GPU Computing
Supercomputing for the Masses

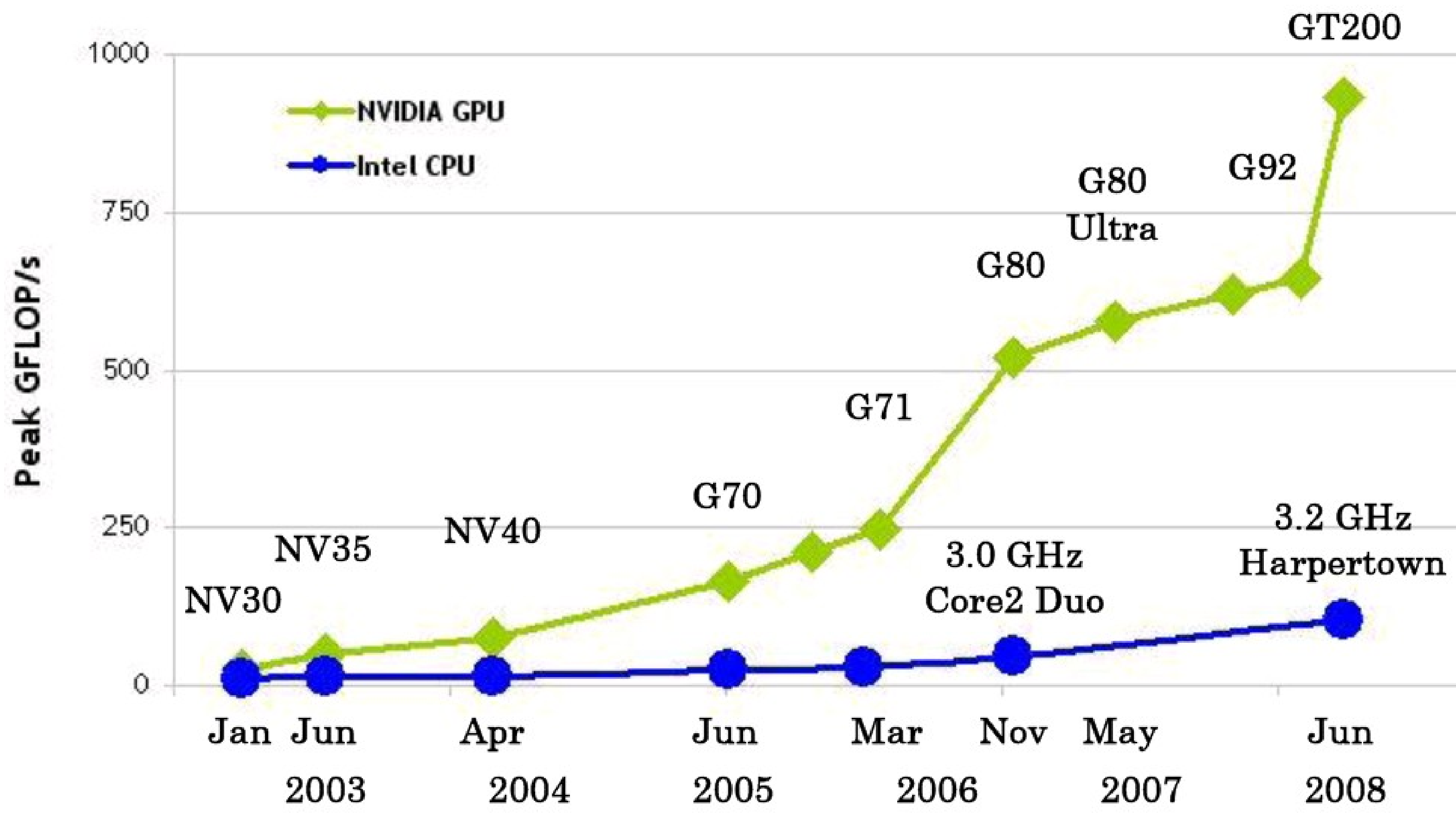
Problem

- „Free lunch is over“
- Demand for more computational power stays
- How to overcome this problem?
    - I.e. how to write software that benefits from the coming hardware performance improvements?
- Current trend: going multi-core
    - CPUs with 2, 4, 6, 8, 12, … cores
    - How to use them efficiently?

GPU Development

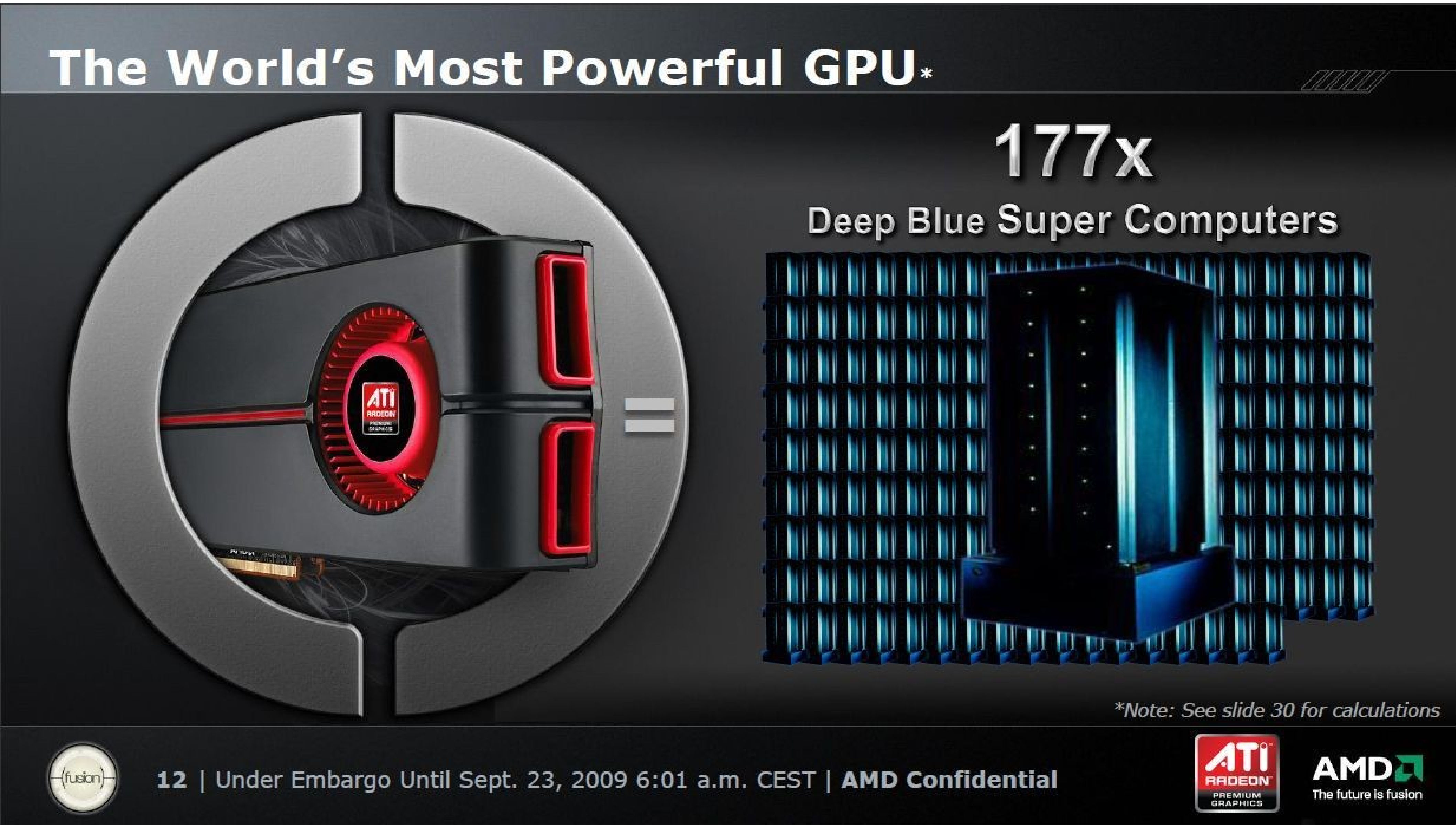
- GPU has evolved into a very powerful and flexible processor
  - Highly multicore (currently upto 1600 processors)
  - Very high memory bandwidth (150 GB/s)
  - Full IEEE 754-2008 support
  - ECC memory support
  - High market presence
- Performance still doubles with each generation

GPU vs. CPU



Peak GFLOP/s
1000
750
500
250
0
NVIDIA GPU
Intel CPU
NV30
NV35
NV40
G70
G71
G80
G80 Ultra
G92
GT200
3.0 GHz Core2 Duo
3.2 GHz Harpertown
Jan
Jun
Apr
Jun
Mar
Nov
May
Jun
2003
2004
2005
2006
2007
2008

What does that give us?



The World's Most Powerful GPU*

177x

Deep Blue Super Computers

*Note: See slide 30 for calculations

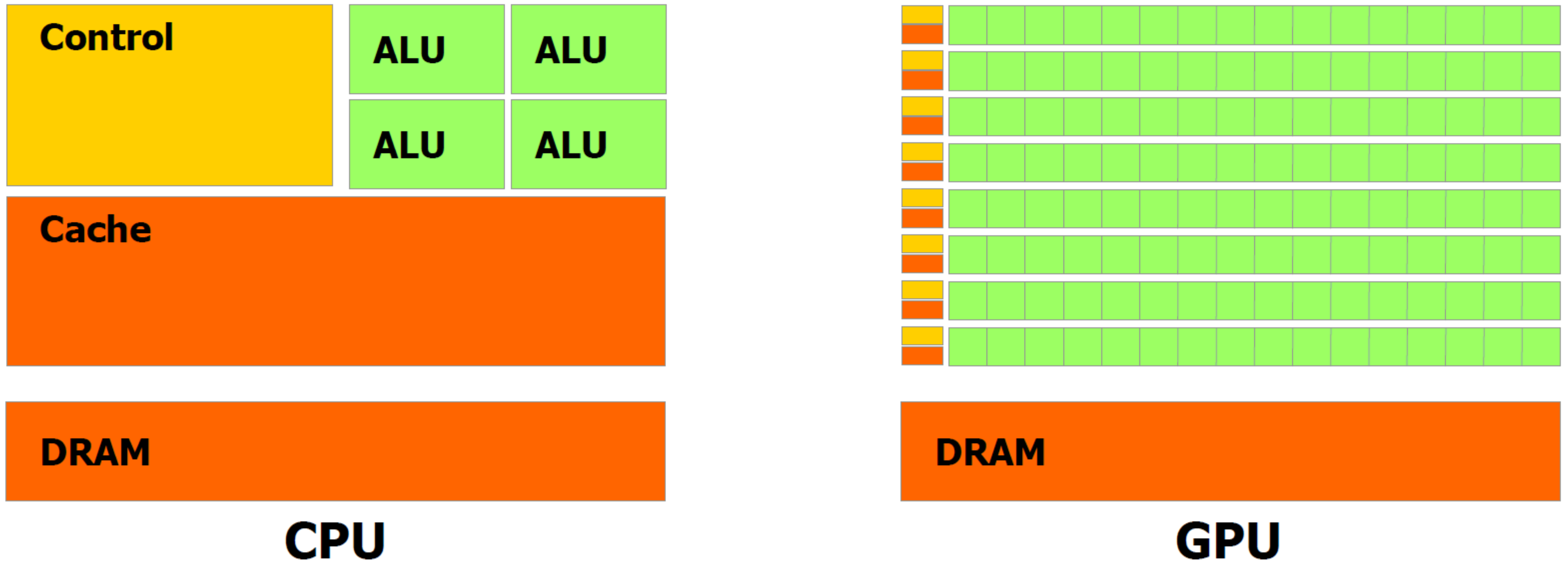
12 | Under Embargo Until Sept. 23, 2009 6:01 a.m. CEST | AMD Confidential

fusion
ATI RADEON PREMIUM GRAPHICS
AMD
The future is fusion

GPU vs. CPU

- ~2 Tflops GPU vs. 100 Gflops CPU
  - this difference created interesting dynamic
- Design philosophy
  - CPU optimized for sequential code performance
    - Sophisticated control logic
    - Large caches
    - Relaxed memory model → lower memory bandwidth
  - GPU optimized for throughput of massive number of threads
    - Minimizing control logic
    - Small caches
    - Simple memory model without legacy constraints

GPU vs. CPU



- How to access that raw power?
  - Traditionally extremely intricated (GPGPU)

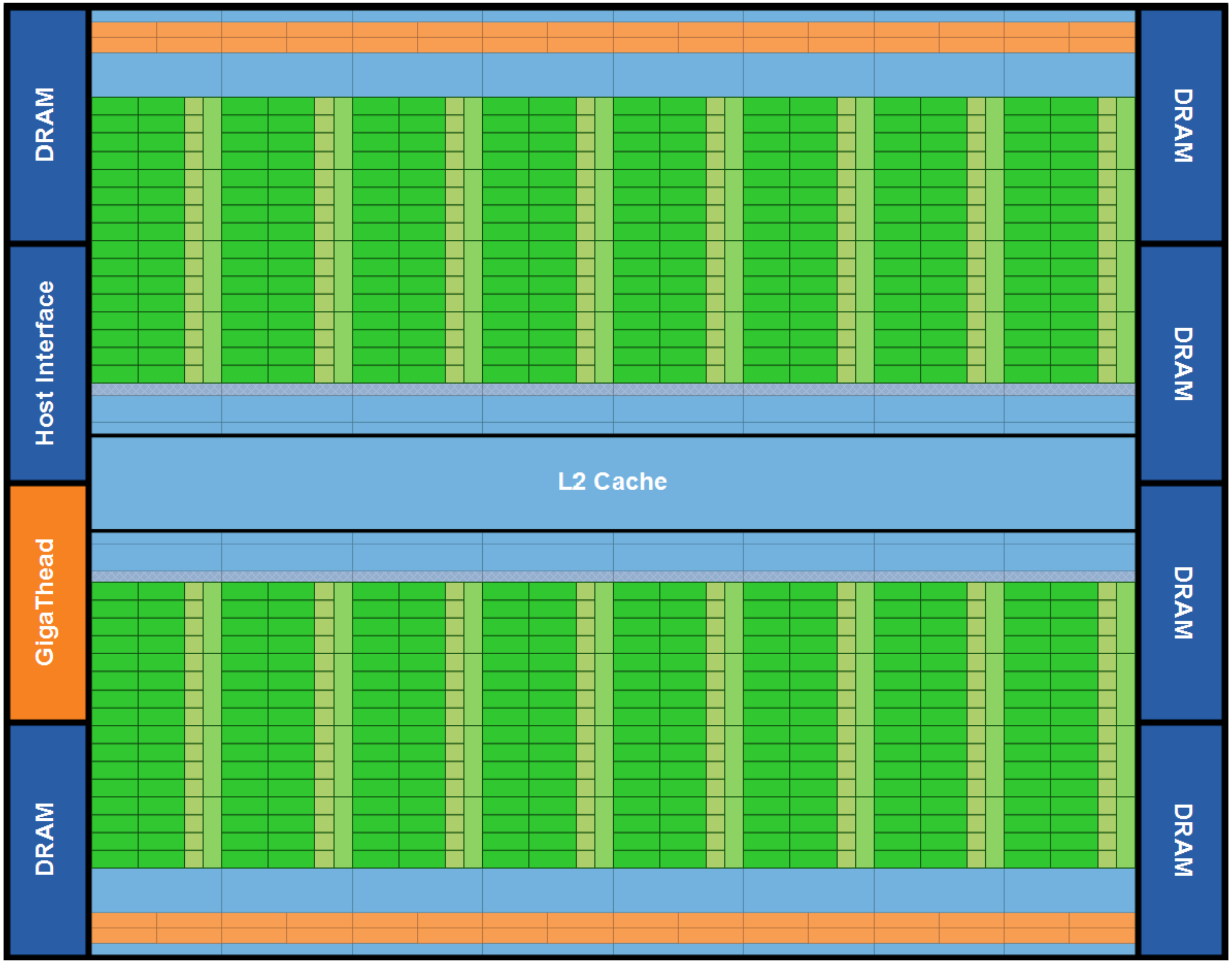
Compute Unified Device Architecture (CUDA)

- Introduced November 2006 by NVIDIA
  - GeForce 8800 - unified programmable processors
- Framework for parallel programming
  - Language
  - API
  - Runtime
  - Compiler
- Finally GPU fully programmable for general computations → GPU computing
- Upcoming standards OpenCL, DirectCompute

CUDA Platform Model

- One *host*
- One or more *devices*
  - Physically separate
  - Consisting of streaming multiprocessors of cores
- Host executes computations on device by invoking a *kernel* (kernel *launch*)
  - Large number of threads
  - In SPMD fashion

The Hardware



CUDA Execution Model

- Kernel
  - Executed on device
- Host program
  - Defining context for kernels
  - Manages execution of kernels
- Kernel launch (defining index space)
  - kernel instance (thread) for each point in ind. space
  - Execute same program

Data Parallel Programming Model

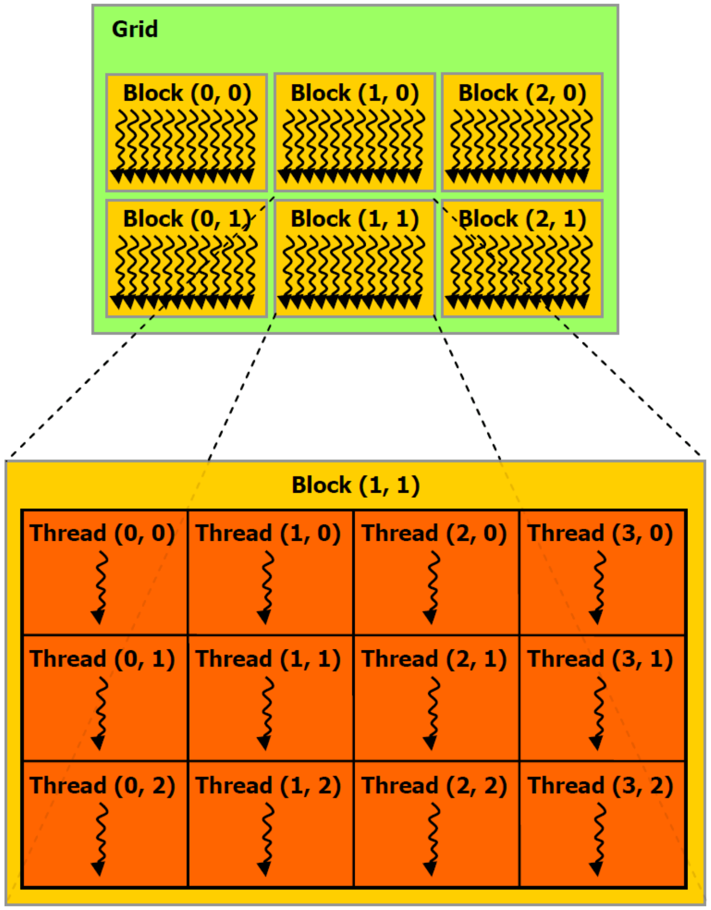
- Primary programming model
- Index space defines
  - *Threads*
  - Mapping of data onto threads
- Hierarchical data parallel model
  - Hierarchy of thread groups, *blocks*
  - Blocks organized inside a *grid*
  - Explicitly defined by user/host

CUDA Threads

- Extremely leightweight
- Each thread has its unique thread ID
- Via built-in variable *threadIdx*
  - 3-component vector, i.e. *threadIdx.x*, …
  - Threads form 1-, 2-, or 3-dim *thread block*s
  - Convenient mapping from domain, e.g. matrix
- Well-defined mapping threadIdx → thread ID
  1. (x) → x
  2. (x, y) → x + y*DIM_X
  3. (x, y, z) → x + y*DIM_X + z*DIM_Y*DIM_X

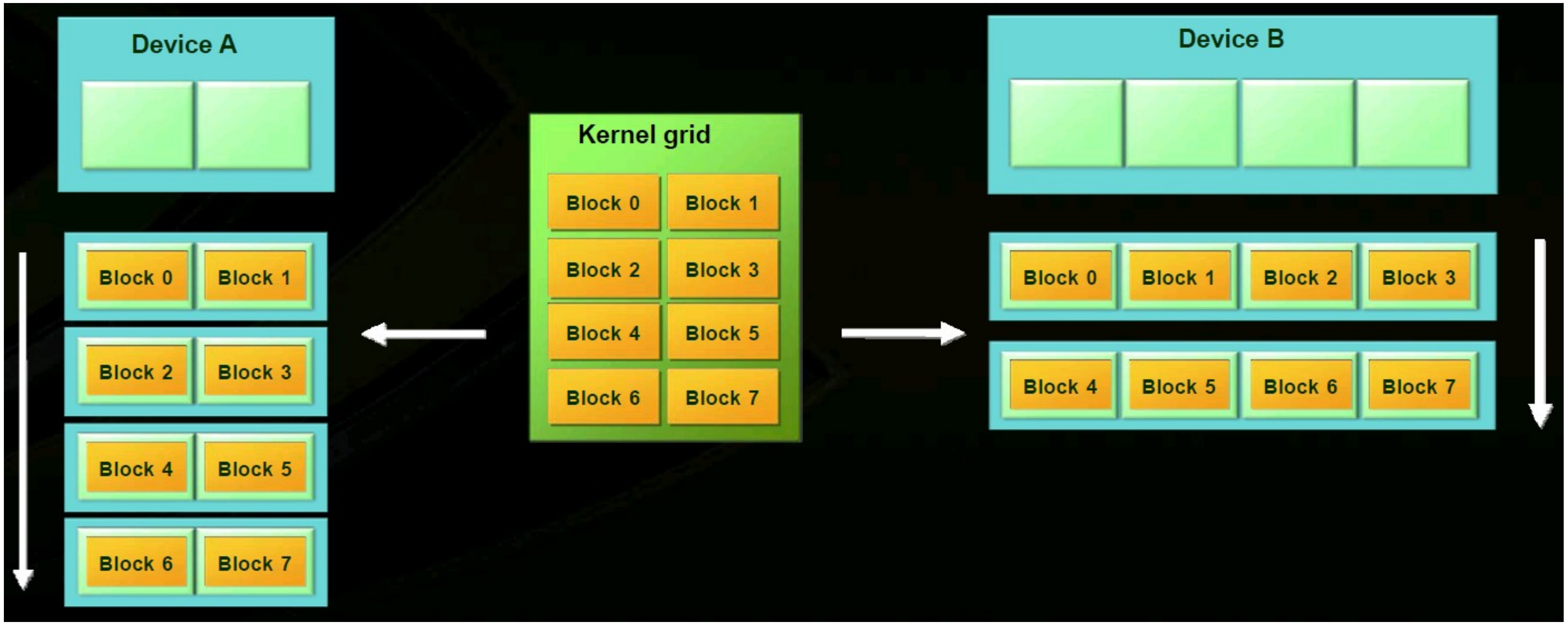
Grid of Thread Blocks

- Threads organized in *thread blocks* (3-dim)
- Thread blocks organized in 2-dim *grids*



Scalable Programming Model

- Coarse subdivision of index space into blocks
  - Threads inside using fine-grained data parallelism
- Blocks are independent from each other!
- Blocks can be scheduled in any order



CUDA Programming Model

- Main program runs on *host*
- Using GPU as co-processor (*device*)
  - Physically separate
  - CUDA threads execute on *device*
- Both host, device maintain own memory space
  - *Host memory vs. device memory*
  - Data can be copied between
  - I/O data has to be copied to and from device

CUDA Program

- Consists of both host and device code
  - Host code is C++
  - Device code is „C plus annotations“
- Alternating phases running on host and devices
  - Code exhibiting rich data parallelism implemented on device
  - Otherwise implemented on host

NVIDIA C Compiler

- nvcc separates both host and device code
  - Host code compiled on host's compiler
  - Device code compiled by nvcc for device
    - Embedded in host object file
  - Linking with CUDA runtime library cudart.dll
- Useful options
  - -deviceemu
  - -Xptxas -v

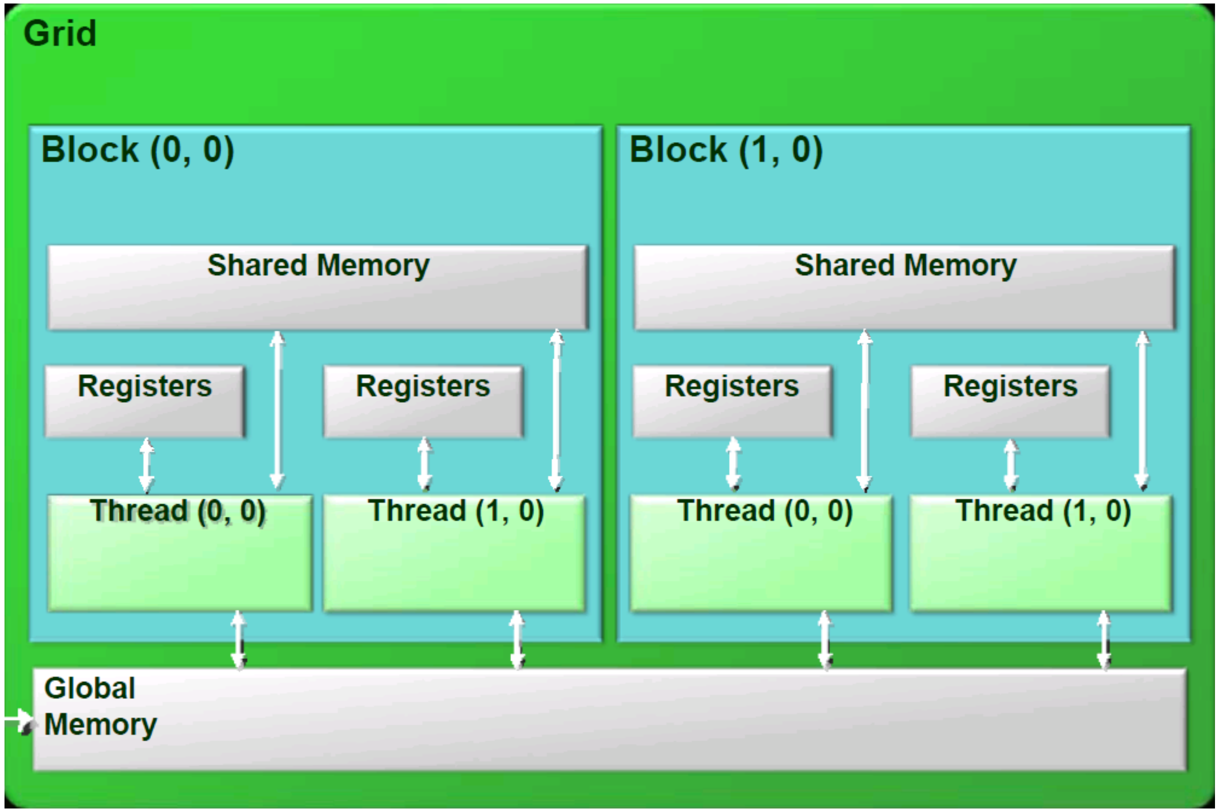
```
1>ptxas info    : Compiling entry function '_Z9matrixMulPfS_S_i'
1>ptxas info    : Used 10 registers, 2076+16 bytes smem, 12 bytes cmem[1]
```

CUDA Program Structure

```
int main() {
    // host data setup
    ...
    // device data setup
    ...
    // transfer of input data to device
    ...
    // execution on device
    ...
    // transfer of results from device
    ...
    // free device and host data
    ...
}
```

CUDA Memories

- Types
  - Constant
  - Global
  - Shared
  - Registers
- API calls
  - cudaMalloc, cudaFree
  - cudaMemcpy



CUDA Program Structure

```
int main() {
    // host data setup
    float* a_h = ...; float* b_h = ...; int N = DIM;
    // device data setup
    float *a_d, *b_d;
    cudaMalloc((void **) &a_d, sizeof(float)*N);
    cudaMalloc((void **) &b_d, sizeof(float)*N);
    // transfer of data to device
    cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
    // “execution” on device
   cudaMemcpy(b_d, a_d, sizeof(float)*N, cudaMemcpyDeviceToDevice);
    // transfer of results from device
    cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    // free device and host data
    cudaFree(a_d); cudaFree(b_d);
    ...
}
```

Kernel

- C function declared with __*global*__
  - New keywords __*global*__, __*device*__, __*host*__
- Specifies code to be executed by all threads
- Instance of single-program multiple-data parallel programming style (SPMD)

Kernel Launch

- Invoked (aka *launched)* by host code
- *Execution configuration*
  - Specifies nr. of CUDA threads executing the kernel
  - New syntax *foo<<<dimGrid, dimBlock>>>(...)*
- Executed as grid of threads in thread blocks
  - 3-dim thread block (*threadIdx*)
  - 2-dim grid (*blockIdx*)

Kernel Launch Example

```
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

...
// configure thread blocks
dim3 dimBlock(dim);        // == dimBlock(dim, 1, 1)
dim3 dimGrid(1, 1);        // == dimGrid(1, 1, 1) == 1
// execution configuration – at runtime kernel launch
VecAdd<<<dimGrid, dimBlock>>>(a_d, b_d, c_d);
...
```

Matrix Multiplication

```
void matrixMul(
     float* A, float* B, float* C,
     int dim)
{
  for (int r=0; r<dim; ++r) {
    for (int c=0; c<dim; ++c)
    {
      float value=0.f;
      for (int k=0; k<dim; ++k)
       value += A[r*dim+k]*B[k*dim+c];
      C[r*dim + c] = value;
    }
  }
}
```

```
__global__ void matrixMul(
     float* A, float* B, float* C,
     int dim)
{
  int r = threadIdx.y;
  int c = threadIdx.x;

  float value = 0.f;
  for (int k=0; k<dim; ++k)
    value += A[r*dim+k] * B[k*dim+c];
  C[r*dim + c] = value;
}
// configure thread blocks
dim3 block(dim, dim);
dim3 grid(1, 1);
// execution configuration
matrixMul<<<grid, block>>>(a, b, c);
```

Grid of Thread Blocks rev.

- Threads form a (upto) 3-dim thread block
  - Dictated by scarce HW resources
  - Currently max. 512 threads per block
- Blocks organized in (upto) 2-dim grid
- How to choose execution configuration?
- Number of blocks guided by size of the data!
- Remember: These blocks are executed independently, to guarantee transparent scaling.

Matrix Multiplication rev.

- Re-adjusting matrix multiplication configuration

```
int size = 20;
// configure thread blocks
dim3 blockDim(size, size);
dim3 gridDim(dim/size, dim/size);

// execution configuration
matrixMul<<<gridDim, blockDim>>>(a, b, c, dim);
```

Matrix Multiplication rev.

- How to access current block inside kernel?
- Via built-in variable *blockIdx*

```
__global__ void matrixMul(float* A, float* B, float* C, int dim)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col  = blockIdx.x*blockDim.x + threadIdx.x;

    float value = 0.f;
    for (int k=0; k<dim; ++k)
        value += A[row*dim+k] * B[k*dim+col];
    C[row*dim + col] = value;
}
```

Matrix Multiplication

n	1000	5000	7000	7500	8000
CPU simple	4	799	2583	3255	4460
GPU simple	0,1	14	43	55	

Memory Access Efficiency

```
__global__ void matrixMul(float* A, float* B, float* C, int dim)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col  = blockIdx.x*blockDim.x + threadIdx.x;
    float value = 0.f;
    for (int k=0; k<dim; ++k)
        value += A[row*dim+k] * B[k*dim+col];
    C[row*dim + col] = value;
}
```

- 2 global memory accesses vs 2 fp operations
- Compute to global memory access (CGMA) ratio
  - limits Gflops performance to 25% of mem bandwidth!
- 160GB/s → 40 Gflops << 1 Tflops!!

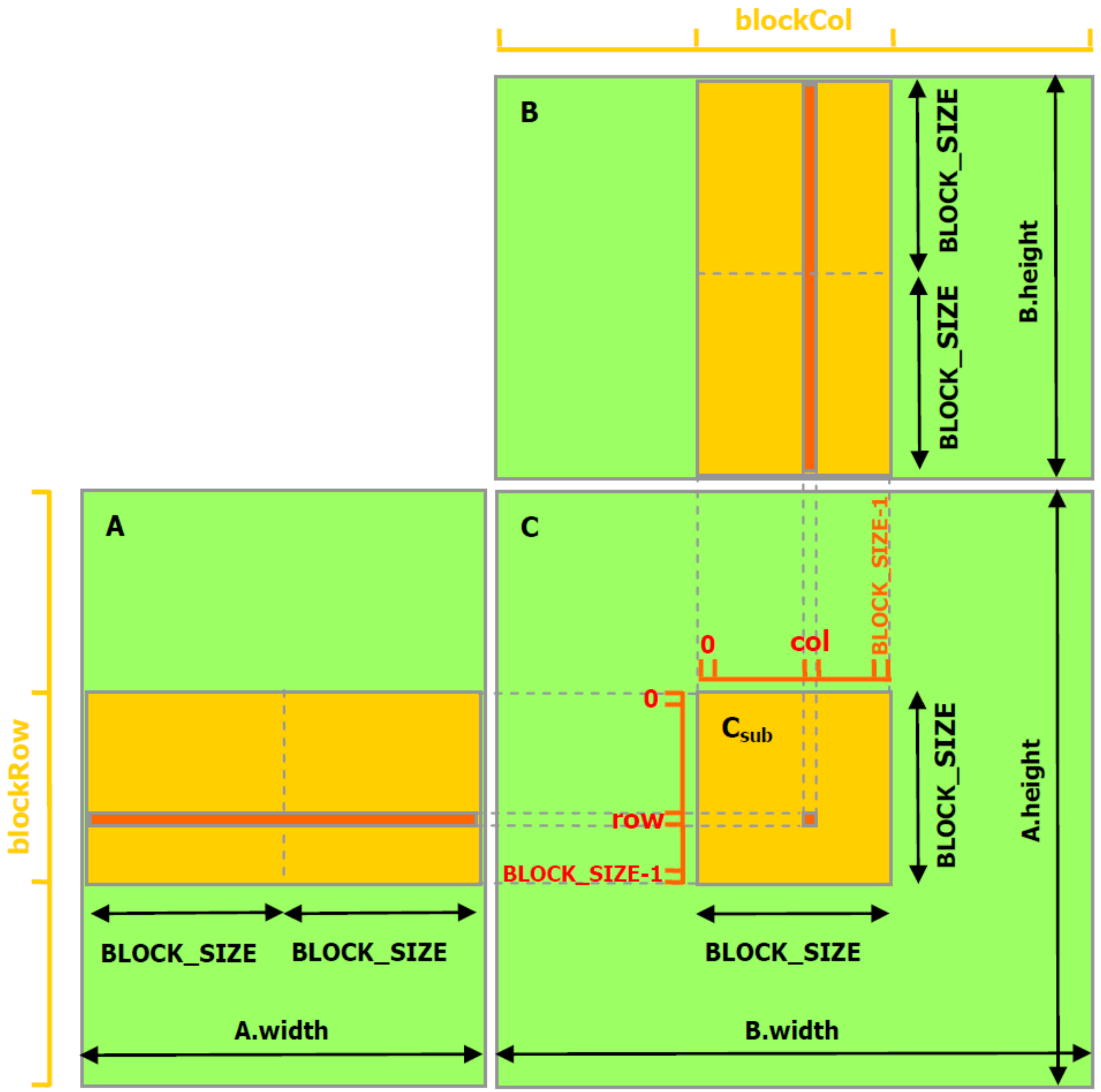
Shared Memory to the Rescue

- Observation in matrixMul()
  - Each global element is read dim times!
- Classic tradeoff
  - Global memory is large and slow
  - Shared memory is small (16384 bytes) but fast
- Strategy
  - Partition data into smaller *tiles* such that each tile fits into shared memory

Thread Cooperation and Shared Memory

- Again: Thread blocks are required to execute independently
- Threads <u>within one block</u>(!) can cooperate
  - Synchronizing their execution
  - Sharing data
- Intrinsic function __syncthreads()
  - Leightweight barrier at which all threads must wait
- Shared memory declaration __shared__
  - Low-latency memory near each processor core (L1)

Matrix Multiplication rev.



Matrix Multiplication rev.

```
__global__ void matrixMul(float* A, float* B, float* C, int dim) {
    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int row = by*TILE_DIM + ty; int col = bx*TILE_DIM + tx;
    // loop through all row/column elements
    float value = 0.f;
    for (int m=0; m<dim/TILE_DIM; ++m) {
        // collaborative loading of tiles into shared memory
        As[ty][tx] = A[row*dim + (m*TILE_DIM + tx)];
        Bs[ty][tx] = B[(m*TILE_DIM + ty)*dim + col];
        __syncthreads();
        for (int k=0; k<TILE_DIM; ++k)
            Value += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }
    C[row*dim + col] = value;
}
```

Matrix Multiplication

n	1000	5000	7000	7500	8000
CPU simple	4	799	2583	3255	4460
GPU simple	0,1	14	43	55	
GPU smarter	0,02	2	6	8	9,5

Matrix Multiplication

n	1000	5000	7000	7500	8000
CPU simple	4	799	2583	3255	4460
GPU simple	0,1	14	43	55	
GPU smarter	0,02	2	6	8	9,5
GPU sm 3.0		1,4	4	5	6

Further Optimization Routes

- Avoid *divergence* in SIMT execution!
  - Will result in repeated execution of the whole kernel
- Coalesce memory access to <u>global</u> memory!
  - Read from consecutive DRAM locations
- Data prefetching
- Improve instruction mix, loop unrolling
- Trade of scarce HW resources
  - Watch out for *performance cliffs*
- Try and measure!

Higher-level CUDA programming

- Several available libraries
  - CUBLAS
    - Implementation of BLAS on top of CUDA driver
  - CUFFT
    - Implementation of FFT
  - Thrust
    - CUDA library of parallel algorithms with an interface resembling the C++ STL

Thrust

```
int main() {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device @ 145M keys/second
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

OpenCL

- Initiated by Apple
- Open standard since last year
- Fundamental concepts map directly to CUDA
    - Work item ↔ thread
    - Work group ↔ block
    - NDRange ↔ grid
    - Kernel ↔ kernel
    - Device ↔ device
    - Host ↔ host

Conclusions?

- Easy to achieve impressive initial speedup
  - Iff there is data parallelism to begin with!
- Transparent scaling with new hardware
- Re-learning
  - But that adds to the fun
- Already in use in the real world

CUDA In Use

CUDA In Use

References

- Rob Farber's series „CUDA Supercomputing for the Masses“, thanks for letting me use his title! http://www.drdobbs.com/architecture-and-design/207200659
- D. Kirk, W. Hwu „Programming Massively Parallel Processors“, Morgan Kaufmann, 2010
- NVIDIA „CUDA Programming Guide“
- CUDA Zone on http://www.nvidia.com/object/cuda_home_new.html