Introduction to Numerical General Purpose GPU Computing with NVIDIA CUDA

Part 1:
Hardware design and programming model

Dirk Ribbrock

Faculty of Mathematics, TU dortmund
2016

Table of Contents

Why parallel processing?

Parallel Proc. Implementations

CPU vs. GPU

GPU hardware model

GPU‘s programming model

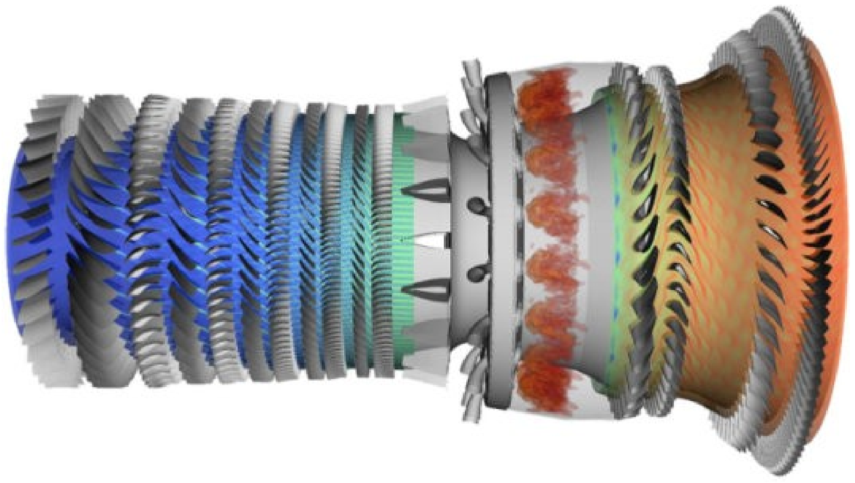
Scientific App. On GPU

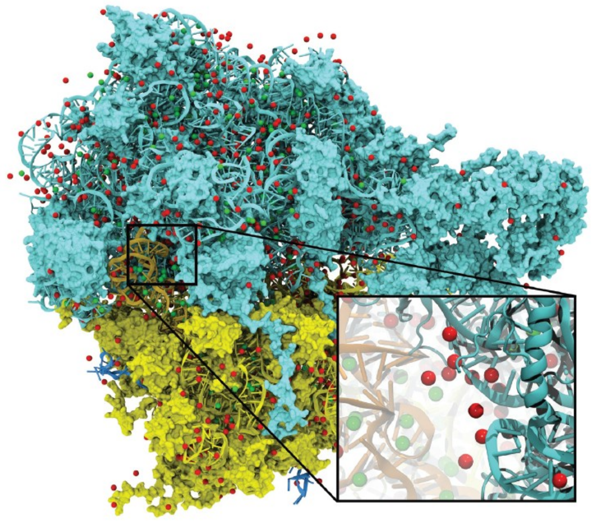
Performance Optimization

Why Parallel Processing?

Our main challenge in scientific computing

- Long simulation times on single Processors for large problems
- High computational cost to run on super computers
- Low or moderate grid resolutions to keep the cost low





Why Parallel Processing

▶ Moore’s Law

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to Continue”

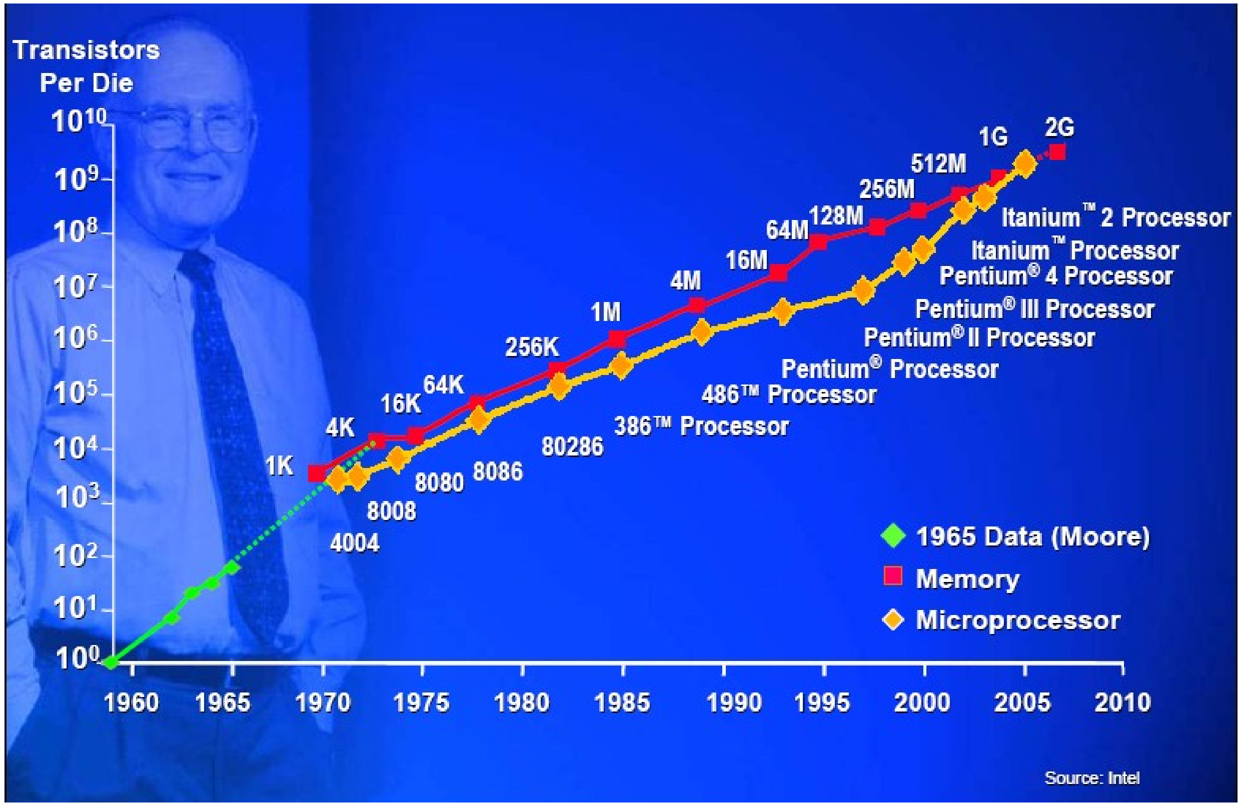
Gordon Moore (Intel), 1965

“*OK, maybe a factor of two every* ***two years.”***

Gordon Moore (Intel), 1975 [paraphrased]

Why Parallel Processing

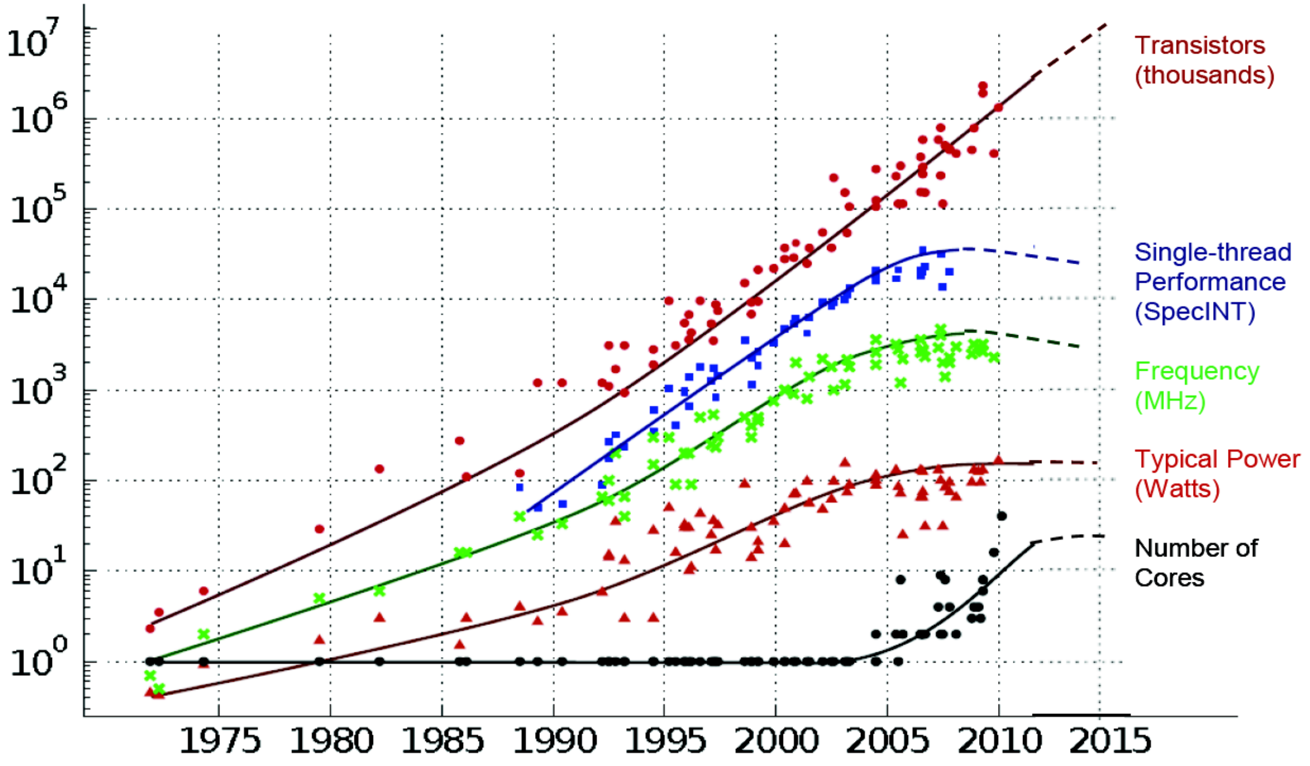
The Trend (1960-2005)



Why Parallel Processing

technische universität dortmund

The Trend, (1970-2010)



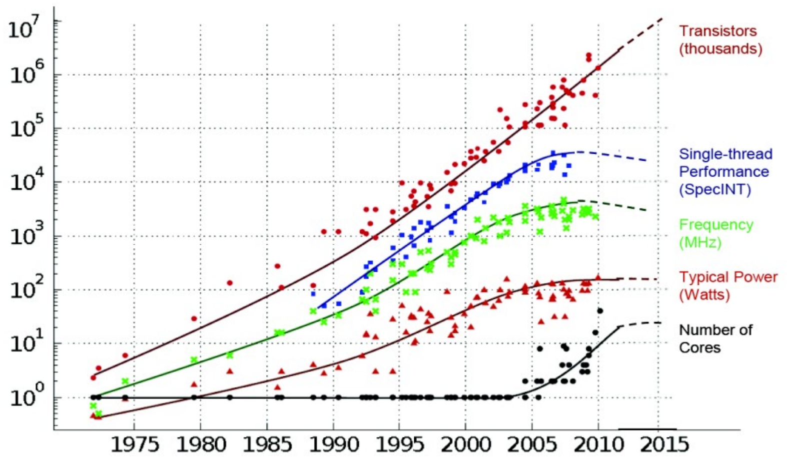
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Why Parallel Processing

technische universität dortmund

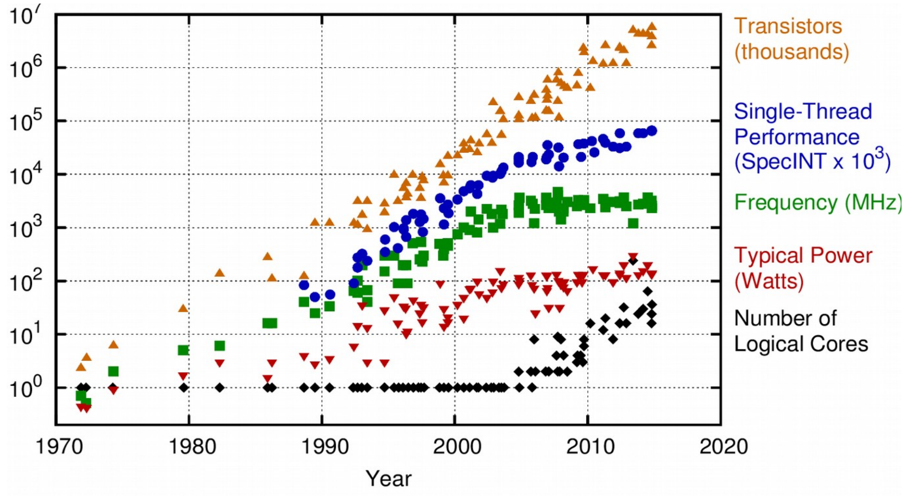
The Trend, (1960-2015)

Moor’s Extrapolation



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Actual Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Lesson’s learnt

- Number of transistors and cores have keep increasing!
- Performance/core is only slightly increased.
- Frequency has remained constant to control heat/power.
- One must go for parallel implementations.

Parallel Processing Implementations

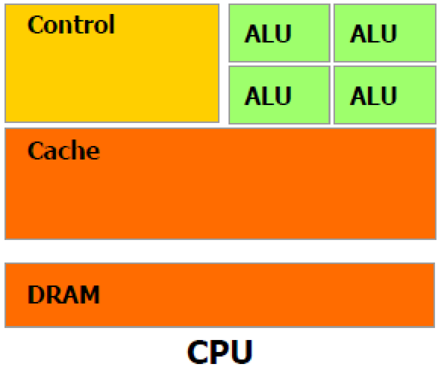
Major approaches

Distributed Memory	Message Passing Interface (MPI)
Shared Memory	OpenMP, Pthreads, Intel's TBB,…
GPGPU	**CUDA**, OpenACC, OpenCL,…

Parallel Design: CPU vs GPU

Single Instructions, Multiple Data (SIMD)

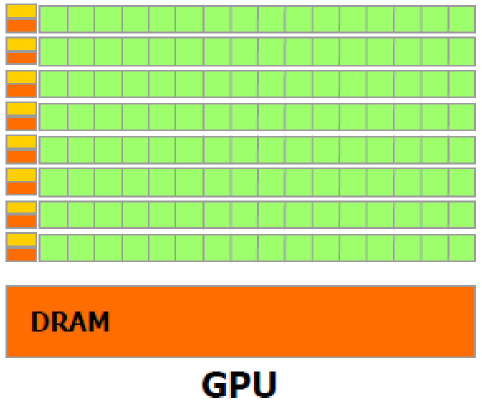
- Large data caching and flow control units
- Few number of ALUs (cores)
- **Example: Intel Xeon E5-2670 CPU**
  - 8 cores (16 threads)
  - 2.6 GHz
  - 2.3 billion transistors
  - 20 MB on chip cache
  - Flexible DRAM size



Parallel Design: CPU vs GPU

Single Instructions, Multiple Threads (SIMT)

- Small cache and control flow units
- Large number of ALUs (cores)
- **Example: Kepler K20x GPU**
  - 2688 (14 x 192) processor cores
  - 0.73 GHz
  - 28nm features
  - 7.1 billion transistors
  - 1.5 MB on-chip L2 cache
  - Only 6GB on chip memory



GPU Processing Model

GPUs are designed to apply the
same shading function
to many ***pixels*** simultaneously

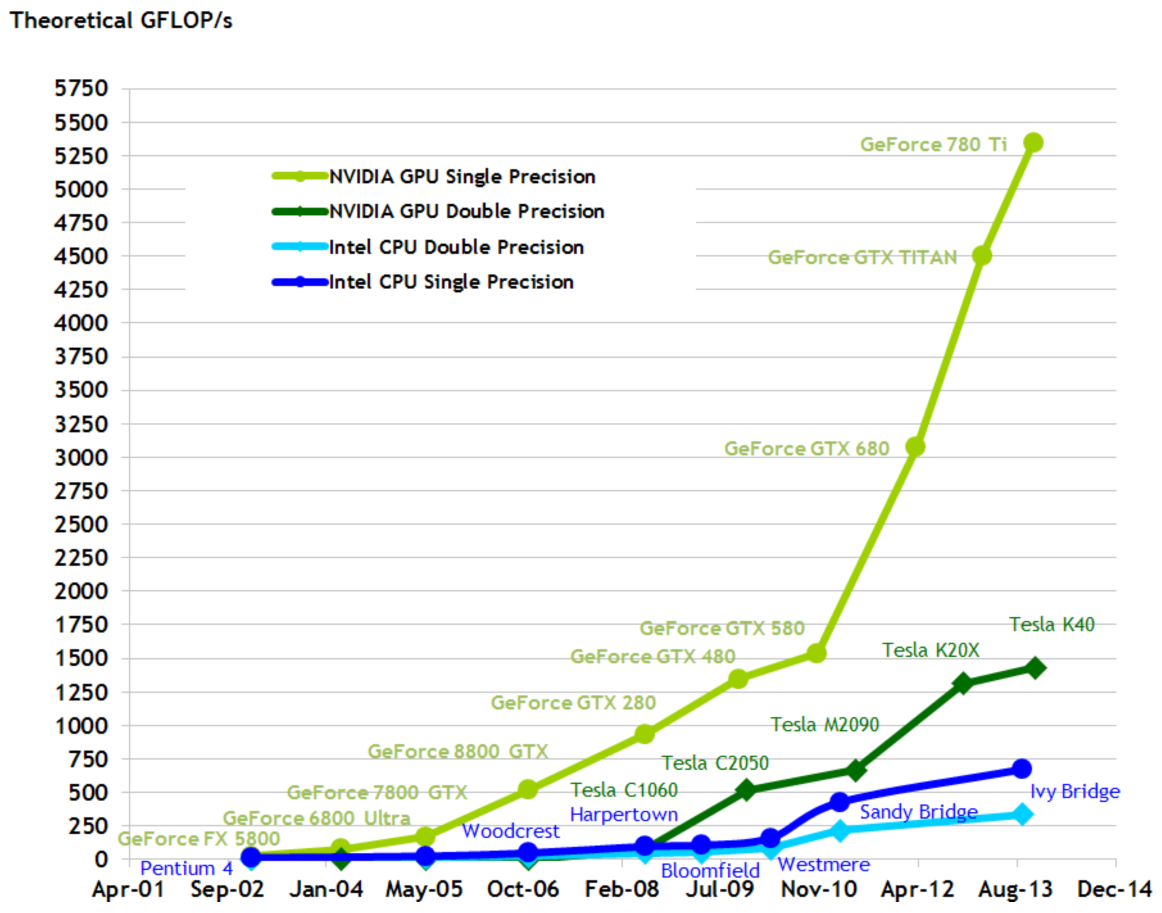
GPUs could be used to apply the
same function
to many ***data*** simultaneously

This is what most scientific computing need!

Parallel Design: CPU vs GPU

GPU Computational Capabilities

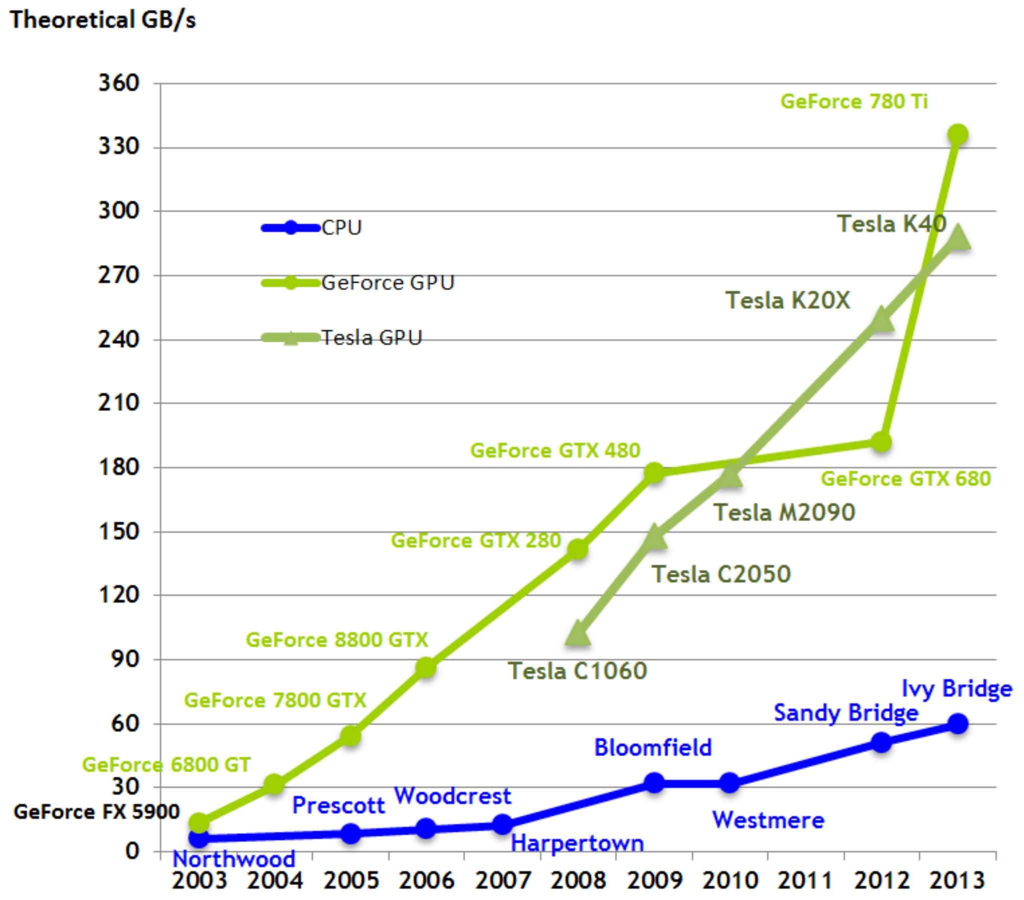
- High floating point power (5.3 TFlops in SP, 1.5 TFlops DP)



Parallel Design: CPU vs GPU

GPU Computational Capabilities

- High Memory Bandwidth (more than 300 GB/s)



GPU Architecture

technische universität dortmund

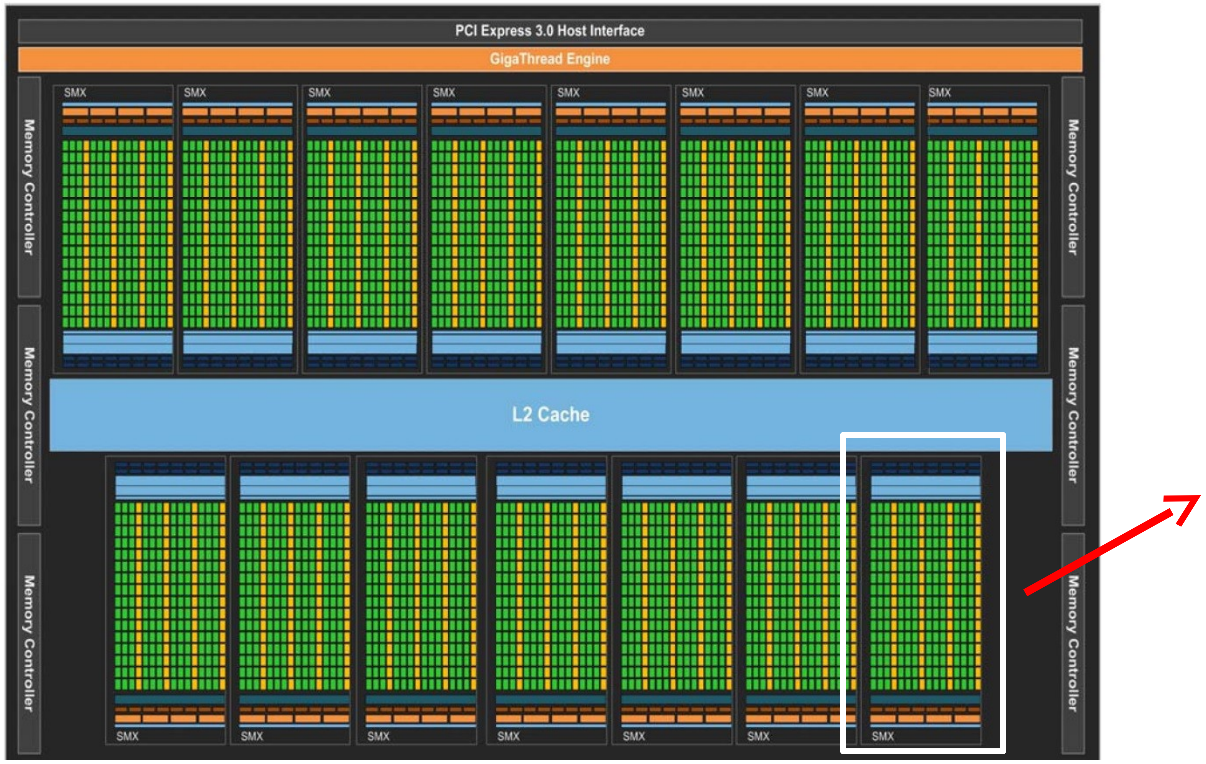
nVIDIA GPU Generations

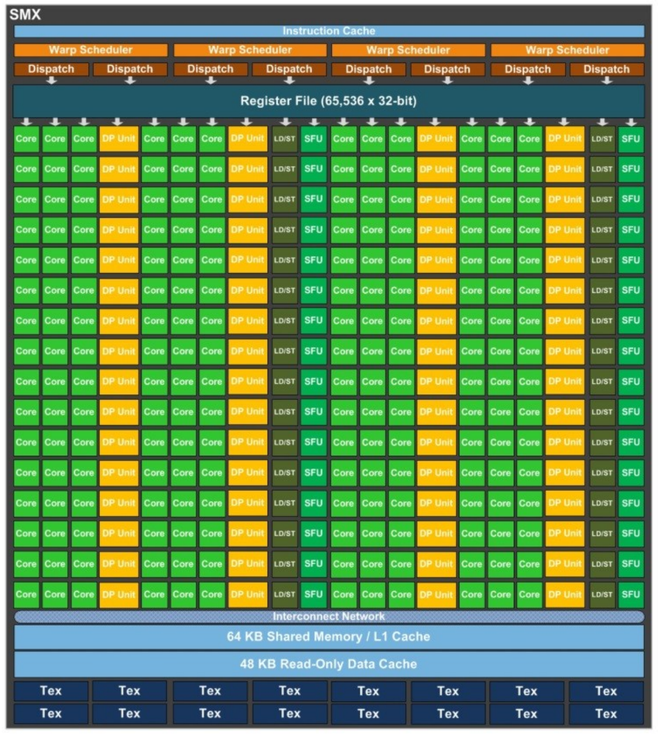
- GPUs come in different generations, e. g., Tesla, Fermi, Kepler,…
- Each is labeled with a specific Compute Capability, e.g., 1.x, 2.x, 3.x, …

GPU	G80	GT200	Fermi	Kepler
Transistors	681 million	1.4 billion	3.0 billion	7.0 billion
CUDA Cores	128	240	512 @ 1.15 GHz	2688 @ 0.73 GHz
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock	1344 FMA ops/clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock	2688 FMA ops/clock
Special Function Units (SFUs) / SM	2	2	4	32
Warp schedulers (per SM)	1	1	2	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB	Configurable 48 KB, 16 KB or 32 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB	Configurable 48 KB, 16 KB or 32 KB
L2 Cache	None	None	768 KB	1.5 MB
ECC Memory Support	No	No	Yes	Yes
Concurrent Kernels	No	No	Up to 16	Up to 32 + Dyn. Parallel
Load/Store Address Width	32-bit	32-bit	64-bit	64-bit

GPU Hardware Architecture

- Set of SIMD Streaming Multiprocessors (SMX)
- Each Multiprocessor has its own set of computational resources.



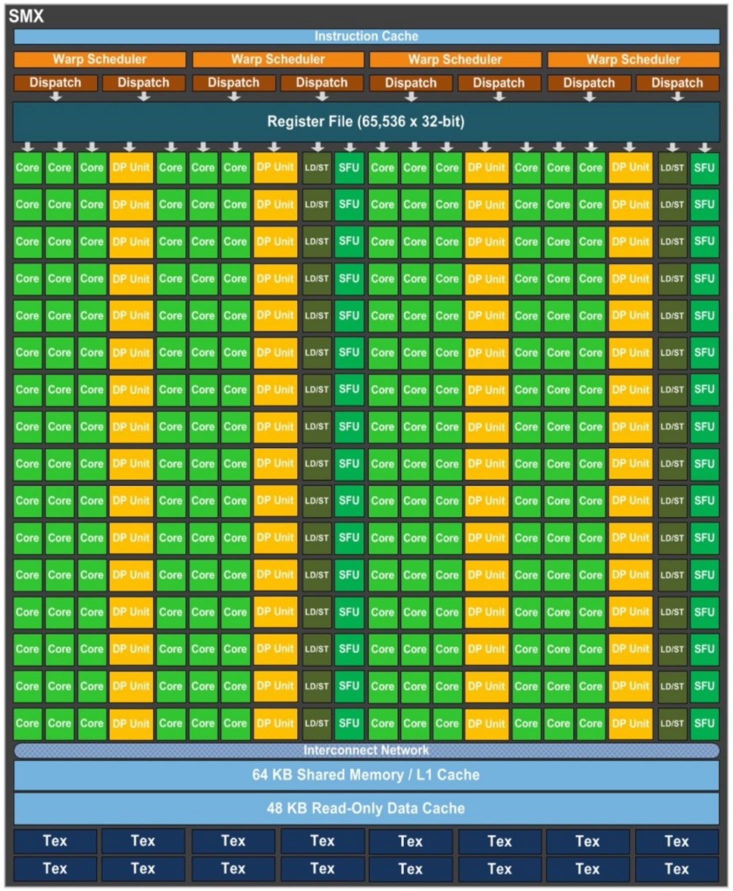

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

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

GPU Architecture

▶ Kepler Architecture (Compute Capability 3.x)

- 2688 cores are divided among14 SMXs, each having 192 processor cores.
- Each 3 cores serve as 1 double precision unit.
- Each SMX multiprocessor has a set of:
  - 65 KB L1 / Shared memory
  - 48 KB read-only caches
  - Constant and texture caches
  - Registers
- 32 special function units.

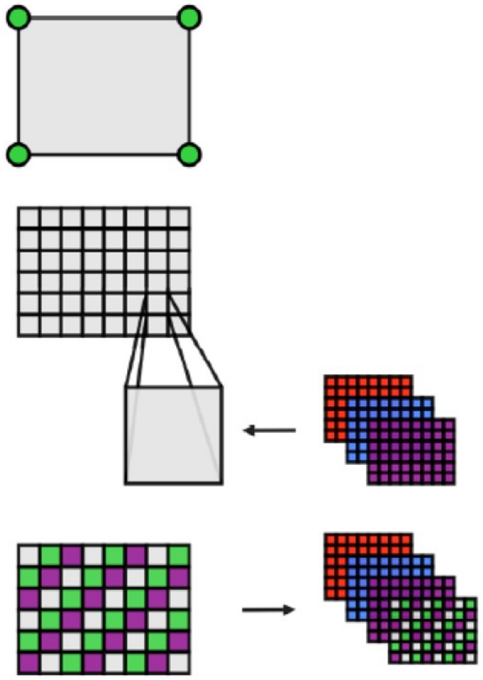

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

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

Programming On GPUs

Graphical Languages, e.g., OpenGL, DirectX,…

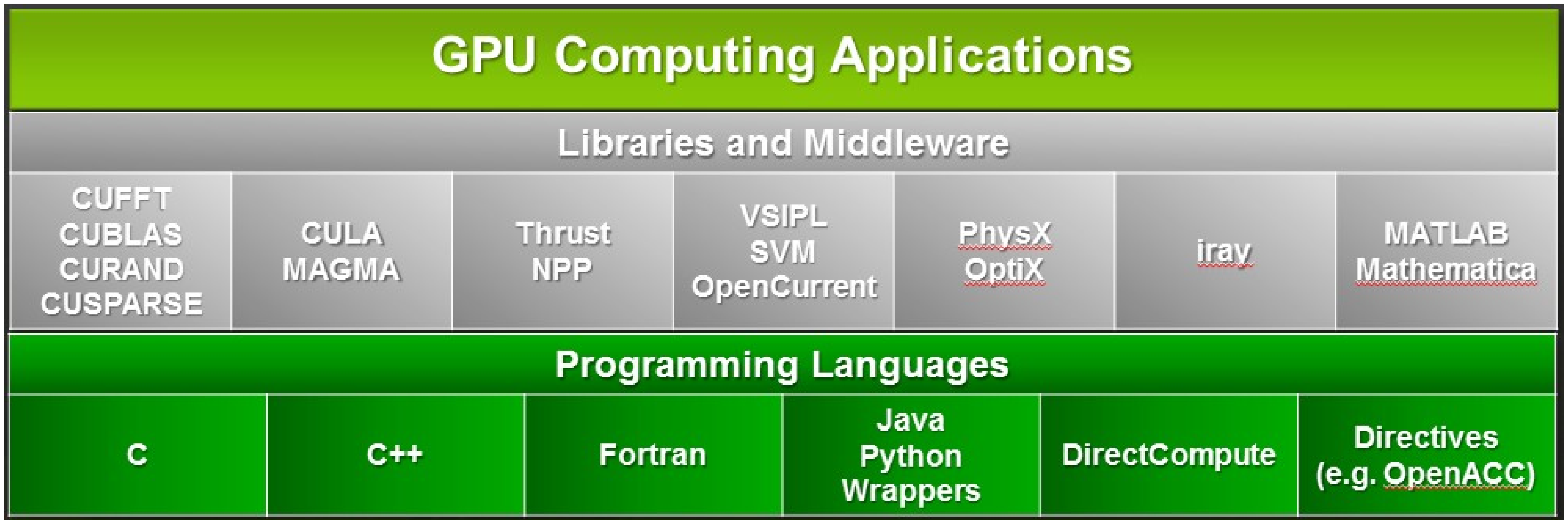
- Using graphics instructions for scientific calculations
- Very hard to develop codes for non-expert programmers
- Unable to fully exploit the computational power of GPUs
- Low overall efficiency



Courtesy, John Owens, UC Davis

Programming On GPUs

- **GPGPU Languages e.g. CUDA, OpenCL, OpenACC**
  - Designed specifically for scientific programming.
  - Relatively easy implementations.
  - Can extract almost all the power of hardware.
  - High numerical performances are then achievable.



GPU Computing Applications

Libraries and Middleware

CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica

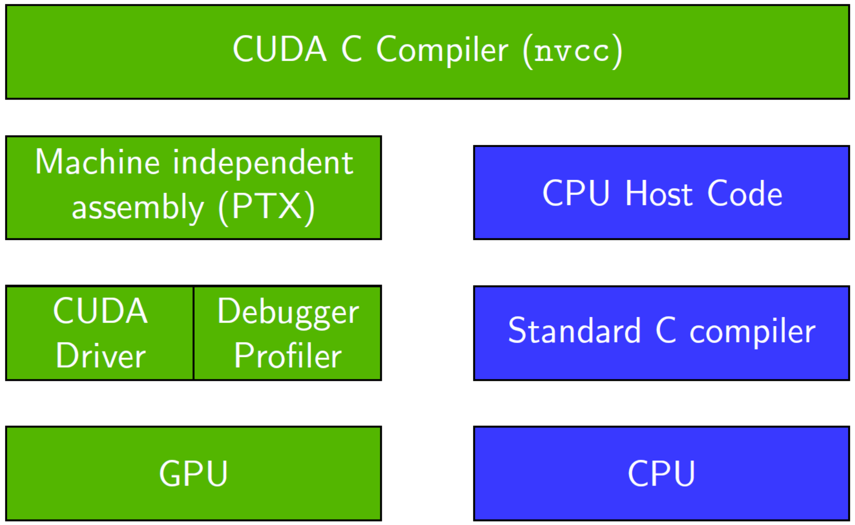
Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

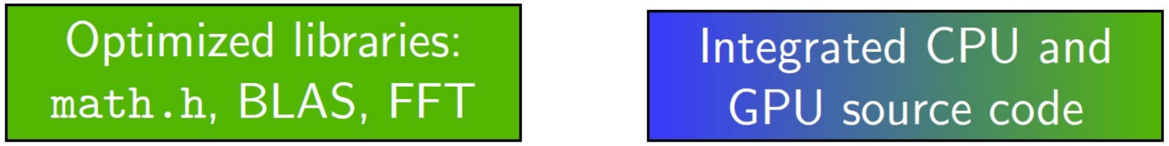
Compute Unified Device Architecture

CUDA Programming

CUDA Toolkit



CUDA Software Development Kit (SDK)

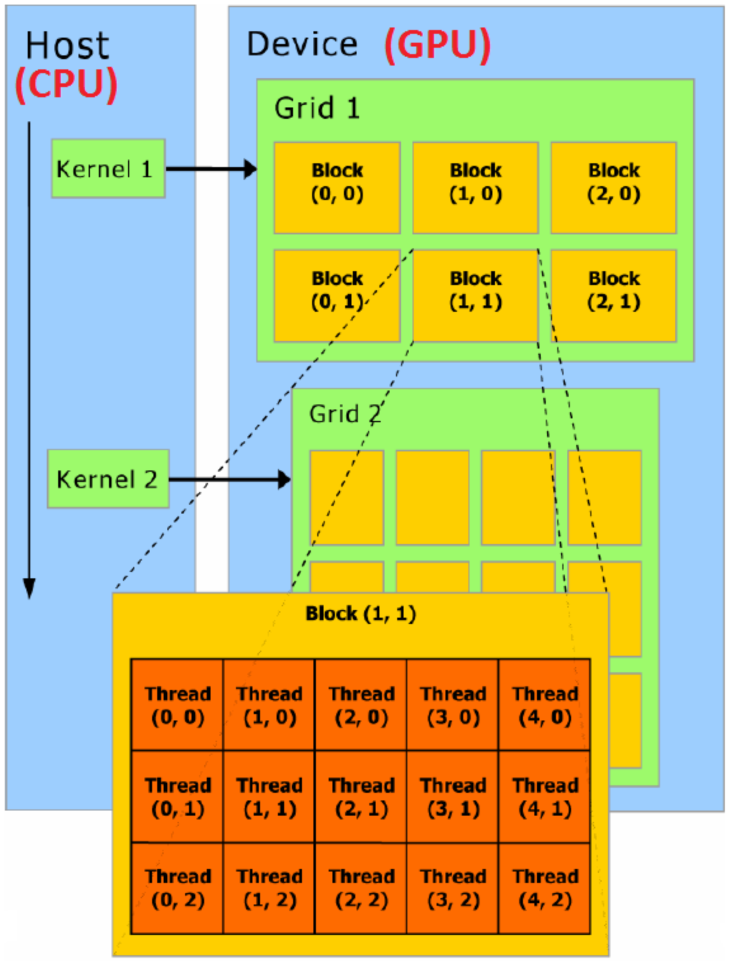

Optimized libraries: math.h, BLAS, FFT
Integrated CPU and GPU source code

CUDA Programming

Programming Model of CUDA

- Fine-grained parallelization by launching ***many*** active ***threads*** via ***kernels***
- Coarse grained parallelization via ***blocks*** and **grid.**
- Threads are grouped into ***blocks(1D, 2D or 3D)***
- Blocks are organized into a ***grid(1D, 2D or 3D)***
- **Kepler** supports max. ***2048*** active ***threads*** per SMX
- Threads are lightweight:
  – Small creation overhead
  – "instant "switching
  – Efficiency achieved through1000's of threads
- For a complete Device query see:

https://www.microway.com/hpc-tech-tips/nvidia-tesla-k20-gpu-accelerator-kepler-gk110-up-close/



CUDA Programming

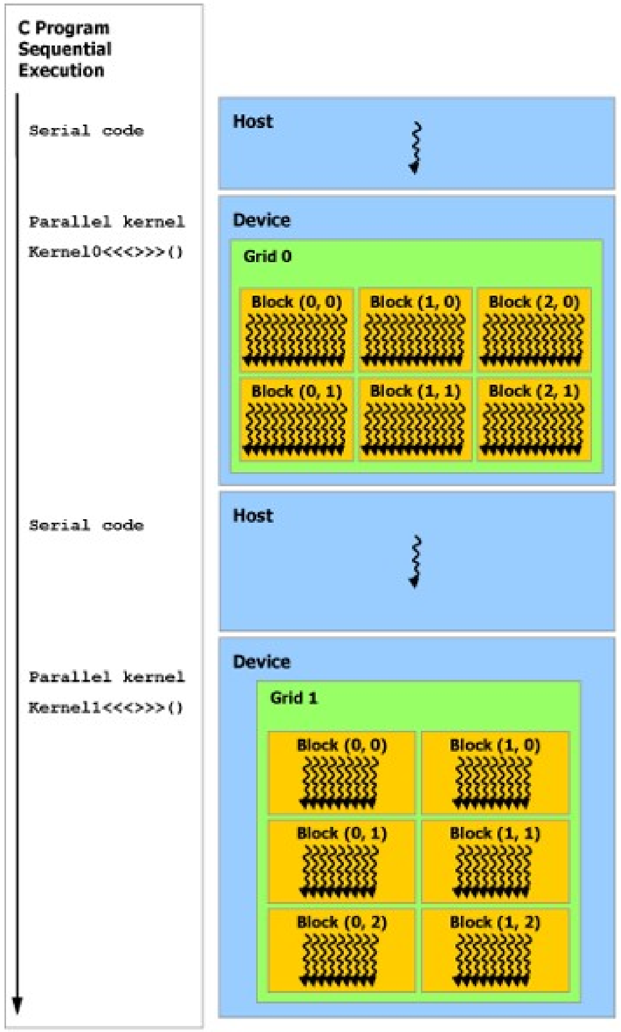
Essential CUDA Extensions to C/C++

- Kernel execution directives
  - `myfunction<<<GridDim, BlockDim>>> (…)`
- Built-in variables for grid/block size and block/thread index
  - `threadIdx.x` , `threadIdx.y` , …
  - `blockIdx.x` , `blockIdx.y`, … , `blockDim.x`, …
- Function type qualifiers
  - Specify where to call and execute a function
  - `__device__` , `__global__` and `__host__`
- Variable type qualifiers
  - `__device__` , `__constant__` and `__shared__`

CUDA Programming

Heterogeneous workflow

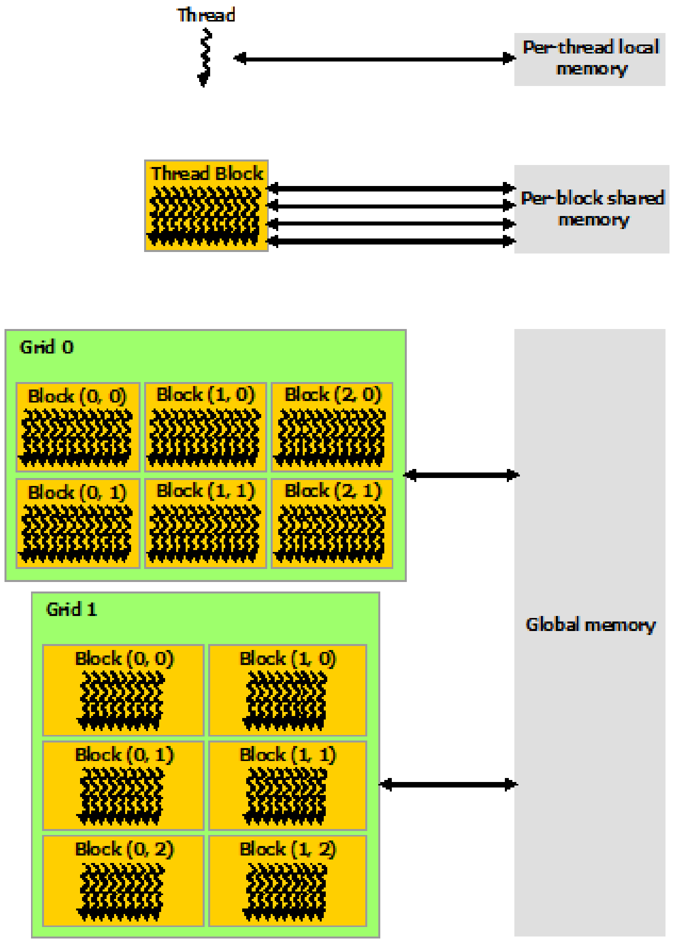
- kernels execute on a GPU and the rest of the C program executes on a CPU.
- CUDA threads execute on a physically separate device.
- Allows for asynchronous pre- and post-processing on CPU.
- CUDA assumes that both the host and the device maintain their own separate memory spaces in DRAM.



CUDA Programming

The memory hierarchy

- The grid of blocks in each kernel has access to *global memory.*
- Data dispatched from global memory is stored in fast *L2 cache lines.*
- Threads within a block can read from and write to *shared memory* asynchronously.
- Each thread has access to on-chip *local memory*.
- Different memories make up the so-called *device memory.*



What happens to a block?

- **Software**
  - Threads from *one block* may cooperate:
    - using data in shared memory
    - can get synchronized.

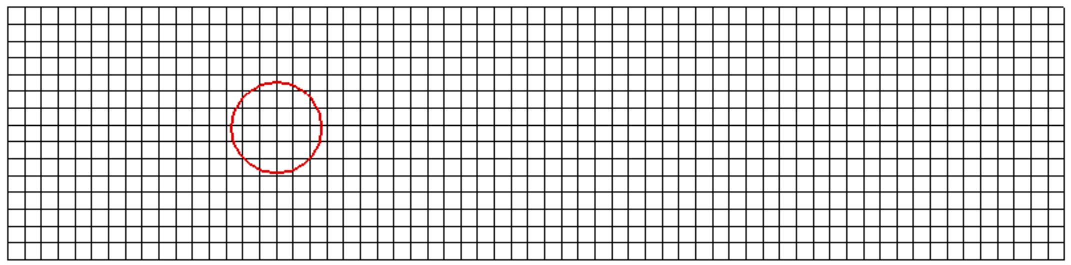
- **Hardware**
  - A block runs on *one multiprocessor.*
  - Hardware is free to schedule any block on any multiprocessor
  - More than one block can reside on one multiprocessor
  - A block is split into multiple *warps* of 32 threads (details given later).

CUDA Programming

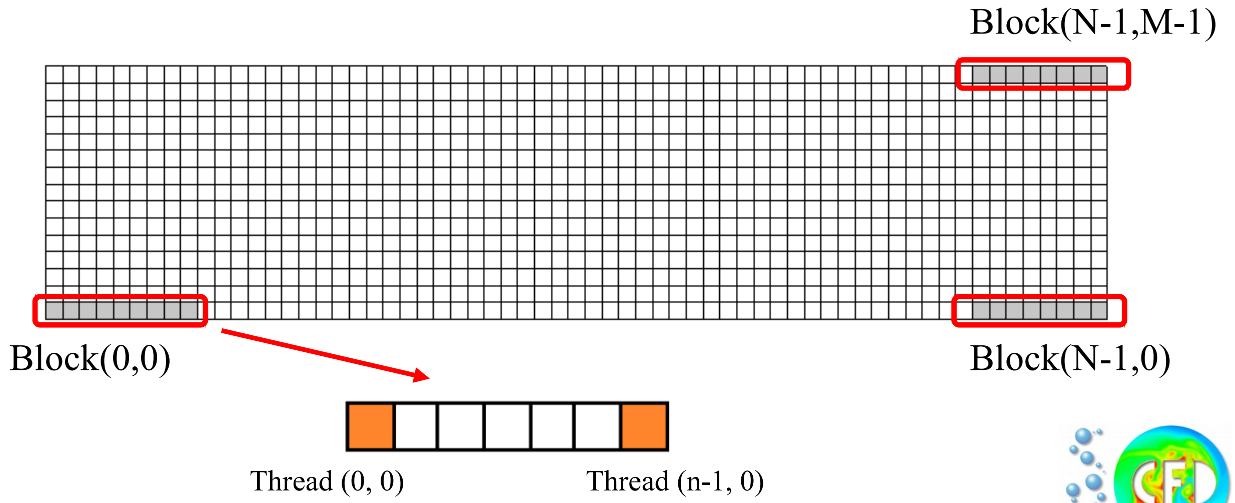
How do threads perform calculations in parallel?

- In some numerical scientific applications, each thread is in charge of one data element in your computational domain.

Your computational Grid



Your CUDA Grid



The nvcc compiler workflow

- GPU kernels are typically stored in files ending with .cu
- The rest of the code could be stored in the same .cu file or separately in other .cu, .c or .cpp files.
- `nvcc` separates the device code form the host code and:
  - Automatically handles #include's and linking libraries
  - Compiles the device code into an assembly form (**ptx** code) and/or binary form (**cubin** object).
  - Modifies the host code to replace <<<…>>> (for kernel calls) with associated CUDA-runtime directives in the ptx code.
  - Uses the host compiler(C/C++) to compile CPU code.
- Application can then
  - Either link to the compiled host code,
  - Or ignore the modified host code (if any) and use the CUDA driver API to load and execute the PTX code or cubin object.

CUDA Programming

A typical CUDA program includes:

- Explicitly managing host and device memory
  - Allocatoin of data on CPU & GPU
  - Transfers of data form CPU to GPU
- Setting the dimensions of blocks and grids.
- Launching kernels on GPU
- Copying the results back to CPU for post-processing.
- Freeing the memory on CPU & GPU.

How a kernel works?

An Element-wise Matrix Addition Code

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

CUDA Programming

How a kernel works?

An Elementwise Matrix Addition Code

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

The nested `for`-loops are replaced with an implicit grid

CUDA Programming

A rather complete example

```
const int N = 1024;
const int blocksize = 16;
```

Input data size and block size

```
__global__
void add_matrix( float* a, float *b, float *c,
int N )
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int index = i + j*N;
if ( i < N && j < N )
c[index] = a[index] + b[index];
}
```

Compute Kernel

```
float *a = new float[N*N];
float *b = new float[N*N];
float *c = new float[N*N];
```

Allocation on CPU

```
for ( int i = 0; i < N*N; ++i ) {
a[i] = 1.0f; b[i] = 3.5f; }
```

Fill arrays

```
float *ad, *bd, *cd;
const int size = N*N*sizeof(float);
cudaMalloc( (void**)&ad, size );
cudaMalloc( (void**)&bd, size );
cudaMalloc( (void**)&cd, size );
```

Allocation on GPU

```
cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
```

Copy data to GPU

```
dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
```

Call the cuda kernels

```
cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
```

Copy result back to CPU

```
cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
delete[] a; delete[] b; delete[] c;
```

Free the memory

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

CUDA Programming

technische universität dortmund

Some key notes

- The size of blocks and grids are determined in accordance to the size of problem and device memory limitations
- Kernel calls are synchronous relative to each other
- Control returns to CPU after launching a kernel (asynchronous to CPU instructions)
- Memory transfers between GPU and CPU are completely synchronous
- Memory transfers using pinned memory are asynchronous

A CFD Example

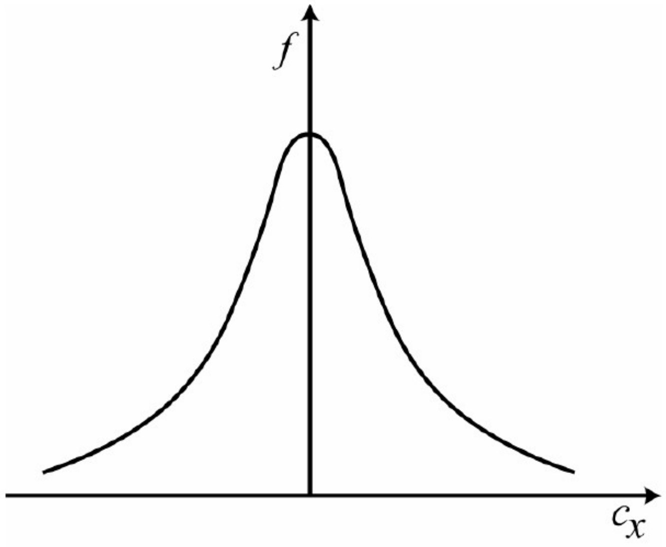
Lattice Boltzmann Simulation

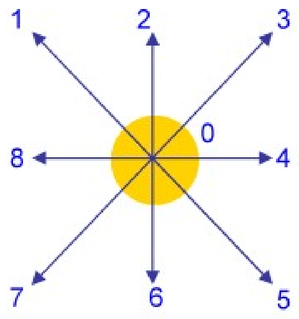
$$\frac{\partial f_i}{\partial t} + c_i.\nabla f_i = Q_i = -\frac{1}{\lambda}(f_i - f_i^{eq}(\rho, u))$$

Distribution function

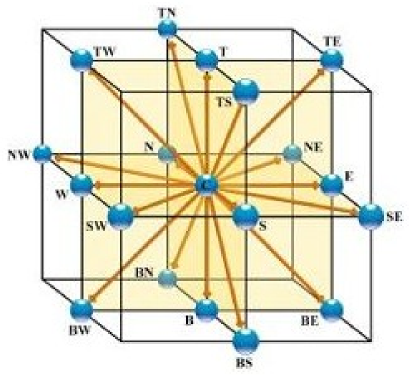
$$\rho = \sum_i f_i \quad , \quad \rho u = \sum_i c_i f_i$$

u is **macroscopic** velocity





LB Model D2Q9



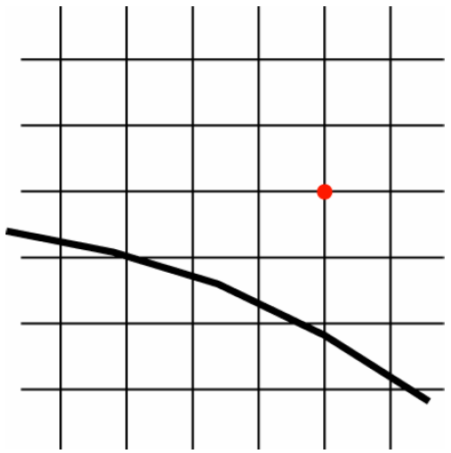
LB Model D3Q19

A CFD Example

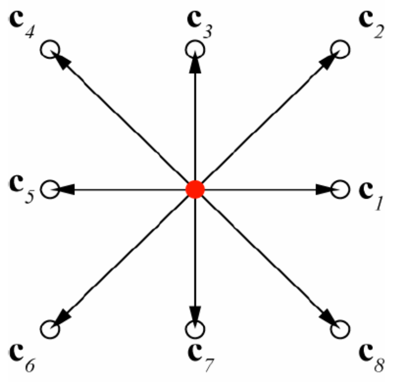
Lattice Boltzmann Simulation

- We use the D2Q9 model for 2D flow with 9 velocities
- The Navier-Stokes Eqs are recovered for incompressible, isothermal flow in hydrodynamic limit

Uniform mesh (lattice)

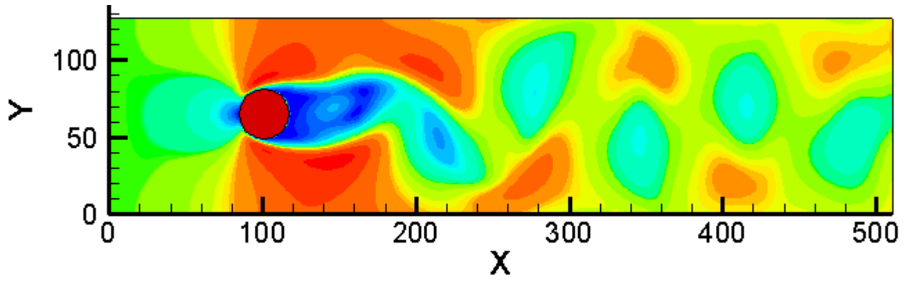


Restrict microscopic velocities to a finite set:



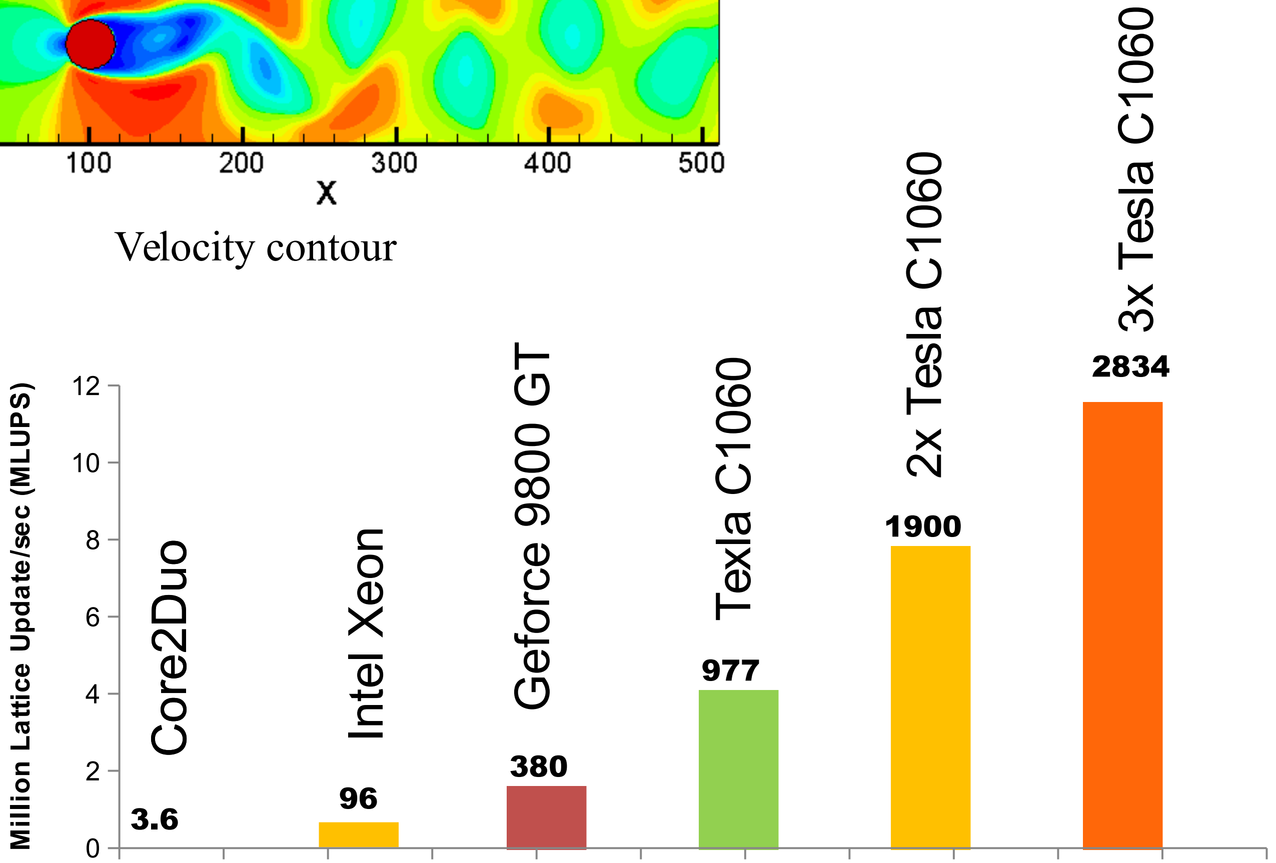
CFD Examples

- Flow past a column of cylinders (2010)



Re = 100 , 128 x 512 Grid

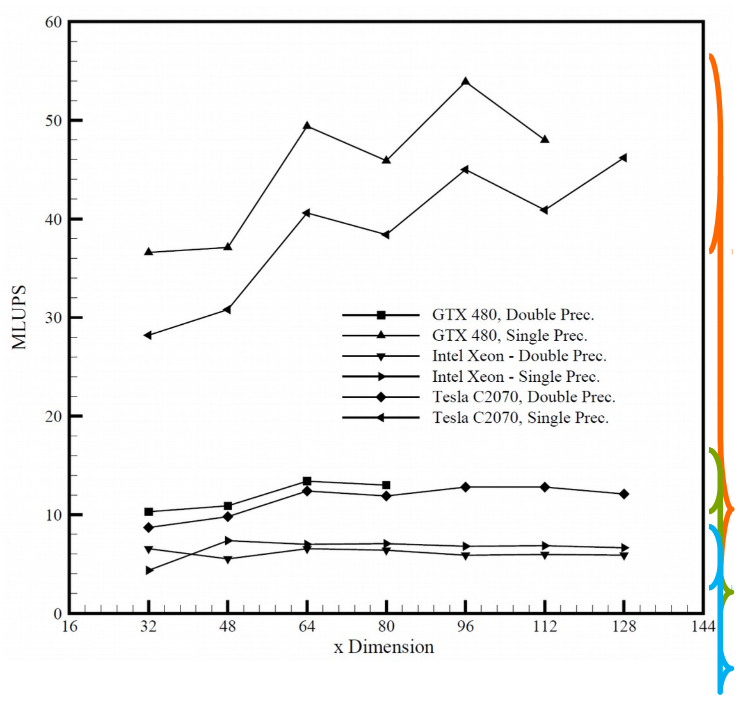
Velocity contour

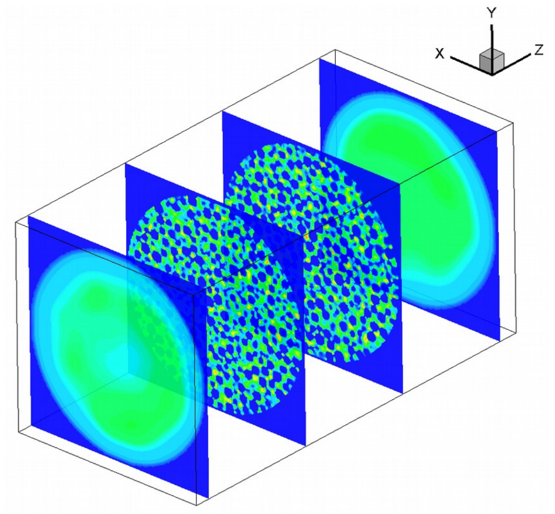


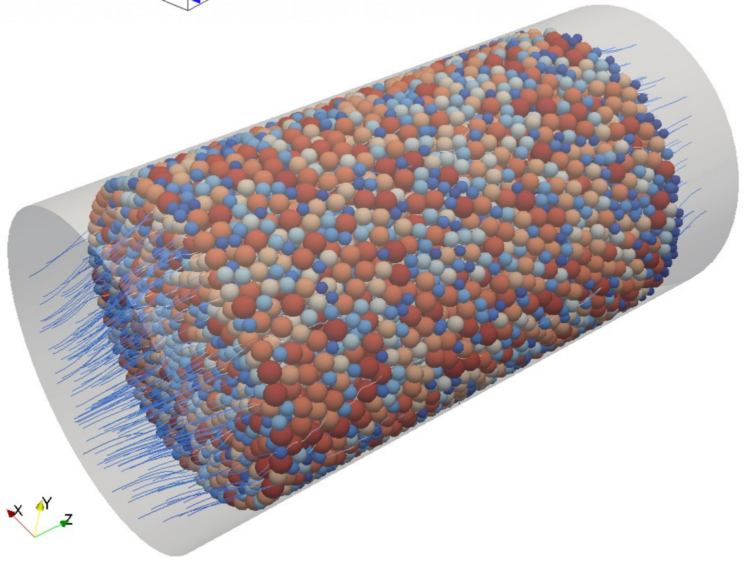
CFD Examples

- 3D Multi-component flow of O_2 and N_2 (2012)

- Air flow segregates into its ingredients
- Multicompent, Entropic LB model





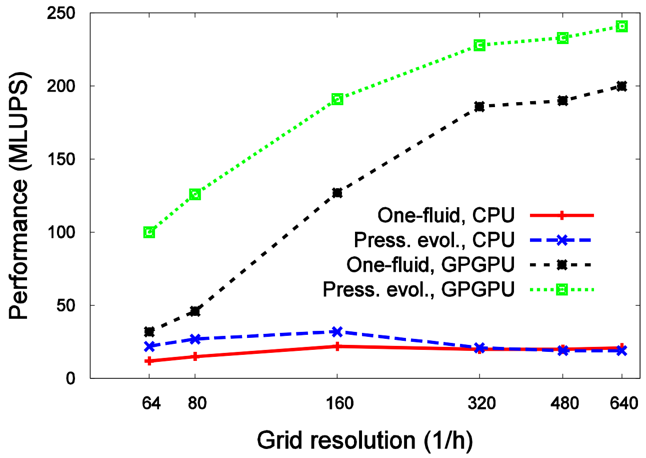


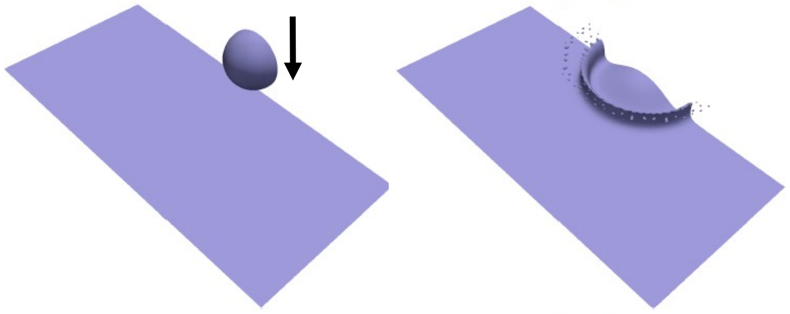
CFD Examples

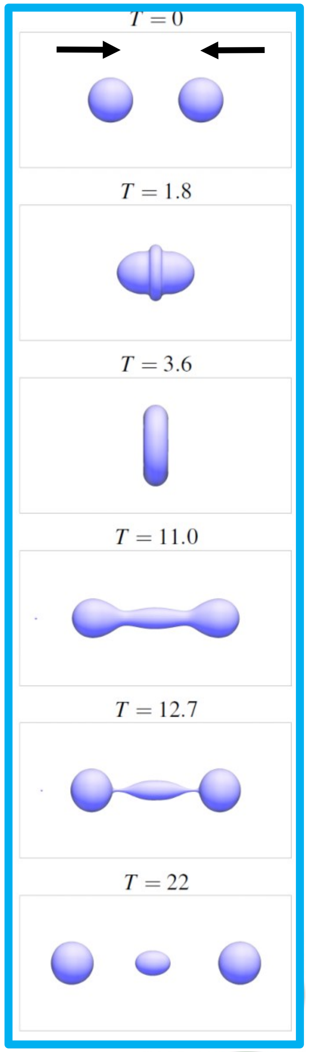
technische universität dortmund

- 2D and 3D two-phase flows (2015)

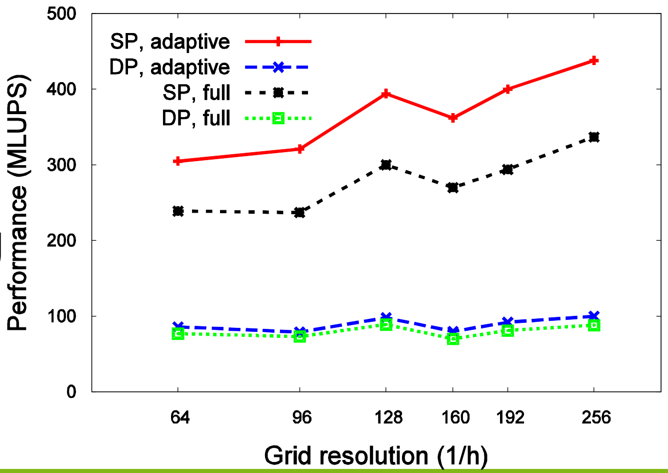
2D scaling

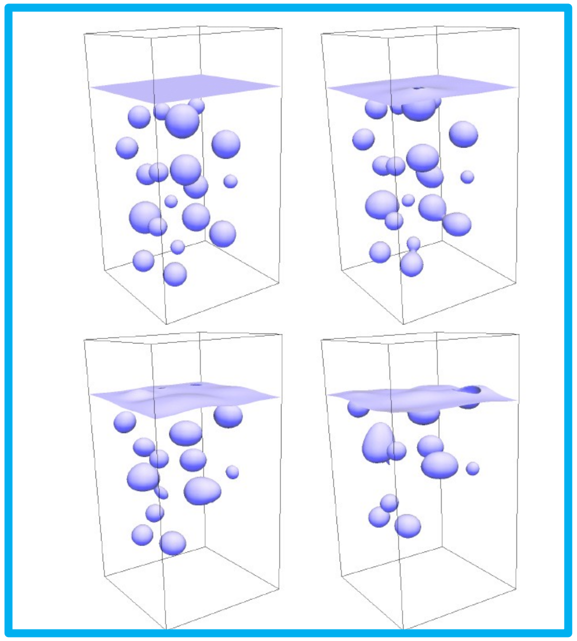






3D scaling





A CFD Example

Performance of our code

- Using different GPU generation, we achieved **10x-20x** speedup.
- Almost real-time simulations for early stage evaluations.
- SP is 3-4 times faster than DP $\Rightarrow$ use this free speedup if possible!!
- These speedups are for an *optimized* version of our code.
- *Otherwise*, the speedup would drop drastically even on a most modern GPUs.

Optimization is Vital !

Performance Optimization

- **4 Major Optimization Strategies**
  - **Memory Access optimization**
  - **Increasing Hardware Occupancy**
  - **Control Flow Optimization**
  - **Instruction Optimization**

The first two are the most important ones

Memory Access Optimization

Why so important?

- Memory transfer accounts for the majority of simulation time in memory bound applications (most large data scientific applications).
- Theoretical bandwidth between GPU DRAM and SMXs is more than 250 GB/s.
- Up to 85% of this bandwidths is achievable only and only if:

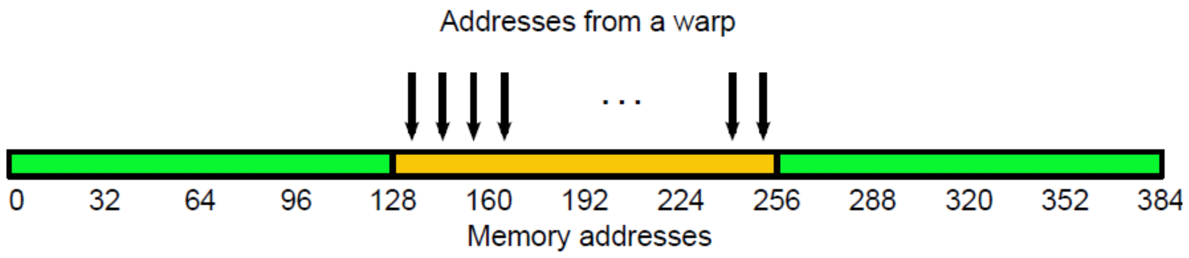
The memory accesses are **coalesced** by threads in a warp

- Otherwise, the effective bandwidths drops to 10% of max value.

Memory Access Optimization

Memory access anatomy

- Memory accesses by a warp (32 threads) are *coalesced* into as few as one transaction when certain access requirements are met



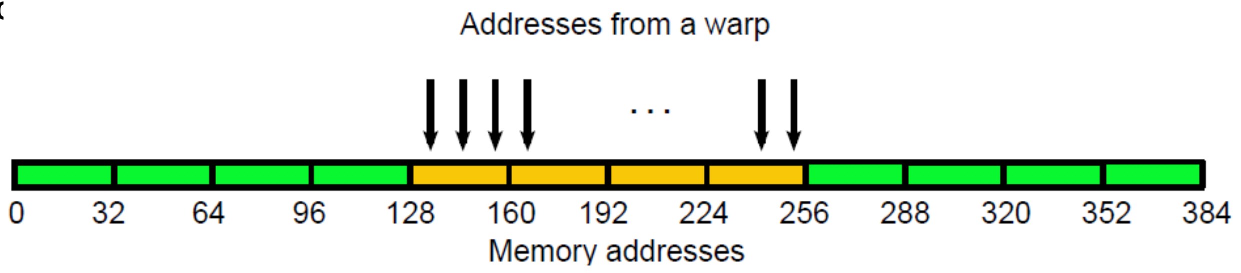
- No. of transactions = number of *cache lines* necessary to service the warp.
- Cache line size: 128 byte L1 segments in Fermi, 32 byte L2 segments in Kepler.
- 100% memory performance if all required data are found in one cache line
- Poorest performance if none of the other data items in the cache line are ever used (*cache thrashing*).
- *Keep block sizes as multiples of 32.*
- *Avoid scattered, non-local data dependencies if possible!*

Memory Access Optimization

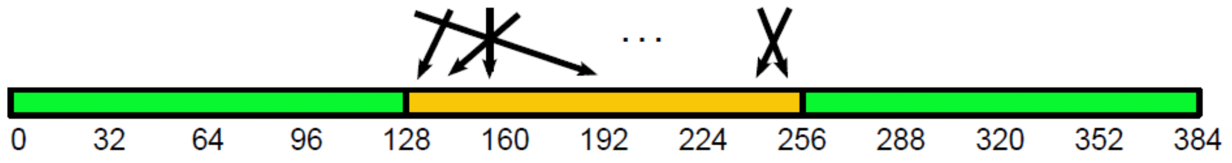
technische universität dortmund

Access pattern examples:

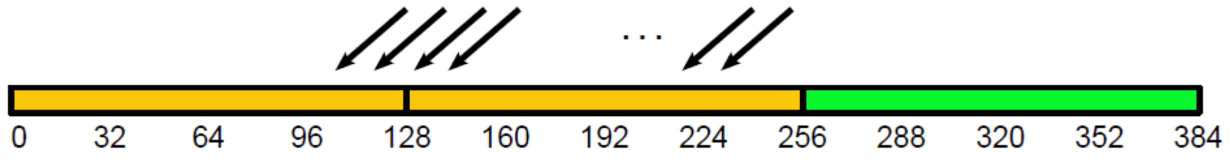
- Efficiency =100%



- Efficiency = 100%



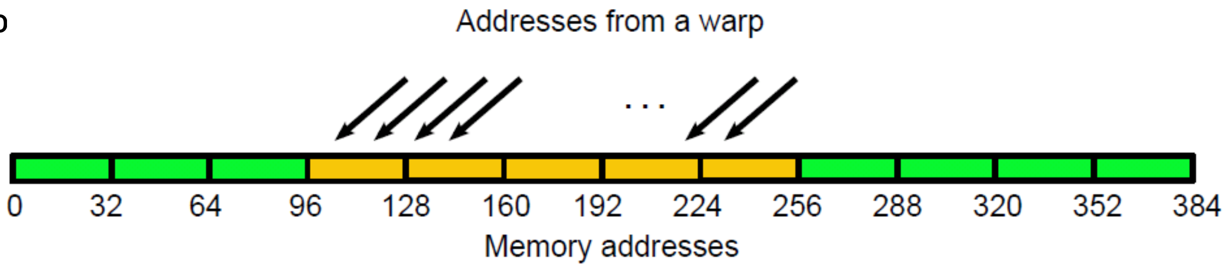
- Efficiency = 50%



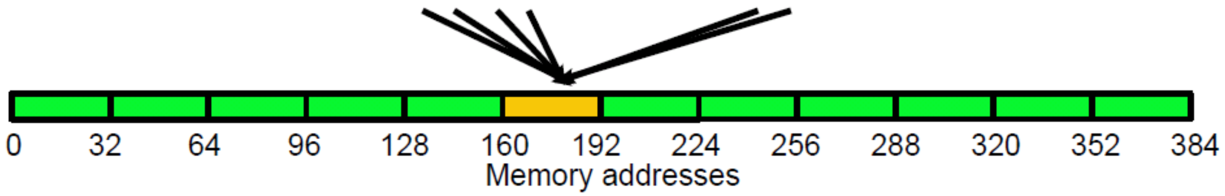
Memory Access Optimization

Access pattern examples:

- Efficiency =100%



- Efficiency = (4/32)*100=12.5%



Increasing Occupancy

Multiprocessor Occupancy

- Each MP has a limited register and shared memory
- Each MP manages a maximum of 2048 threads simultaneously
- Each thread takes up a certain number of registers and shared memory

$$\text{Occupancy} = \frac{\text{number of active threads per multiprocessor}}{\text{maximum number of possible active threads}}$$

- Care must be taken to keep the Occupancy above 25%
- A 100% occupancy does NOT mean a high performance!!!!

Increasing Occupancy

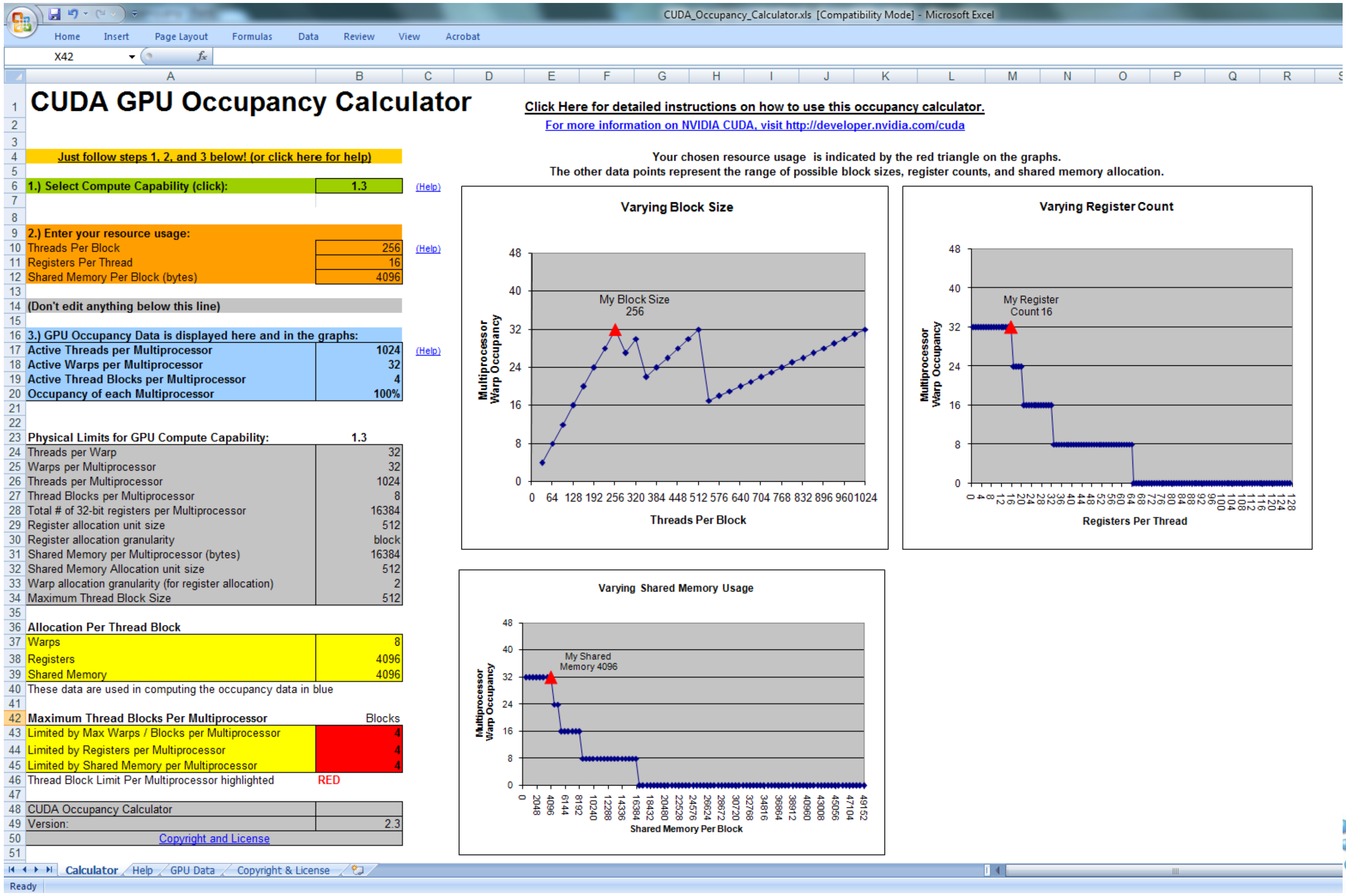
How to control Occupancy?

- Use the compiling option: `--ptxas-options=-v`
  to probe your kernels for register and shared memory consumption
- Force a maximum number of register for each thread using: `-maxrregcount=##`
- Kernel's shared memory consumption can not be forced explicitly,
  its all inaside your code
- Experiment with numbers to find a proper balance, using…..

CUDA Occupancy Calculator

Increasing Occupancy

- **Occupancy calculator**



CUDA_Occupancy_Calculator.xls [Compatibility Mode] - Microsoft Excel

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

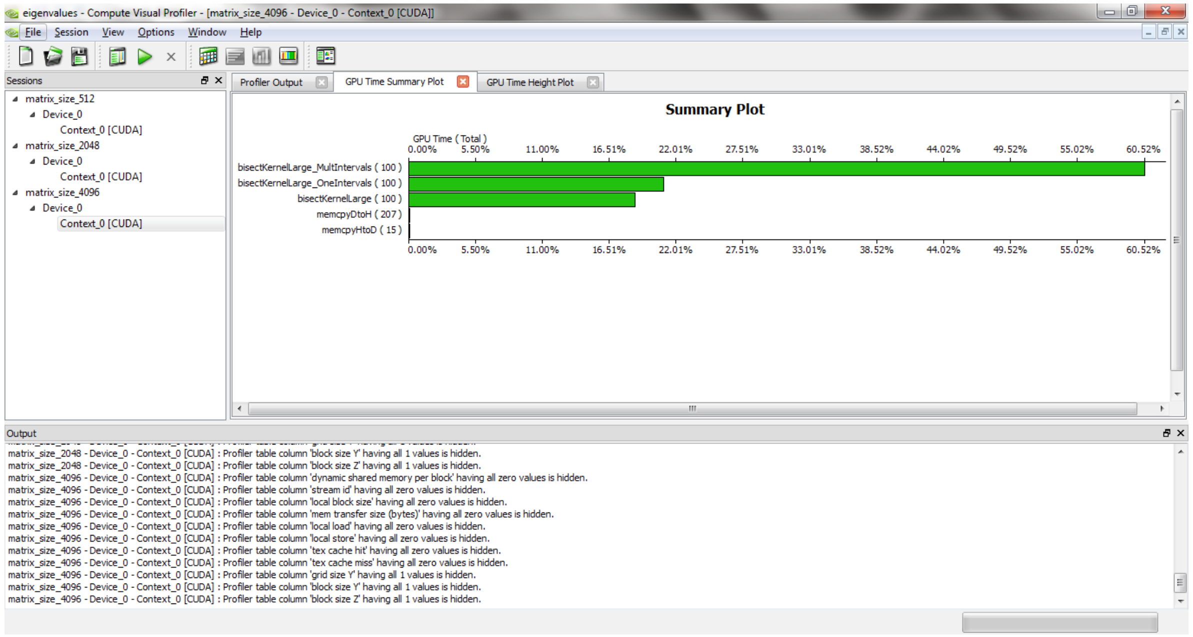
Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

	A	B	C
1	CUDA GPU Occupancy Calculator		
4	Just follow steps 1, 2, and 3 below! (or click here for help)		
6	1.) Select Compute Capability (click):	1.3	(Help)
9	2.) Enter your resource usage:		
10	Threads Per Block	256	(Help)
11	Registers Per Thread	16	
12	Shared Memory Per Block (bytes)	4096	
14	(Don't edit anything below this line)		
16	3.) GPU Occupancy Data is displayed here and in the graphs:		
17	Active Threads per Multiprocessor	1024	(Help)
18	Active Warps per Multiprocessor	32	
19	Active Thread Blocks per Multiprocessor	4	
20	Occupancy of each Multiprocessor	100%	
23	Physical Limits for GPU Compute Capability:	1.3	
24	Threads per Warp	32	
25	Warps per Multiprocessor	32	
26	Threads per Multiprocessor	1024	
27	Thread Blocks per Multiprocessor	8	
28	Total # of 32-bit registers per Multiprocessor	16384	
29	Register allocation unit size	512	
30	Register allocation granularity	block	
31	Shared Memory per Multiprocessor (bytes)	16384	
32	Shared Memory Allocation unit size	512	
33	Warp allocation granularity (for register allocation)	2	
34	Maximum Thread Block Size	512	
36	Allocation Per Thread Block		
37	Warps	8	
38	Registers	4096	
39	Shared Memory	4096	
40	These data are used in computing the occupancy data in blue		
42	Maximum Thread Blocks Per Multiprocessor	Blocks	
43	Limited by Max Warps / Blocks per Multiprocessor	4	
44	Limited by Registers per Multiprocessor	4	
45	Limited by Shared Memory per Multiprocessor	4	
46	Thread Block Limit Per Multiprocessor highlighted	RED	
48	CUDA Occupancy Calculator		
49	Version:	2.3	
50	Copyright and License		

Performance Optimization

Profiling and final check

- Always profile your kernels to evaluate:
  - Memory access quality
  - kernel Occupancy
  - Each kernel's contribution to the total time
- Use the Compute Visual Profiler to check these



```
matrix_size_2048 - Device_0 - Context_0 [CUDA] : Profiler table column 'block size Y' having all 1 values is hidden.
matrix_size_2048 - Device_0 - Context_0 [CUDA] : Profiler table column 'block size Z' having all 1 values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'dynamic shared memory per block' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'stream id' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'local block size' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'mem transfer size (bytes)' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'local load' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'local store' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'tex cache hit' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'tex cache miss' having all zero values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'grid size Y' having all 1 values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'block size Y' having all 1 values is hidden.
matrix_size_4096 - Device_0 - Context_0 [CUDA] : Profiler table column 'block size Z' having all 1 values is hidden.
```

Thank You

Programming On GPUs

CUDA Technology

- Introduced to market by nVIDIA in 2006
- An Integrated computational architecture to exploit all the computational resource of GPUs.
- Comes with a compiler based on C and other scientific languages.
- Enables computing on low price, small GPUs ⇨ **Personal Super Computer**