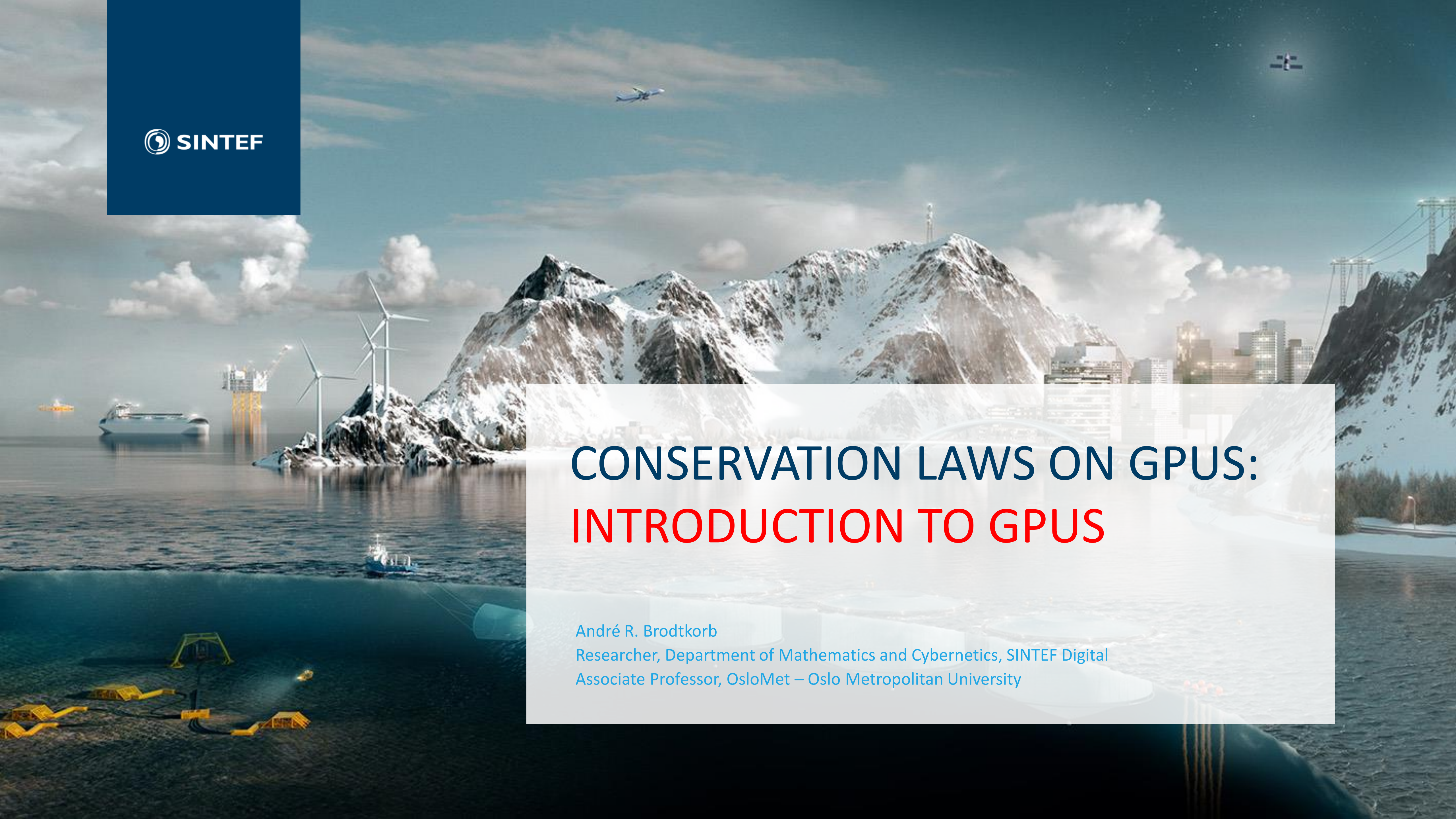


CONSERVATION LAWS ON GPUS: INTRODUCTION TO GPUS

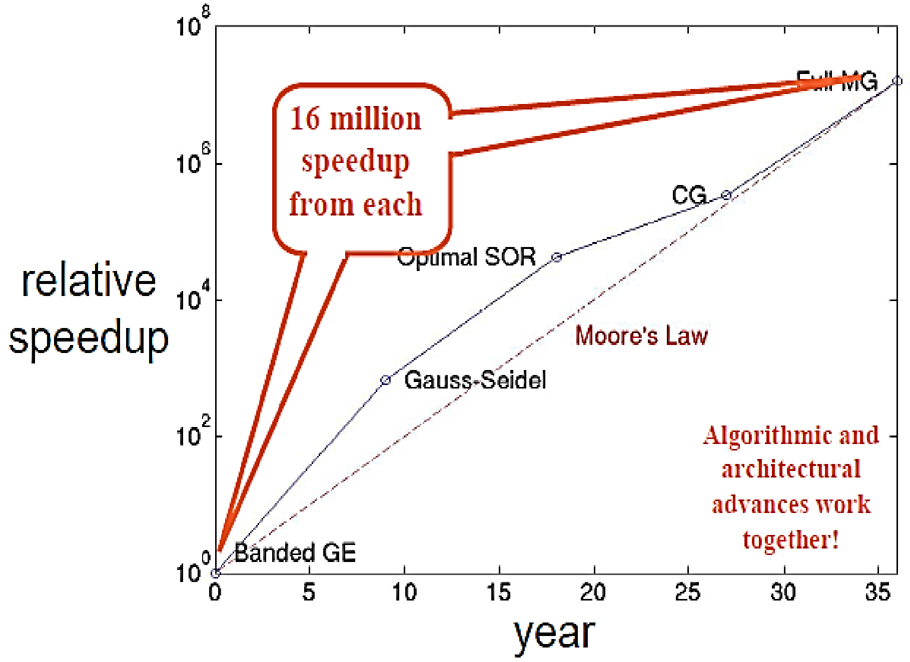
André R. Brodtkorb
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital
Associate Professor, OsloMet – Oslo Metropolitan University

Motivation for going parallel

Why care about computer hardware?

- The key to increasing performance, is to consider the full algorithm and architecture interaction.

- A good knowledge of <u>both</u> the algorithm <u>and</u> the computer architecture is required.



Graph from David Keyes, Scientific Discovery through Advanced Computing, Geilo Winter School, 2008

History lesson: development of the microprocessor 1/2

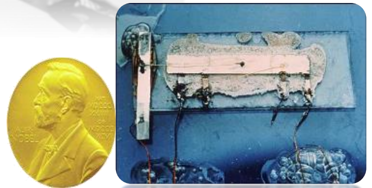
1942: Digital Electric Computer

(Atanasoff and Berry)

1956

1947: Transistor

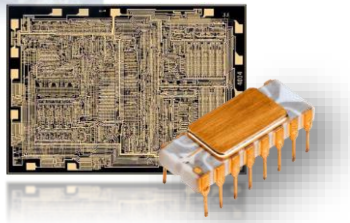
(Shockley, Bardeen, and Brattain)



2000

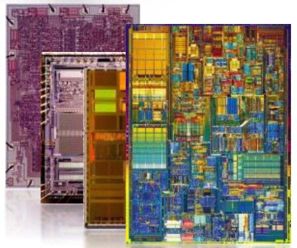
1958: Integrated Circuit

(Kilby)



1971: Microprocessor

(Hoff, Faggin, Mazor)



1971- Exponential growth

(Moore, 1965)

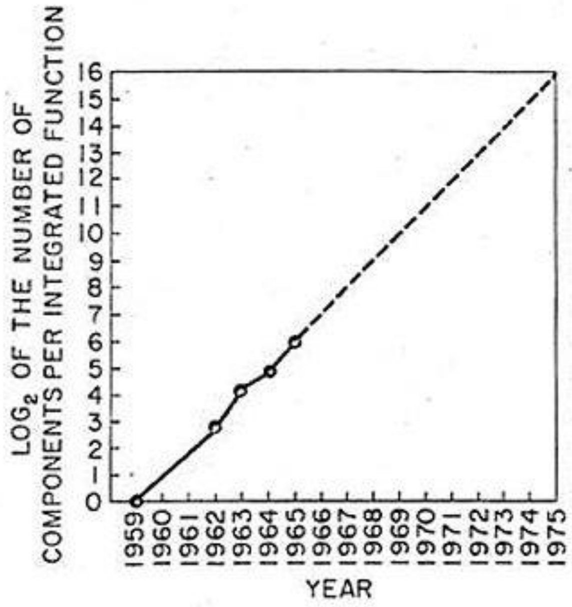
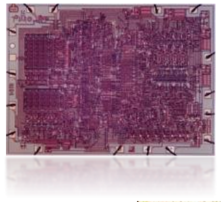


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

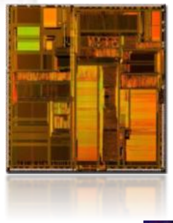
SINTEF

History lesson: development of the microprocessor 2/2

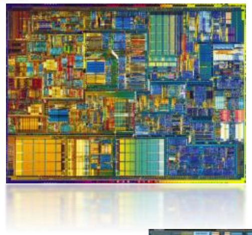


1971: 4004,
2300 trans, 740 KHz

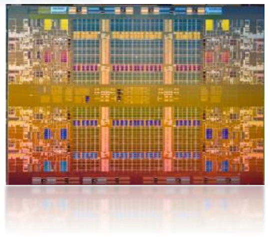
1982: 80286,
134 thousand trans, 8 MHz



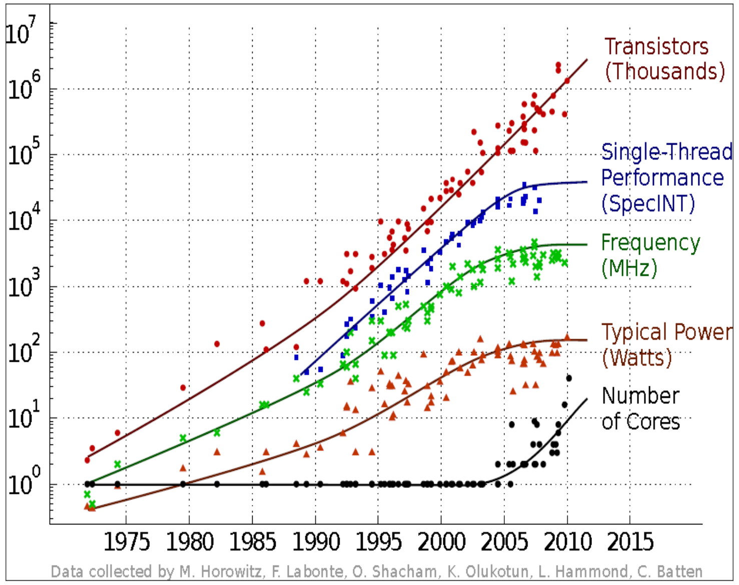
1993: Pentium P5,
1.18 mill. trans, 66 MHz



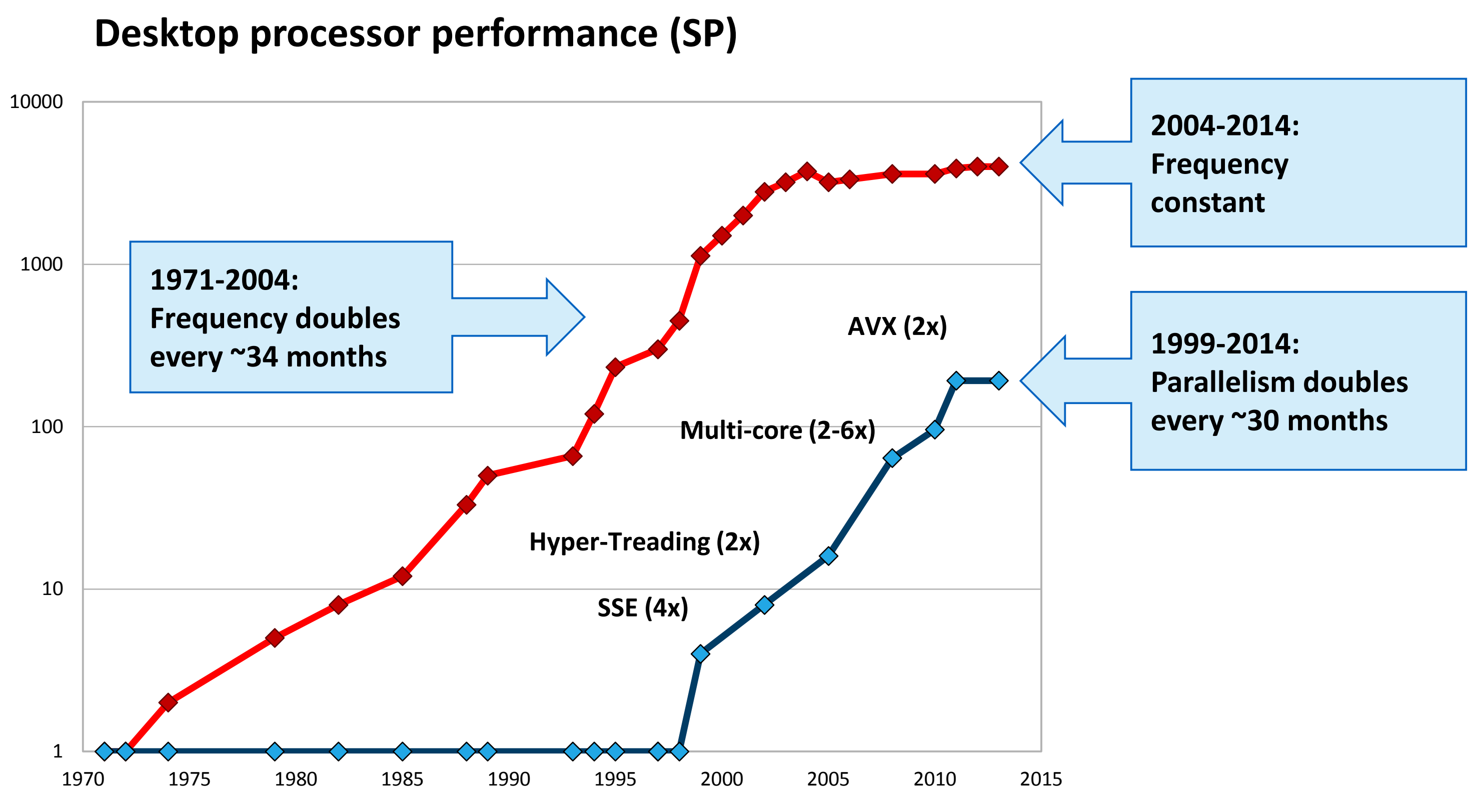
2000: Pentium 4,
42 mill. trans, 1.5 GHz



2010: Nehalem
2.3 bill. Trans, 8 cores, 2.66 GHz



End of frequency scaling



- 1970-2004: Frequency doubles every 34 months (Moore’s law for performance)
- 1999-2014: Parallelism doubles every 30 months

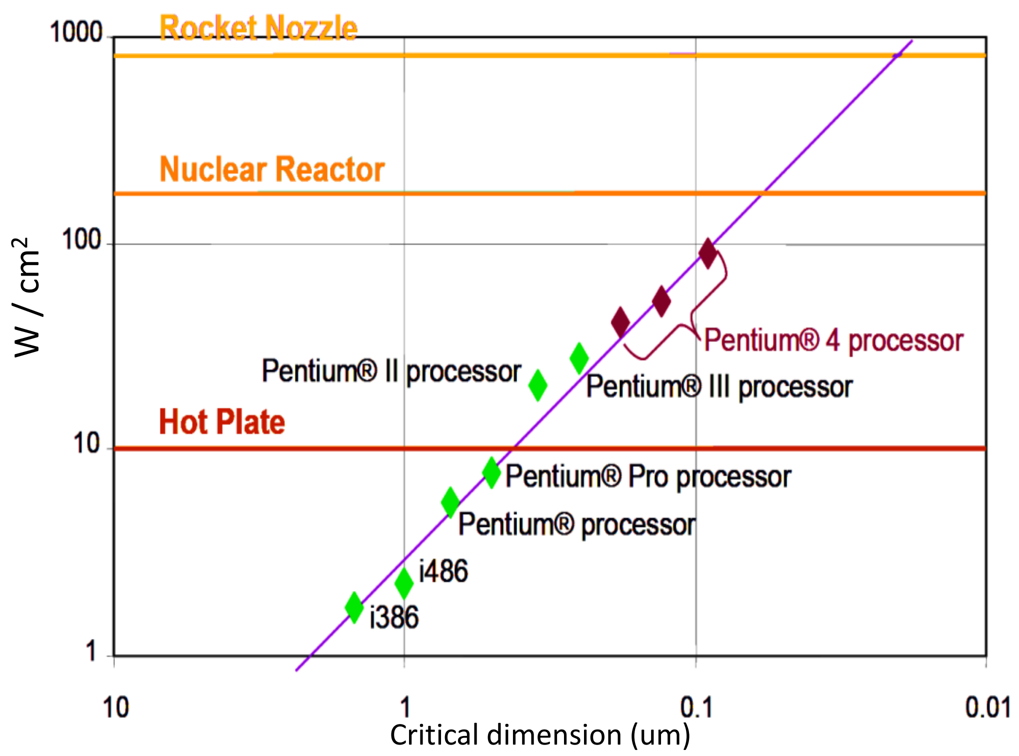
SINTEF

What happened in 2004?

- Heat density approaching that of nuclear reactor core: Power wall

- Traditional cooling solutions (heat sink + fan) insufficient

- Industry solution: multi-core and parallelism!

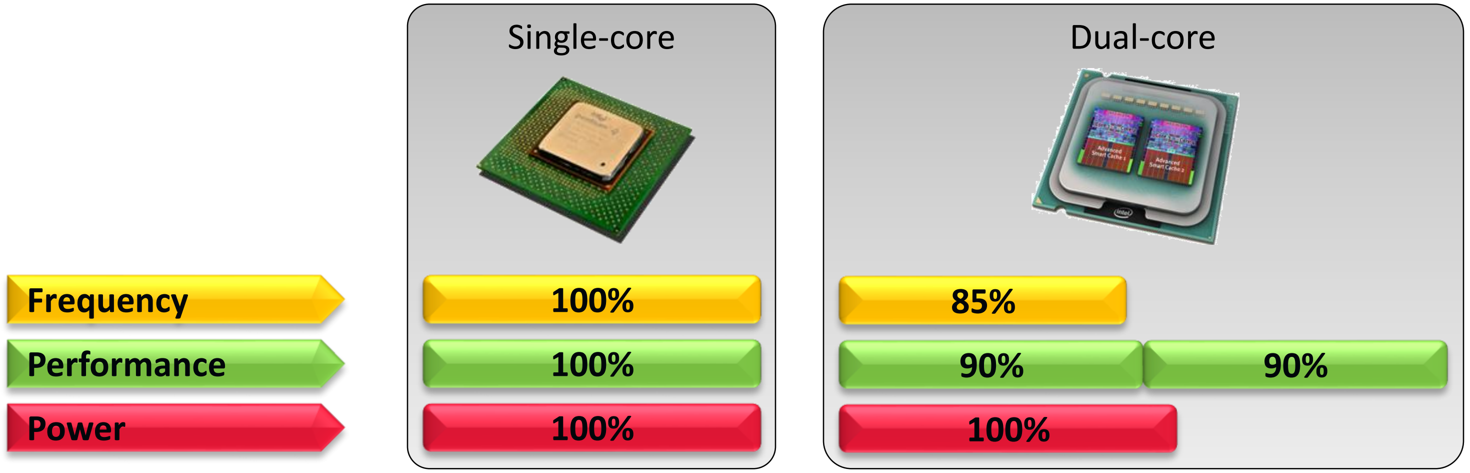


Graph taken from G. Taylor, “Energy Efficient Circuit Design and the Future of Power Delivery” EPEPS’09

Why Parallelism?

The power density of microprocessors
is proportional to the clock frequency cubed:[1]

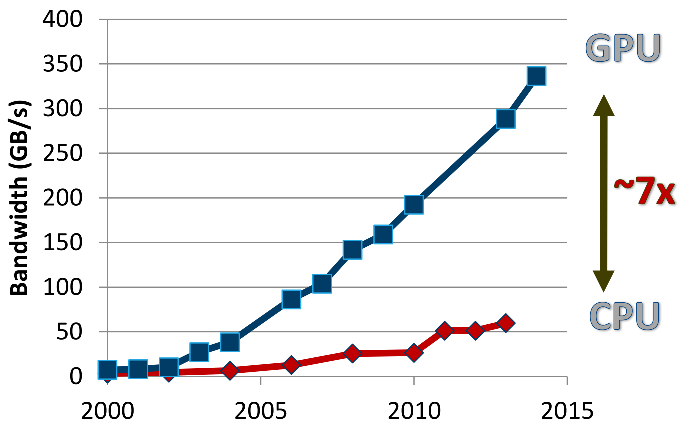
$$P_d \propto f^3$$

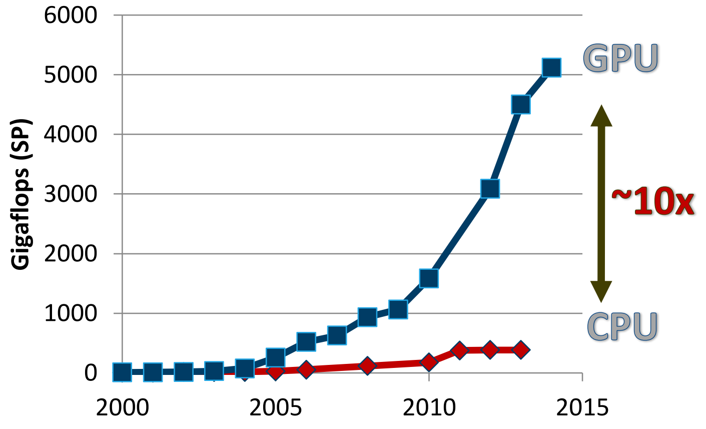


[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

Massive Parallelism: The Graphics Processing Unit

- Up-to 5760 floating point operations in parallel!
- 5-10 times as power efficient as CPUs!





Multi- and many-core processors

A taxonomy of parallel architectures

- A taxonomy of different parallelism is useful for discussing parallel architectures
  - 1966 paper by M. J. Flynn: Some Computer Organizations and Their Effectiveness
  - Each class has its own benefits and uses

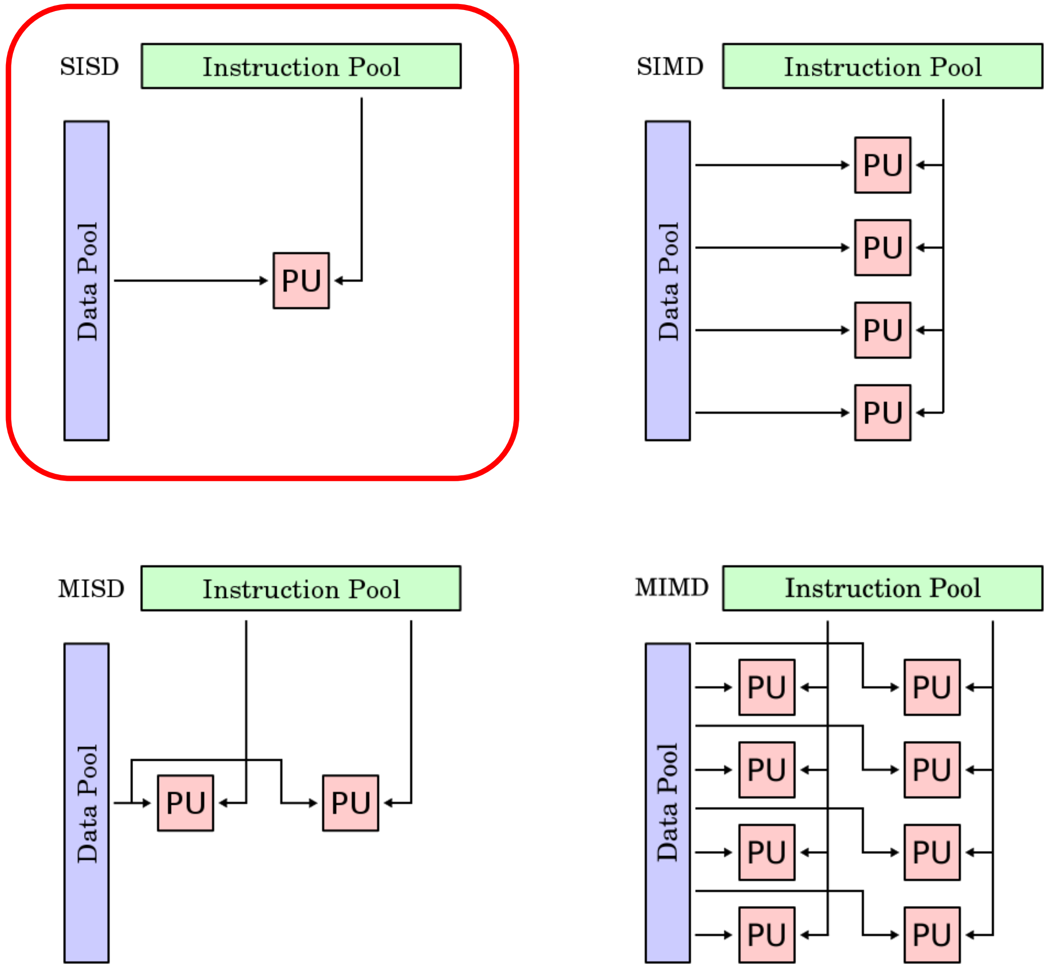
	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

M. J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., 1966

Single instruction, single data

- Traditional serial mindset:
  - Each instruction is executed after the other
  - One instruction operates on a single element
  - The typical way we write C / C++ computer programs

- Example:
  - c = a + b

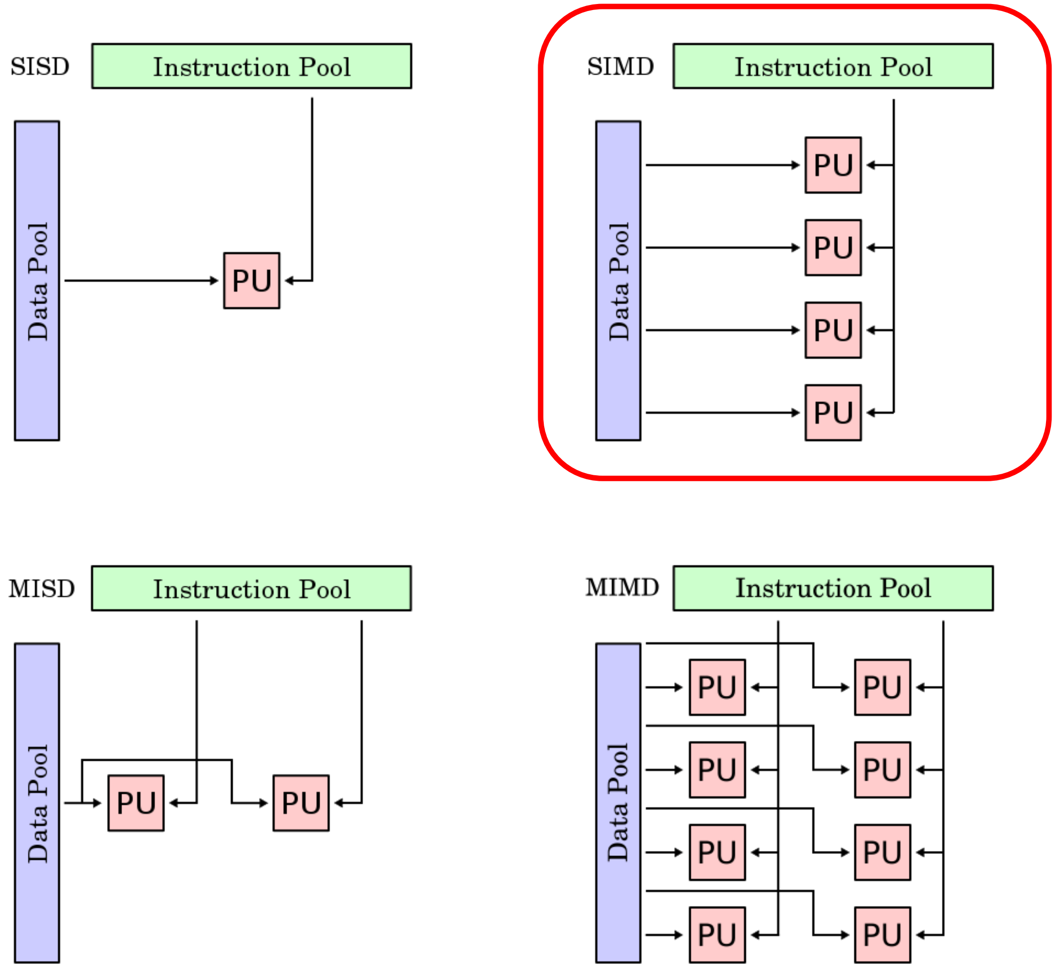


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Single instruction, multiple data

- Traditional vector mindset:
  - Each instruction is executed after the other
  - Each instruction operates on multiple data elements simultaneously
  - The way vectorized MATLAB programs often are written

- Example:
  - c[i] = a[i] + b[i]   i=0…N
  - a, b, and c are vectors of fixed length (typically 2, 4, 8, 16, or 32)

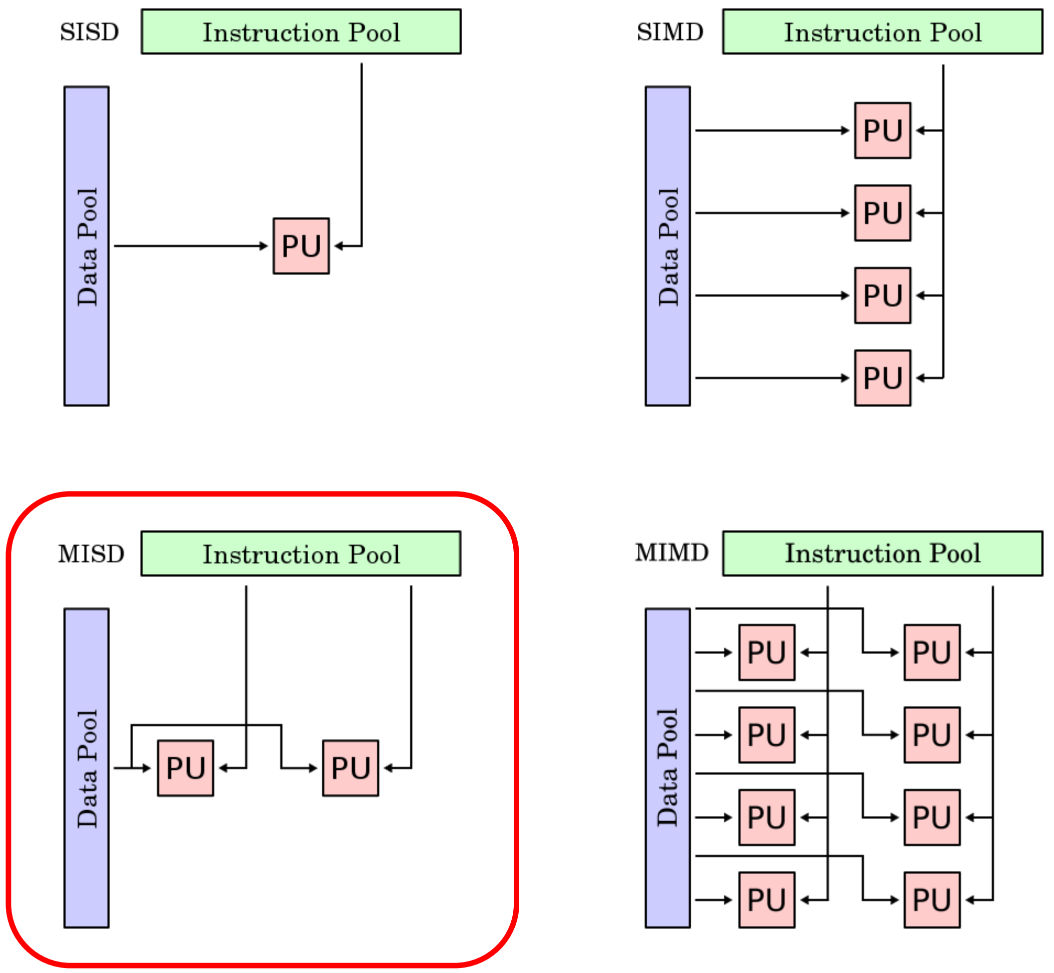


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Multiple instruction, single data

- Only for special cases:
  - Multiple instructions are executed simultaneously
  - Each instruction operates on a single data element
  - Used e.g., for fault tolerance, or pipelined algorithms implemented on FPGAs

- Example (naive detection of catastrophic cancellation):
  - PU1: z1 = x*x – y*y
    PU2: z2 = (x-y) * (x+y)
    if (z1 – z2 > eps) { ... }

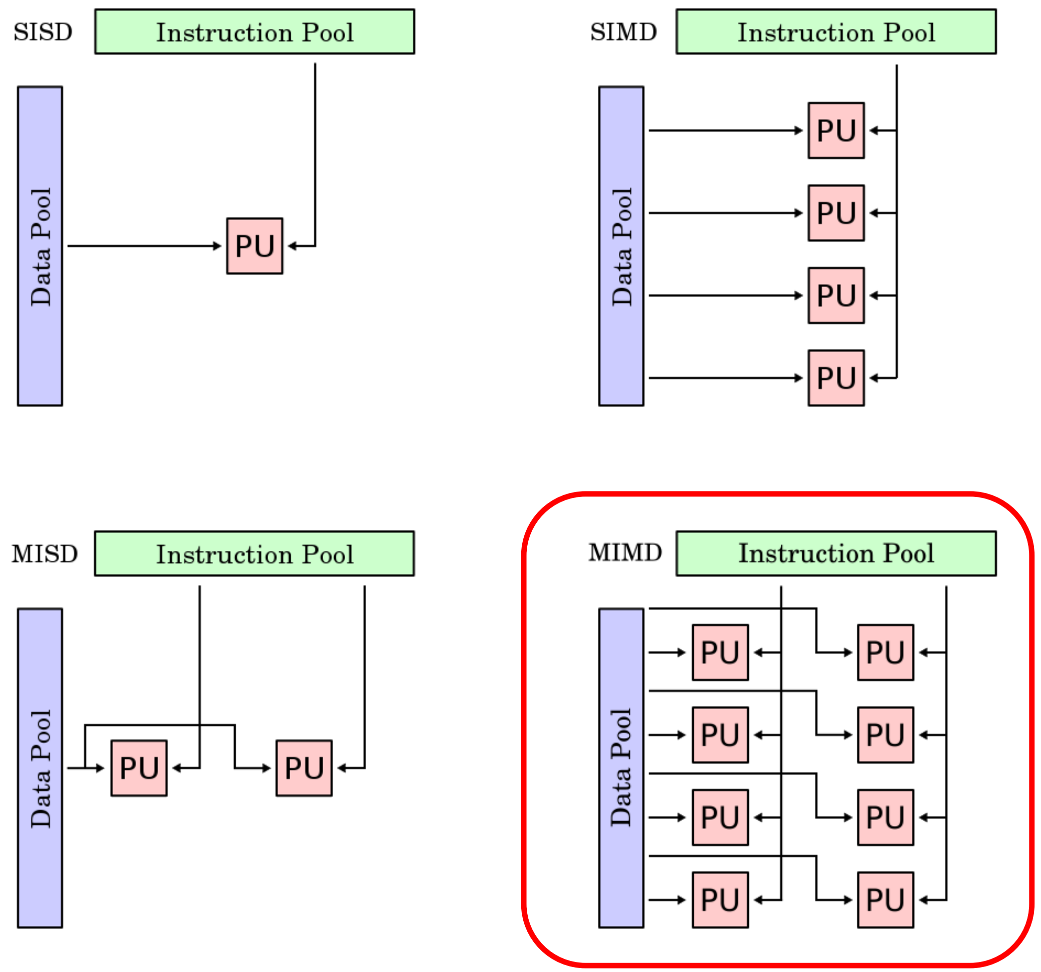


Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

Multiple instruction, multiple data

- Traditional cluster computer
  - Multiple instructions are executed simultaneously
  - Each instruction operates on multiple data elements simultaneously
  - Typical execution pattern used in task-parallel computing

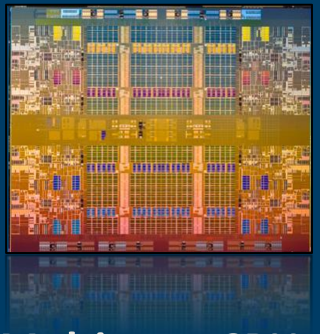
- Example:
  - PU1: c = a + b
    PU2: z = (x-y) * (x+y)
    variables can also vectors of fixed length (se SIMD)



Images from Wikipedia, user Cburnett, CC-BY-SA 3.0

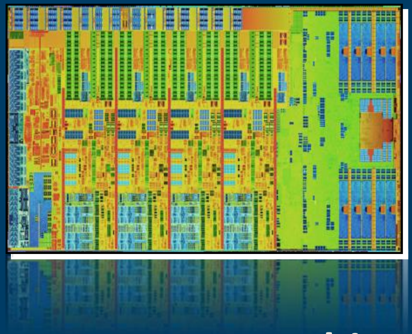
Multi- and many-core processor designs

- Today, we have
  - 6-60 processors per chip
  - 8 to 32-wide SIMD instructions
  - Combines both SISD, SIMD, and MIMD on a single chip
  - Heterogeneous cores (e.g., CPU+GPU on single chip)



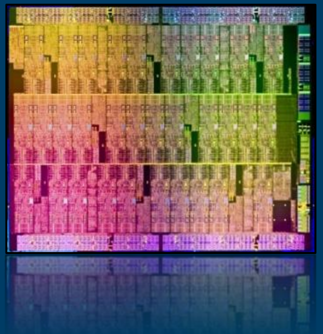
Multi-core CPUs:
x86, SPARC, Power 7

→



Heterogeneous chips:
Intel Haswell, AMD APU

←

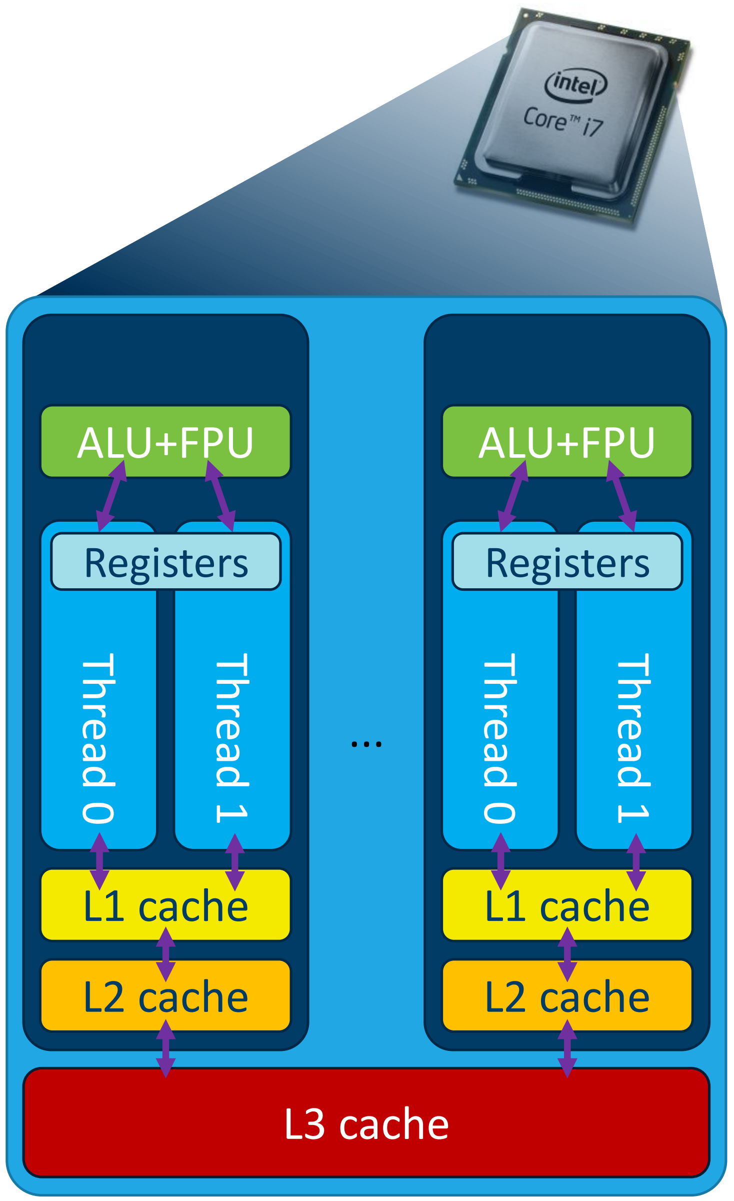


Accelerators:
GPUs, Xeon Phi

Multi-core CPU architecture

- A single core
  - L1 and L2 caches
  - 8-wide SIMD units (AVX, single precision)
  - 2-way Hyper-threading (hardware threads)
    When thread 0 is waiting for data,
    thread 1 is given access to SIMD units
  - Most transistors used for cache and logic

- Optimal number of FLOPS per clock cycle:
  - 8x: 8-way SIMD
  - 6x: 6 cores
  - 2x: Dual issue (fused mul-add / two ports)
  - Sum: 96!

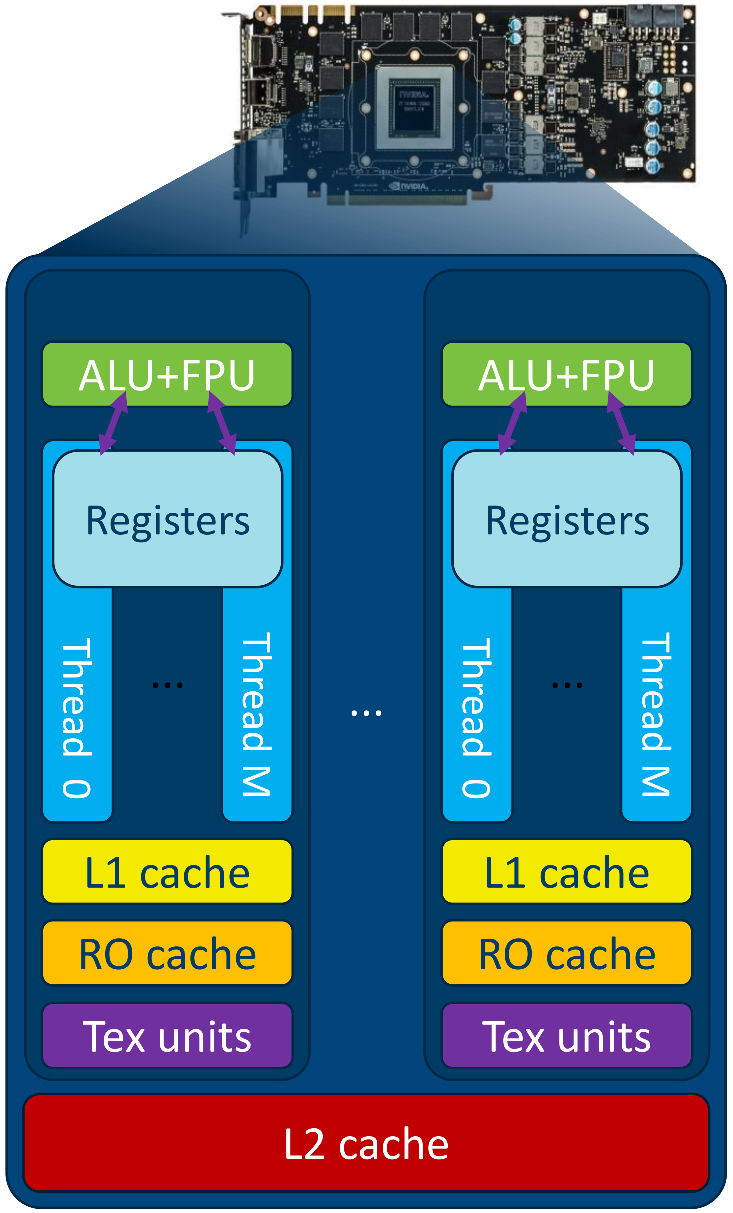


Simplified schematic of CPU design

Many-core GPU architecture

- A single core (Called streaming multiprocessor, SMX)
  - L1 cache, Read only cache, texture units
  - Six 32-wide SIMD units (192 total, single precision)
  - Up-to 64 warps simultaneously (hardware warps)
    Like hyper-threading, but a warp is 32-wide SIMD
  - Most transistors used for floating point operations

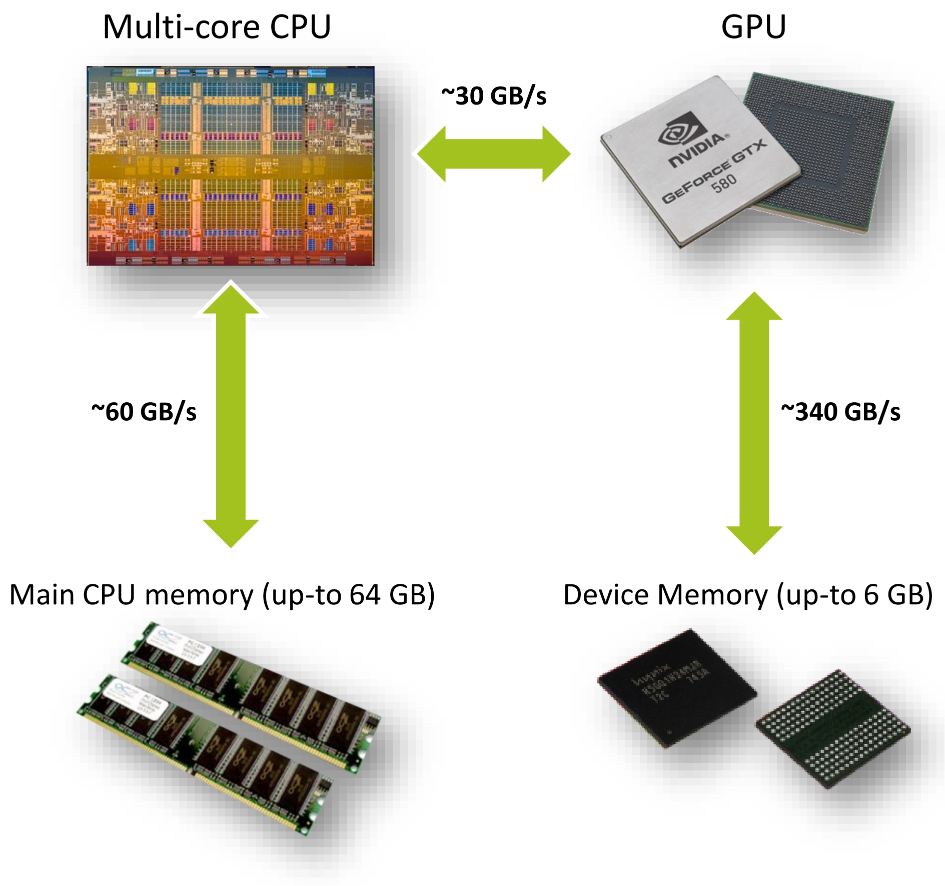
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: Six SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!



Simplified schematic of GPU design

Heterogeneous Architectures

- Discrete GPUs are connected to the CPU via the PCI-express bus
  - Slow: 15.75 GB/s each direction
  - On-chip GPUs use main memory as graphics memory
- Device memory is limited but fast
  - Typically up-to 6 GB
  - Up-to 340 GB/s!
  - Fixed size, and cannot be expanded with new dimm's (like CPUs)

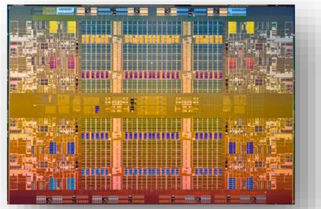


Parallel algorithm design

Type of parallel processing

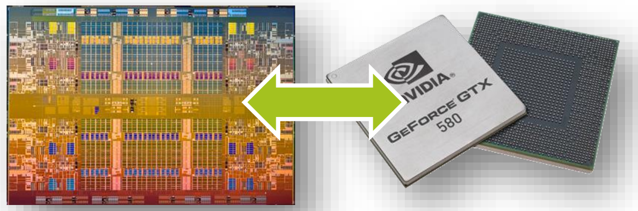
- When the processors are symmetric (identical), we tend to use symmetric multiprocessing.
  - Tasks will take the same amount of time independent of which processor it runs on.
  - All procesors can see everything in memory

Multi-core CPU



- If we have different processors, we revert to heterogeneous computing.
  - Tasks will take a different amount of time on different processors
  - Not all tasks can run on all processors.
  - Each processor sees only part of the memory

Multi-core CPU GPU



- We can even mix the two above, add message passing, etc.!

SINTEF

Mapping an algorithm to a parallel architecture

- Most algorithms are like baking recipies, Tailored for a single person / processor:
  - First, do A,
  - Then do B,
  - Continue with C,
  - And finally complete by doing D.

- How can we utilize an "army of identical chefs"?

- How can we utilize an "army of different chefs"?

Picture: Daily Mail Reporter , www.dailymail.co.uk

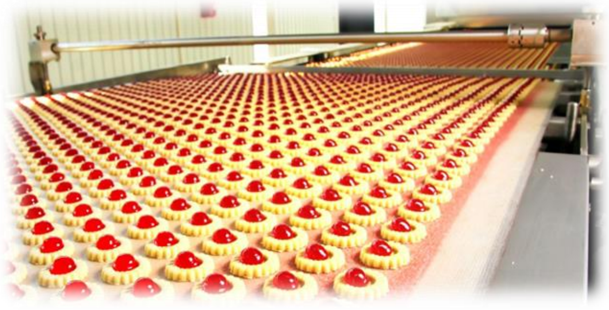
SINTEF

Data parallel workloads

- Data parallelism performs the same operation
  for a set of different input data

- Scales well with the data size:
  The larger the problem, the more processors you can utilize

- Trivial example:
  Element-wise multiplication of two vectors:
  - c[i] = a[i] * b[i]   i=0…N
  - Processor i multiplies elements i of vectors a and b.

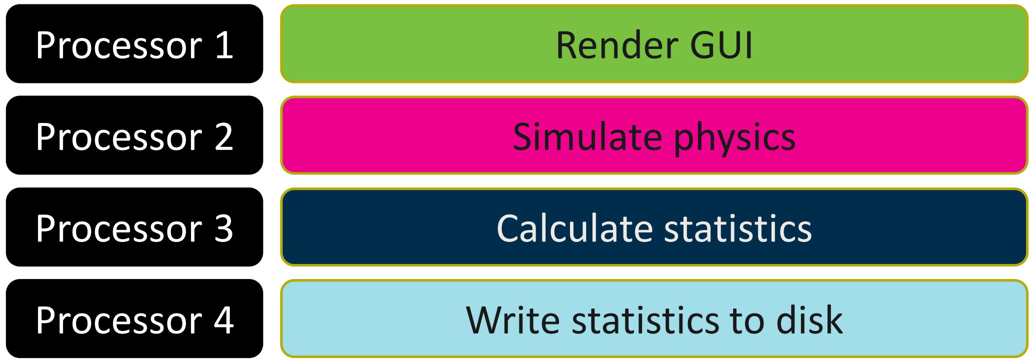


Task parallel workloads 1/3

- Task parallelism divides a problem into subtasks which can be solved individually

- Scales well for a large number of tasks:
  The more parallel tasks, the more processors you can use

- Example: A simulation application:



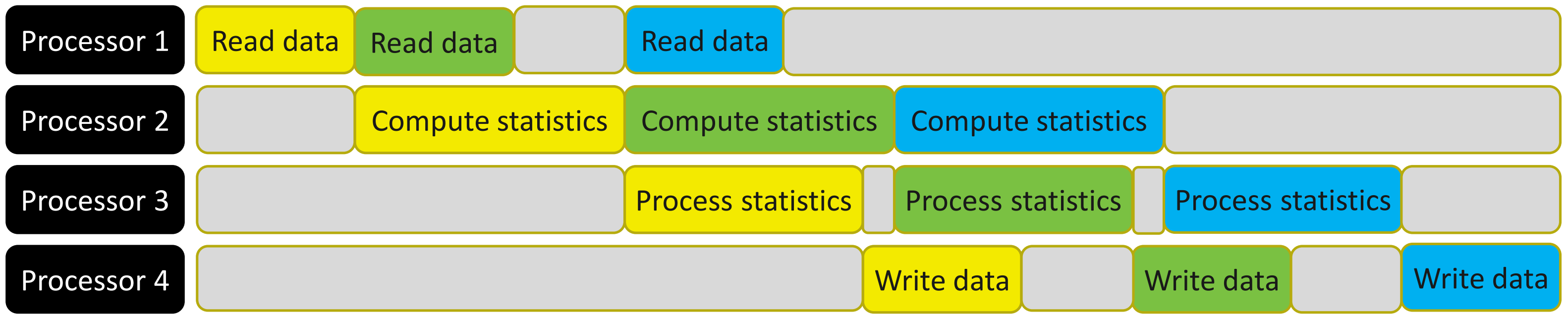
- Note that not all tasks will be able to fully utilize the processor

Task parallel workloads 2/3

- Another way of using task parallelism is
  to execute dependent tasks on different processors

- Scales well with a large number of tasks, but performance limited by slowest stage

- Example: Pipelining dependent operations



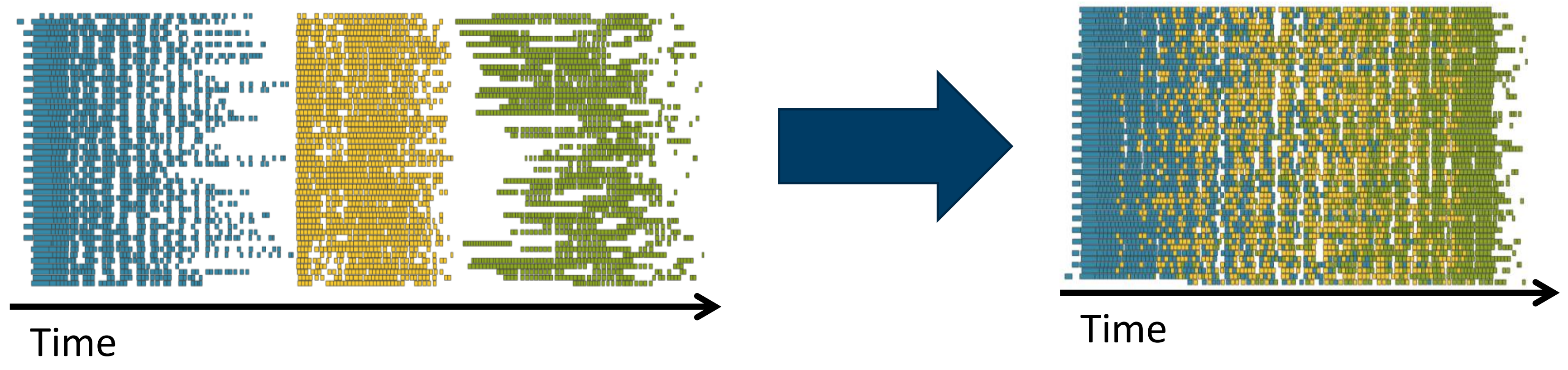
- Note that the gray boxes represent idling: wasted clock cycles!

Task parallel workloads 3/3

- A third way of using task parallelism is
  to represent tasks in a directed acyclic graph (DAG)

- Scales well for millions of tasks, as long as the overhead of executing each task is low

- Example: Cholesky inversion



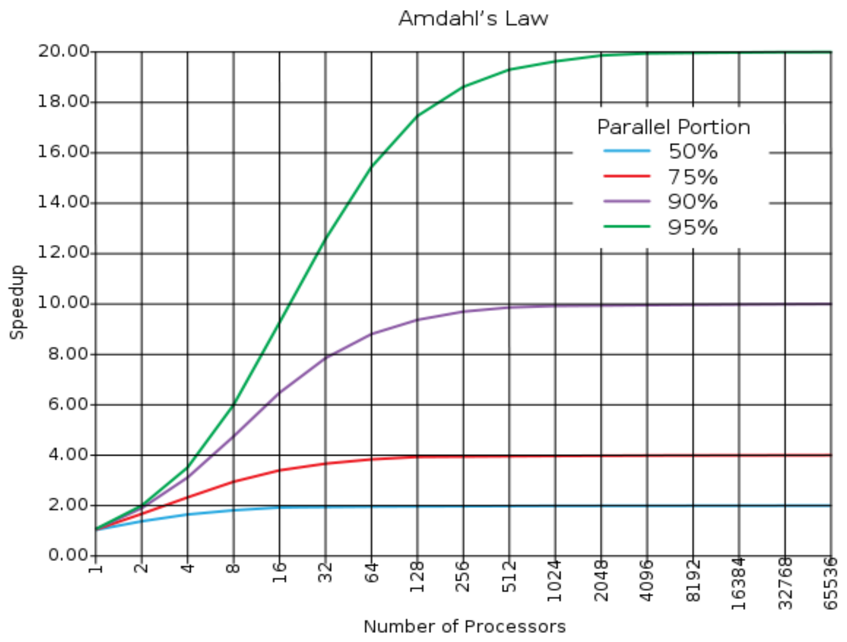
- “Gray boxes” are minimized

Example from Dongarra, On the Future of High Performance Computing: How to Think for Peta and Exascale Computing, 2012

Limits on performance 1/4

- Most algorithms contains a mixture of work-loads:
  - Some serial parts
  - Some task and / or data parallel parts

- Amdahl's law:
  - There is a limit to speedup offered by parallelism
  - Serial parts become the bottleneck for a massively parallel architecture!
  - Example: 5% of code is serial: maximum speedup is 20 times!



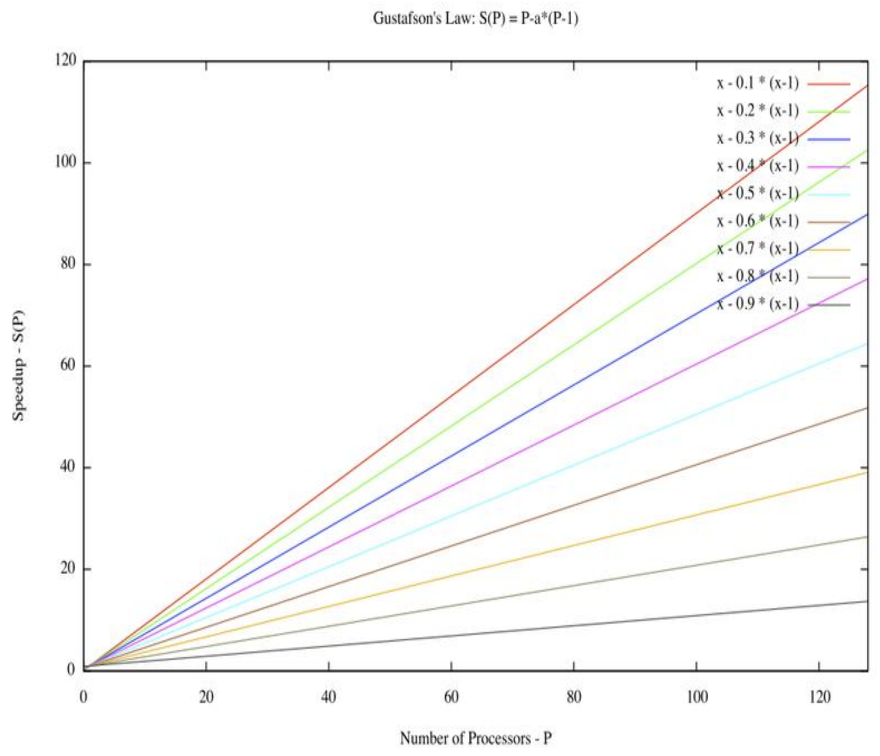
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

S: Speedup
P: Parallel portion of code
N: Number of processors

Graph from Wikipedia, user Daniels220, CC-BY-SA 3.0

Limits on performance 2/4

- Gustafson's law:
  - If you cannot reduce serial parts of algorithm, make the parallel portion dominate the execution time
  - Essentially: solve a bigger problem!



$$S(P) = P - \alpha \cdot (P - 1).$$

S: Speedup
P: Number of processors
α: Serial portion of code

Graph from Wikipedia, user Peahihawaii, CC-BY-SA 3.0

Limits on performance 3/4

- Moving data has become the major bottleneck in computing.
- Downloading 1GB from Japan to Switzerland consumes roughly the energy of 1 charcoal briquette[1].
- A FLOP costs less than moving one byte[2].
- Key insight: flops are free, moving data is expensive

[1] Energy content charcoal: 10 MJ / kg, kWh per GB: 0.2 (Coroama et al., 2013), Weight charcoal briquette: ~25 grams

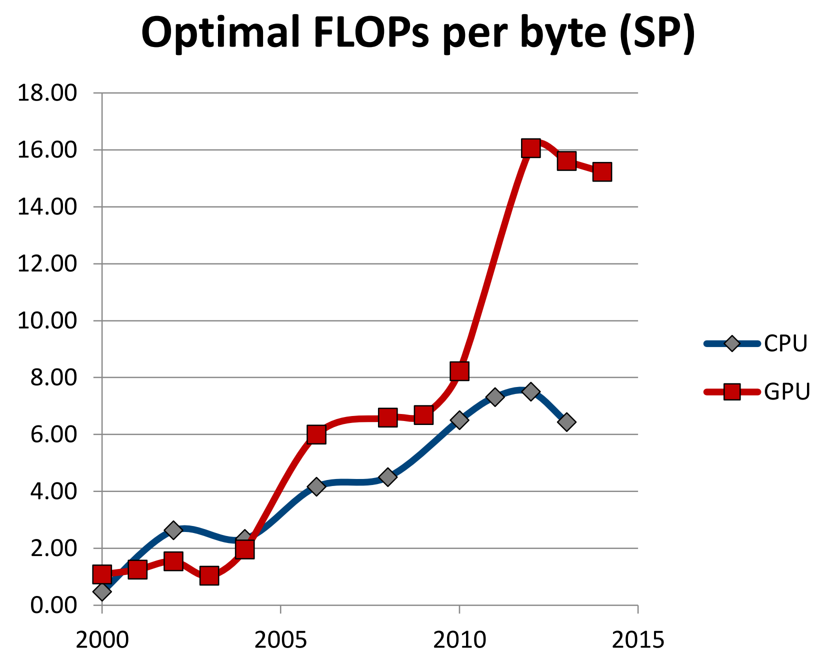
[2]Simon Horst, Why we need Exascale, and why we won't get there by 2020, 2014

Limits on performance 4/4

- A single precision number is four bytes
  - You must perform over 60 operations for each float read on a GPU!
  - Over 25 operations on a CPU!

- This groups algorithms into two classes:
  - Memory bound
    Example: Matrix multiplication
  - Compute bound
    Example: Computing π

- The third limiting factor is latencies
  - Waiting for data
  - Waiting for floating point units
  - Waiting for ...

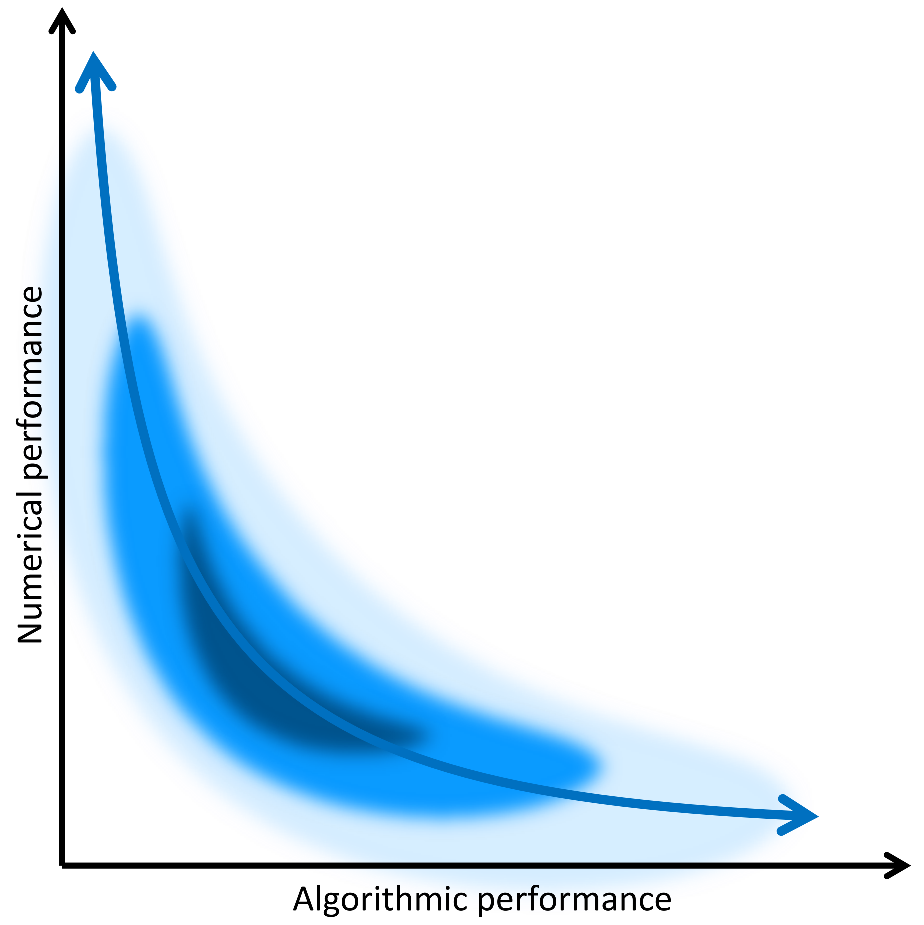


Algorithmic and numerical performance

- Total performance is the product of algorithmic **and** numerical performance
  - Your mileage may vary: algorithmic performance is highly problem dependent

- Many algorithms have low numerical performance
  - Only able to utilize a fraction of the capabilities of processors, and often **worse in parallel**

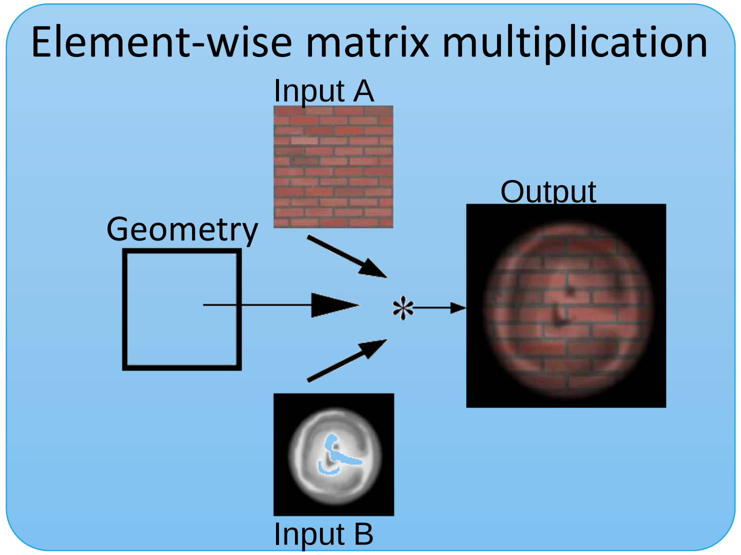
- Need to consider both the algorithm and the architecture for maximum performance

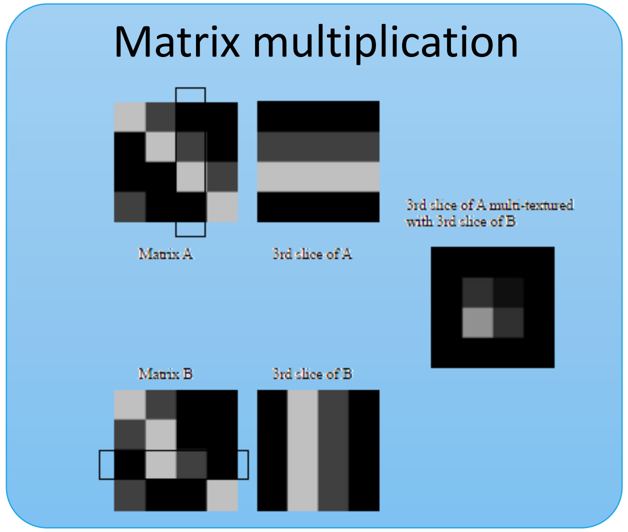


Programming GPUs

Early Programming of GPUs

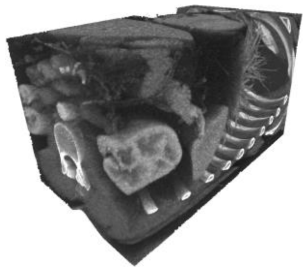
- GPUs were first programmed using OpenGL and other graphics languages
  - Mathematics were written as operations on graphical primitives
  - Extremely cumbersome and error prone
  - Showed that the GPU was capable of outperforming the CPU

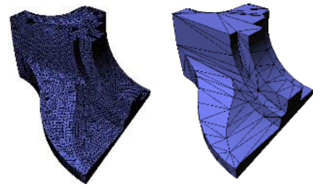


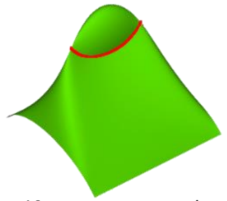


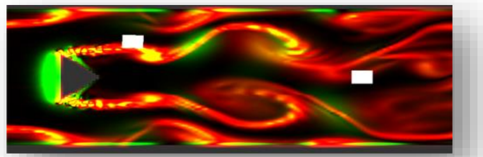
[1] Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

Examples of Early GPU Research at SINTEF

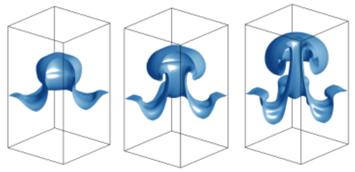

Registration of medical data (~20x)

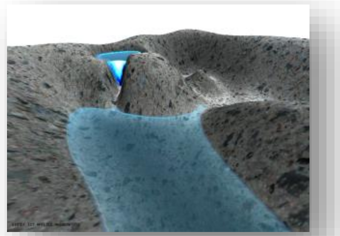

Preparation for FEM (~5x)

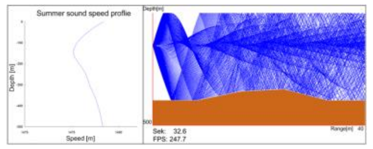

Self-intersection (~10x)


Fluid dynamics and FSI (Navier-Stokes)

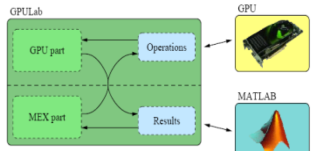
Inpainting (~400x matlab code)


Euler Equations (~25x)


SW Equations (~25x)



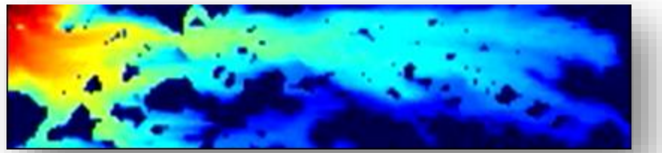
Marine aqoustics (~20x)

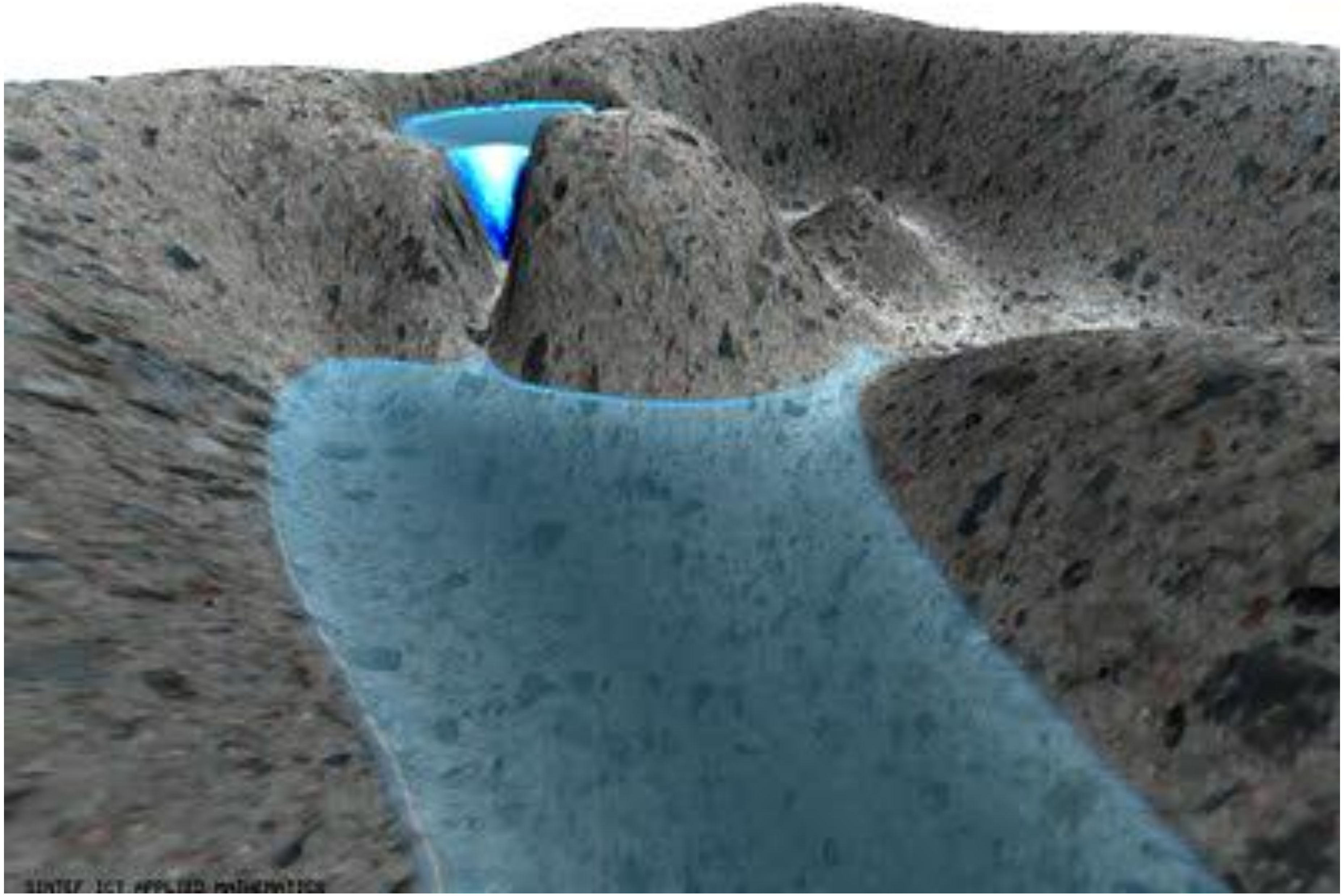


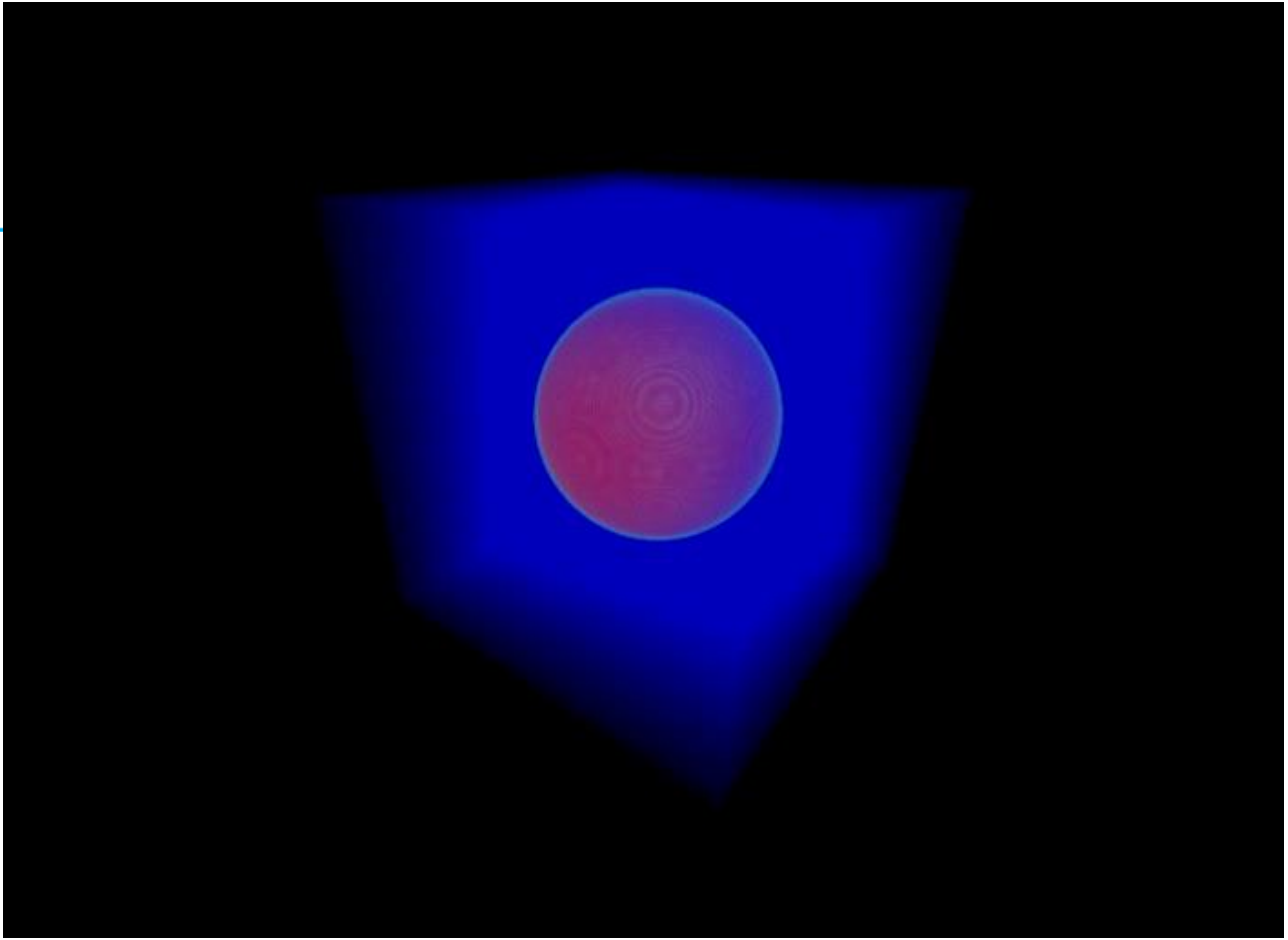
Matlab Interface

$$\begin{bmatrix} b_1 - a_{-\frac{1}{2}} & -a_{\frac{1}{2}} & 0 & 0 & 0 & \cdots & 0 \\ -a_{\frac{1}{2}} & b_2 & -a_{\frac{3}{2}} & 0 & 0 & \cdots & 0 \\ 0 & -a_{\frac{3}{2}} & b_3 & -a_{\frac{5}{2}} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -a_{n-\frac{5}{2}} & b_{n-2} & -a_{n-\frac{3}{2}} & 0 \\ 0 & \cdots & 0 & 0 & -a_{n-\frac{3}{2}} & b_{n-1} & -a_{n-\frac{1}{2}} \\ 0 & \cdots & 0 & 0 & 0 & a_{n-\frac{1}{2}} & b_n - a_{n+\frac{1}{2}} \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ \vdots \\ C_{n-2} \\ C_{n-1} \\ C_n \end{bmatrix}$$

Linear algebra


Water injection in a fluvial reservoir (20x)


SINTEF ICT APPLIED MATHEMATICS

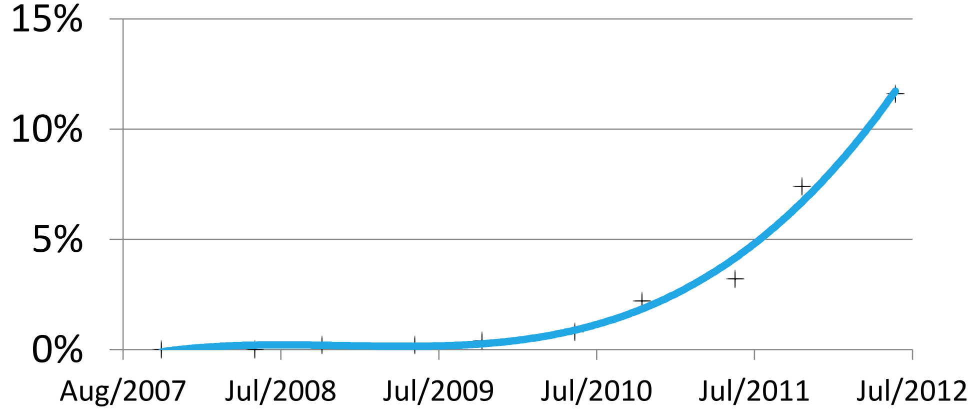


Examples of GPU Use Today

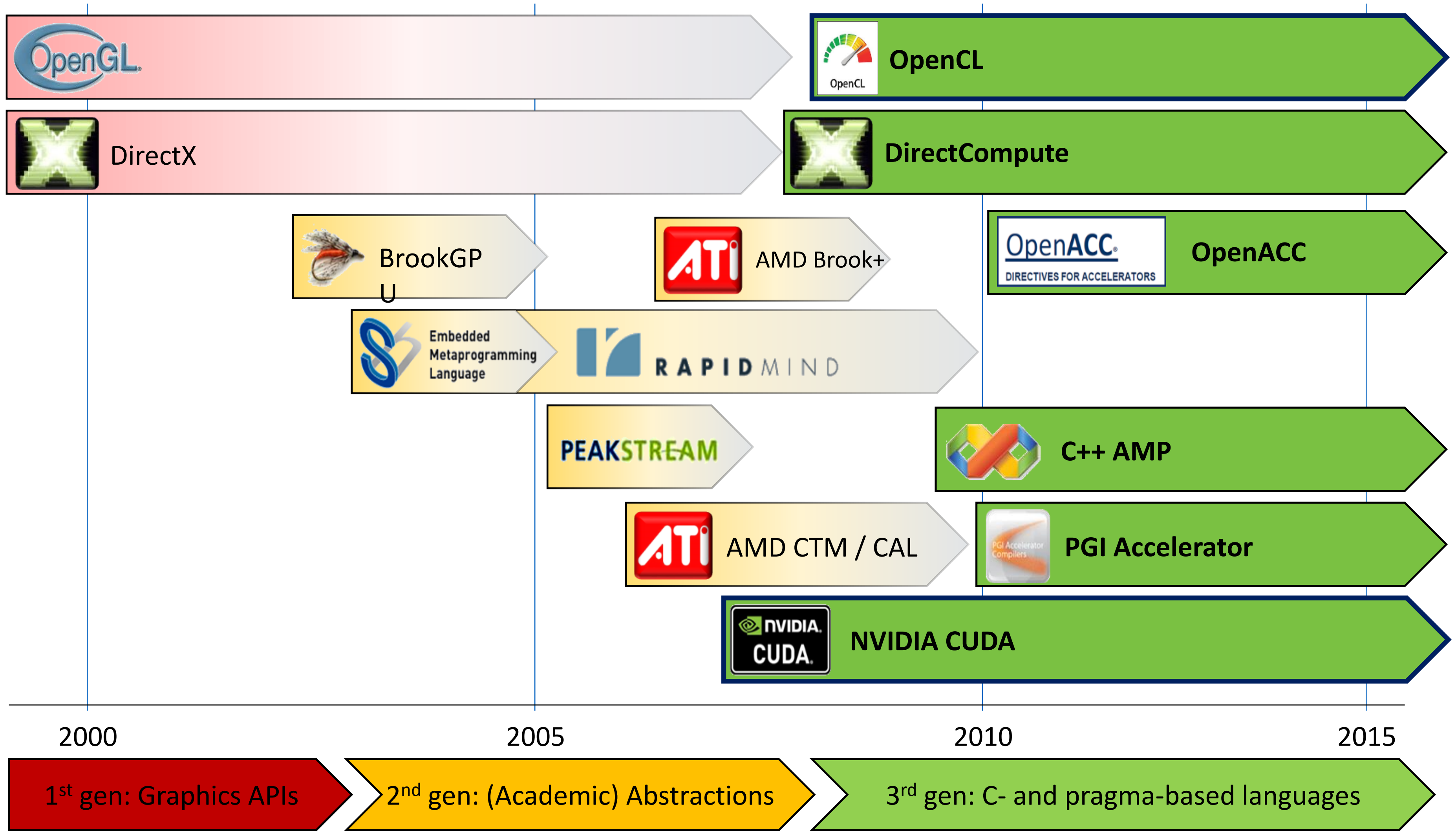
- Thousands of academic papers
- Big investment by large software companies
- Standard in supercomputers
- Huge boost with AI!

Showing 1 - 15 of 1296
Computational Fluid Dynamics Simulations Using
45 x
Application for High-
GPU-Accelerated Large-Eddy Simulation of Turbulent...
Focused Ultrasound -Efficient GPU Simulation Metho...
FMRI Analysis on the GPU
CUDA ACCELERATED FACE RECOGNITION USING LOCAL BINA...
PyCOOL
QnDynCUDA
Integrating CUDA & GNU Autotools
True 4D Image Denoising on the GPU
Application for High-
LASSO Regression Using Distributed GPUs
CUDAfy.NET
GPU and APU computations of Finite Time Lyapunov E...
Exposure Render

GPU Supercomputers on the Top 500 List



GPU Programming Languages



SINTEF

Computing with CUDA

- CUDA has the most mature development ecosystem
  - Released by NVIDIA in 2007
  - Enables programming GPUs using a C-like language
  - Essentially C / C++ with some additional syntax for executing a function in parallel on the GPU

- OpenCL is a very good alternative that also runs on non-NVIDIA hardware (Intel Xeon Phi, AMD GPUs, CPUs)
  - Equivalent to CUDA, but slightly more cumbersome.
  - We will use pyopencl later on!

- For high-level development, languages like OpenACC (pragma based) or C++ AMP (extension to C++) exist
  - Typicall works well for toy problems, but may not always work too well for complex algorithms

Example: Adding two matrices in CUDA 1/2

- We want to add two matrices, a and b, and store the result in c.

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

- For best performance, loop through one row at a time (sequential memory access pattern)

```
void addFunctionCPU(float* c, float* a, float* b,
                    unsigned int cols, unsigned int rows) {
    for (unsigned int j=0; j<rows; ++j) {
        for (unsigned int i=0; i<cols; ++i) {
            unsigned int k = j*cols + i;
            c[k] = a[k] + b[k];
        }
    }
}
```

C++ on CPU

Matrix from Wikipedia: Matrix addition

Example: Adding two matrices in CUDA 2/2

GPU function

Indices

```
__global__ void addMatricesKernel(float* c, float* a, float* b,
                                  unsigned int cols, unsigned int rows) {
     //Indexing calculations
     unsigned int global_x = blockIdx.x*blockDim.x + threadIdx.x;
     unsigned int global_y = blockIdx.y*blockDim.y + threadIdx.y;
     unsigned int k = global_y*cols + global_x;

     //Actual addition
     c[k] = a[k] + b[k];
}
```

Implicit double for loop
for (int blockIdx.x = 0;
         blockIdx.x < grid.x;
         blockIdx.x) { …

Calls GPU function

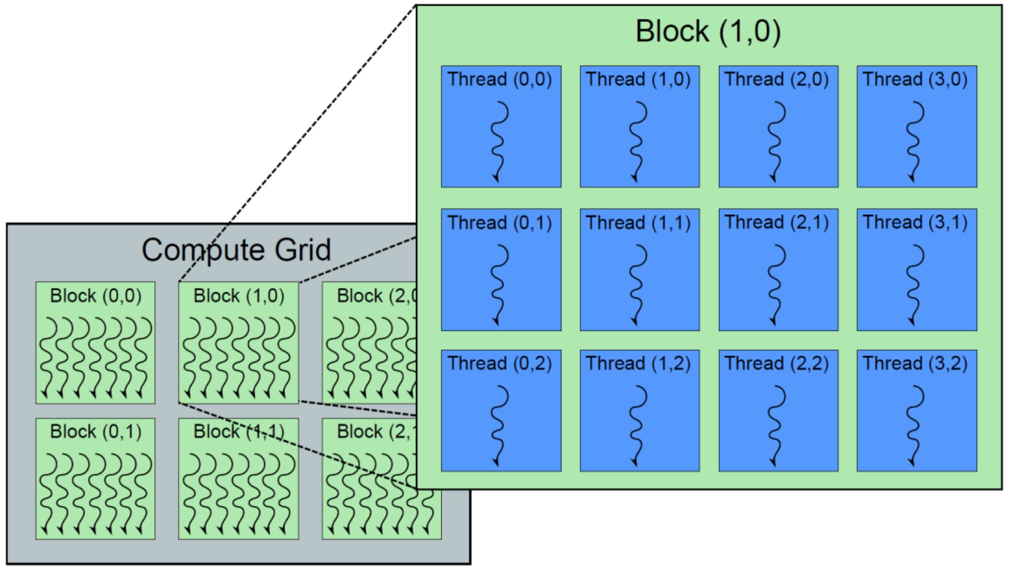
```
void addFunctionCUDA(float* c, float* a, float* b,
           unsigned int cols, unsigned int rows) {
     dim3 block(8, 8);
     dim3 grid(cols/8, rows/8);
     ... //More code here: Allocate data on GPU, copy CPU data to GPU
     addMatricesKernel<<<grid, block>>>(gpu_c, gpu_a, gpu_b, cols, rows);
     ...  //More code here: Download result from GPU to CPU
}
```

SINTEF

Grids and blocks in CUDA

- Two-layered parallelism
  - A block consists of threads: Threads within the same block can cooperate and communicate
  - A grid consists of blocks: All blocks run independently.
  - Blocks and grid can be 1D, 2D, and 3D

- Global synchronization and communication is only possible between kernel launches
  - Really expensive, and should be avoided if possible



CUDA versus OpenCL

- CUDA and OpenCL have a virtually identical programming/execution model
- The largest difference is that OpenCL requires a bit more code to get started, and different concepts have different names.
- The major benefit of OpenCL is that it can run on multiple different devices
  - Supports Intel CPUs, Intel Xeon Phi, NVIDIA GPUs, AMD GPUs, etc.
  - CUDA supports only NVIDIA GPUs.

CUDA versus OpenCL

CUDA	OpenCL
SM (Stream Multiprocessor)	CU (Compute Unit)
Thread	Work-item
Block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
blockIdx * blockDim + threadIdx	get_global_id()
gridDim * blockDim	get_global_size()

CUDA	OpenCL
cudaGetDeviceProperties()	clGetDeviceInfo()
cudaMalloc()	clCreateBuffer()
cudaMemcpy()	clEnqueueRead(Write)Buffer ()
cudaFree()	clReleaseMemObj()
kernel<<<...>>>()	clEnqueueNDRangeKernel()

CUDA	OpenCL
__syncthreads()	barrier()
__threadfence()	No direct equivalent
__threadfence_block()	mem_fence()
No direct equivalent	read_mem_fence()
No direct equivalent	write_mem_fence()

CUDA	OpenCL
__global__ function	__kernel function
__device__ function	No annotation necessary
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

SINTEF

OpenCL matrix addition

```
__kernel void addMatricesKernel(__global float* c, __global float* a,
        __global float* b, unsigned int cols, unsigned int rows) {
    //Indexing calculations
    unsigned int global_x = get_global_id(0);
    unsigned int global_y = get_global_id(1);
    unsigned int k = global_y*cols + global_x;

    //Actual addition
    c[k] = a[k] + b[k];
}

void addFunctionOpenCL() {
    ... //More code here: Allocate data on GPU, copy CPU data to GPU
    //Set arguments
    clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&gpu_c);
    clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&gpu_a);
    clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&gpu_b);
    clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&cols);
    clSetKernelArg(ckKernel, 4, sizeof(cl_int), (void*)&rows);
    // Launch kernel
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &gws, &lws, 0, NULL, NULL);
    ...  //More code here: Download result from GPU to CPU
}
```

GPU function

Calls GPU function

SINTEF

Using Python for GPU Computing

- OpenCL / CUDA are C APIs, which require working in C, and possibly long compilation times
- Even the simplest GPU example will require a lot of boilerplate code
- Pyopencl and PyCuda solves this, by enabling access to the GPU through Python

Example in PyOpenCL – add two vectors

```
%%cl_kernel
__kernel void add_kernel(__global const float *a, __global const float *b,
__global float *c) {
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```

```
#Upload data to the device, allocate output data
…

#Execute program on device
add_kernel(cl_queue, a.shape, None, a_g, b_g, c_g)

#Allocate data on the host for result
c = np.empty_like(a)

#Download data from device to host
cl.enqueue_copy(cl_queue, c, c_g)
```

Summary 1/2

- A function on the GPU is called a kernel
  - Runs in parallel on the GPU
  - Uses massive parallelism to hide memory latency

- The GPU has its own memory
  - Data movement on the GPU is fast
  - Data movement to / from the GPU is slow
  - You need to upload/download data to/from the GPU

- The GPU uses block decomposition
  - In both CUDA and OpenCL, we have blocks consisting of threads (some synchronization possible)
  - The global grid consists of a set of blocks that run in parallel (no synchronization possible)

Summary 2/2

- GPU computing can give you 10x improvement
  - Clever algorithm design can give you higher performance
  - "Porting" to the GPU can give you a slowdown!

- Getting started with GPU computing is easy
  - Installing drivers and tools can be a challenge
  - Using e.g. miniconda environments makes it much easier with tools!
  - Python for developing code is recommended

SINTEF