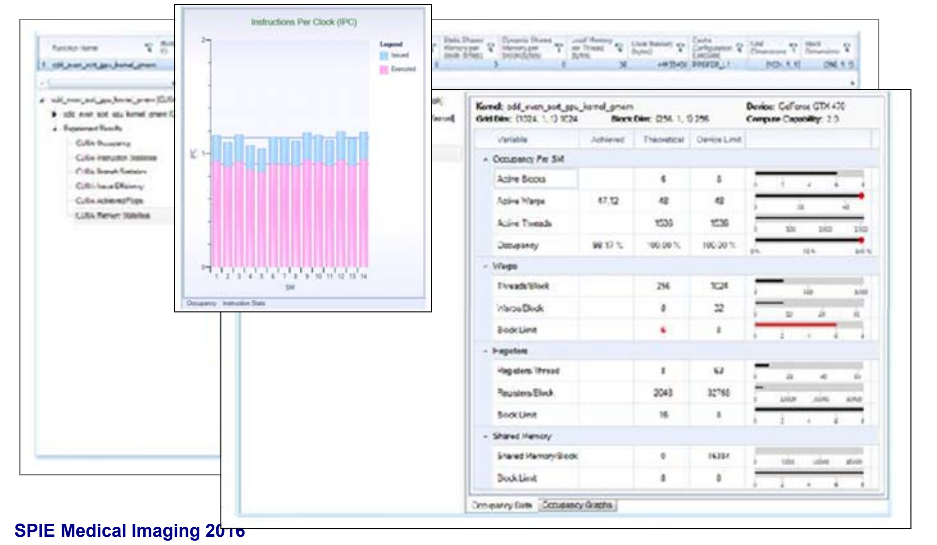
- when memory indexing is irregular

Larger blocks

- more threads can better hide memory latency
- but more threads require more registers → trade-off

but beware of hardware intelligence
some might already be done for you

NVIDIA Parallel Nsight (see demo)



High Performance Computing on the Desktop

PC graphics boards featuring GPUs:

- NVIDIA GeForce, ATI Radeon
- available at every computer store for less than $500
- set up your PC in less than an hour and play

the latest board:

NVIDIA GeForce GTX 980

"Just" Computing

Compute-only (no graphics): NVIDIA Tesla K and M series

K 80

True GPGPU

(General Purpose Computing using GPU Technology)

24 GB memory per card, 560 processors

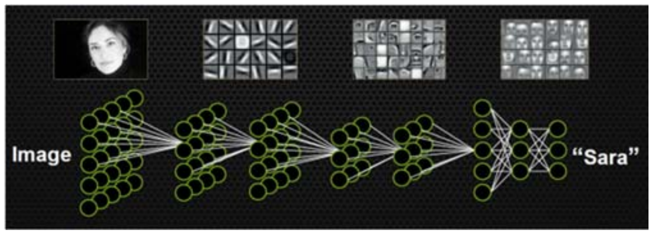
$4,000

Bundle 8 cards into a server: 5,280 processors, 192 GB memory

Recent Hot Topic: Deep Learning

A showcase application for GPUs

- DNN (deep neural networks)
- CNN (convolutional neural networks)
- GPUs shine especially in the training phase
- cuDNN = CUDA deep neural network library



Course Schedule

1:30 – 1:45:	Introduction (Klaus)
1:45 – 2:00:	Parallel programming primer (Klaus)
2:00 – 2:30:	GPU hardware and CUDA basics (Klaus)
2:30 – 3:00:	CUDA API, threads (Sungsoo)
	Coffee Break
3:30 – 4:00:	CUDA memory optimization (Sungsoo)
4:00 – 4:15:	CUDA programming environment (Sungsoo)
4:15 – 4:45:	Multi GPU (Sungsoo)
4:45 – 5:25:	Examples and demo (Klaus, Sungsoo)
5:25 – 5:30:	Closing remarks (Klaus)