SINTEF

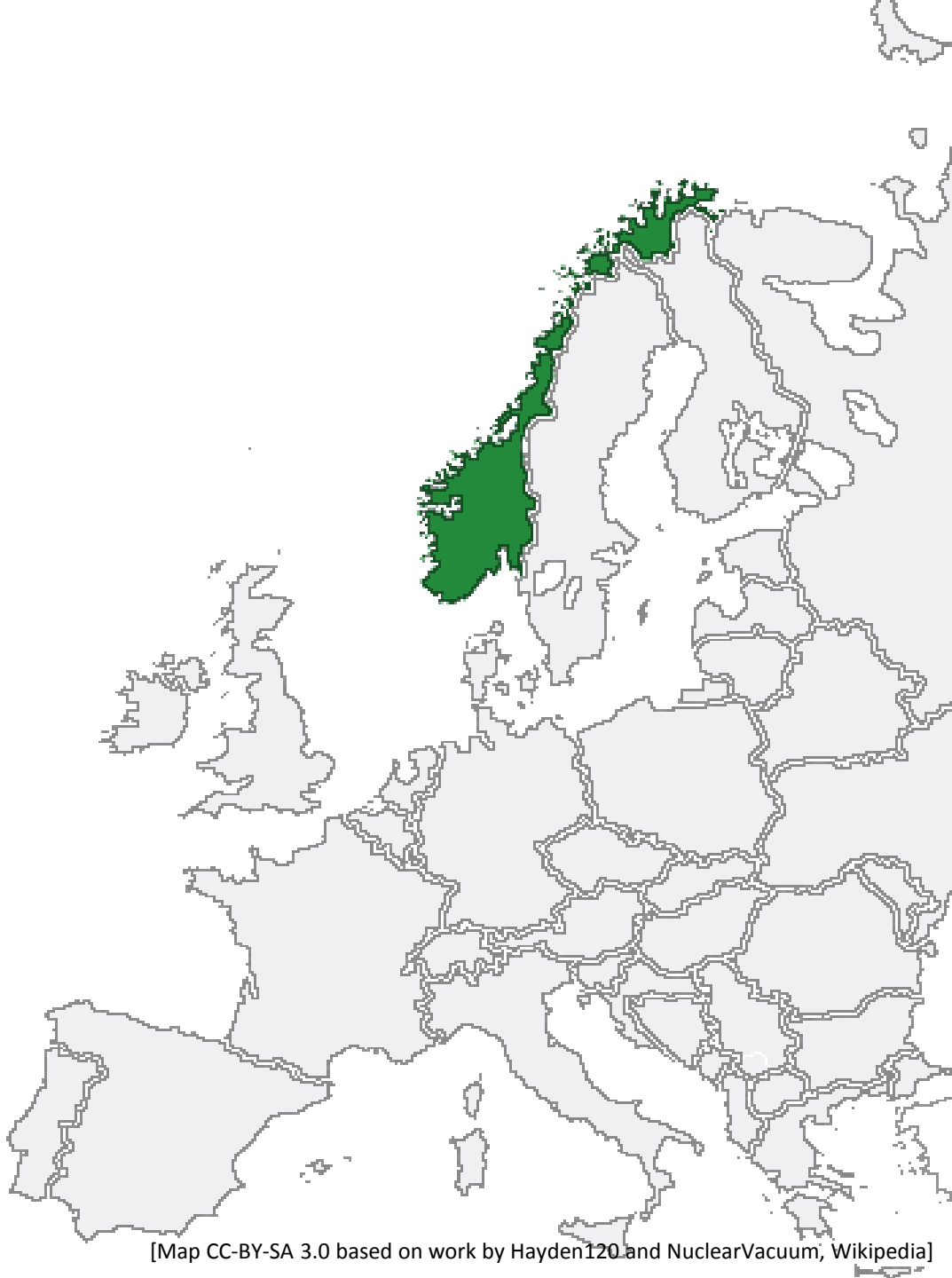
SHORT COURSE ON HIGH-PERFORMANCE SIMULATION WITH HIGH-LEVEL LANGUAGES

INTRODUCTION TO GPUS

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

SINTEF

- Established 1950 by the Norwegian Institute of Technology.
- The largest independent research organisation in Scandinavia.
- A non-profit organisation.
- Motto: “Technology for a better society”.
- Key Figures*
  - 2100 Employees from 70 different countries.
  - 73% of employees are researchers.
  - 3 billion NOK in turnover
    (about 360 million EUR / 490 million USD).
  - 9000 projects for 3000 customers.
  - Offices in Norway, USA, Brazil,
    Chile, and Denmark.



Overview of short course

- Aim of course:
  - To equip you with a set of tools and techniques for working efficiently with high-performance software development.

- Consists of two parts
  - Part 1: Theory. (2-3 hours of lectures)
  - Part 2: Practice. (2+ hours of laboratory exercises)

Outline

- Part 1a – Introduction
  - Motivation for going parallel
  - Multi- and many-core architectures
  - Parallel algorithm design
  - Programming GPUs with CUDA

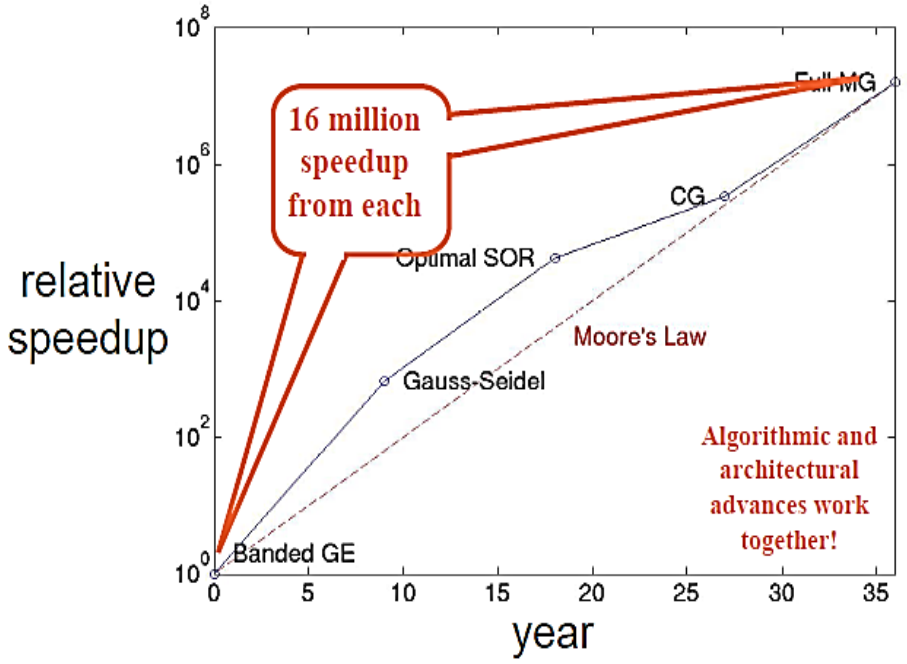
- Part 1b – Solving conservation laws with pyopencl
  - Solving ODEs and PDEs on a computer
  - The heat equation in 1D and 2D
  - The linear wave equation

- Part 1c – Best practices for scientific software development
  - Challenges for scientific software development
  - Best practices for scientific software development

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 both the algorithm and the computer architecture is required.



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

SINTEF

History lesson: development of the microprocessor 1/2

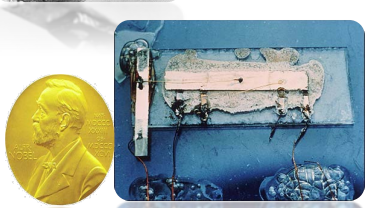
1942: Digital Electric Computer

(Atanasoff and Berry)

1956

1947: Transistor

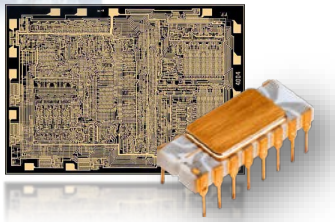
(Shockley, Bardeen, and Brattain)



2000

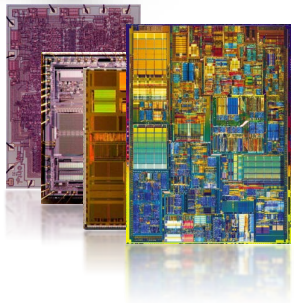
1958: Integrated Circuit

(Kilby)



1971: Microprocessor

(Hoff, Faggin, Mazor)



1971- Exponential growth

(Moore, 1965)

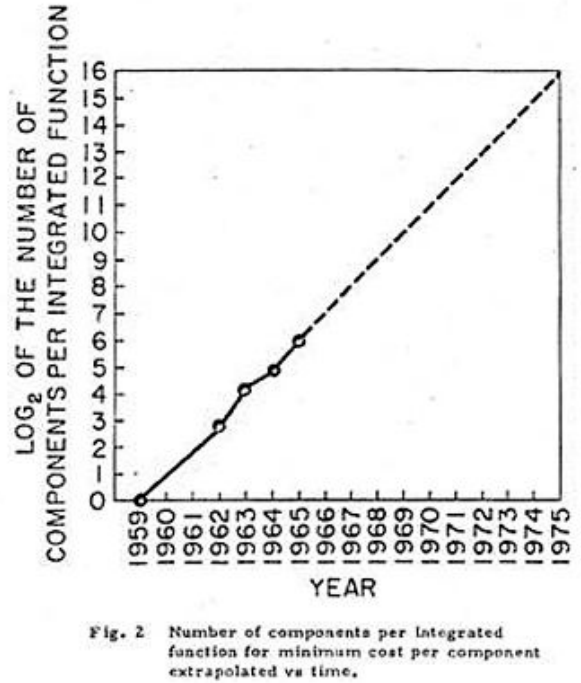
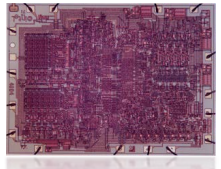
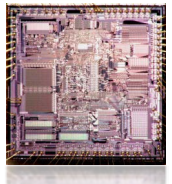


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

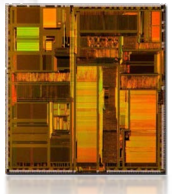
History lesson: development of the microprocessor 2/2



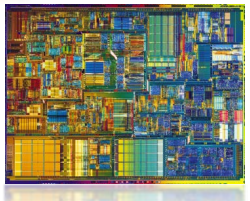
1971: 4004,
2300 trans, 740 KHz



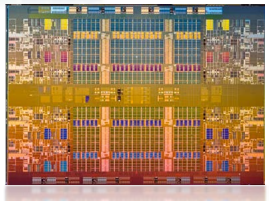
1982: 80286,
134 thousand trans, 8 MHz



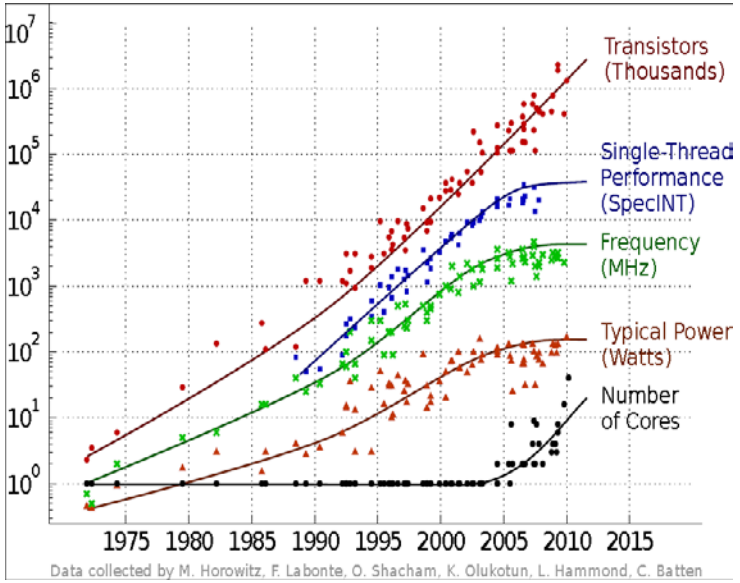
1993: Pentium P5,
1.18 mill. trans, 66 MHz



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

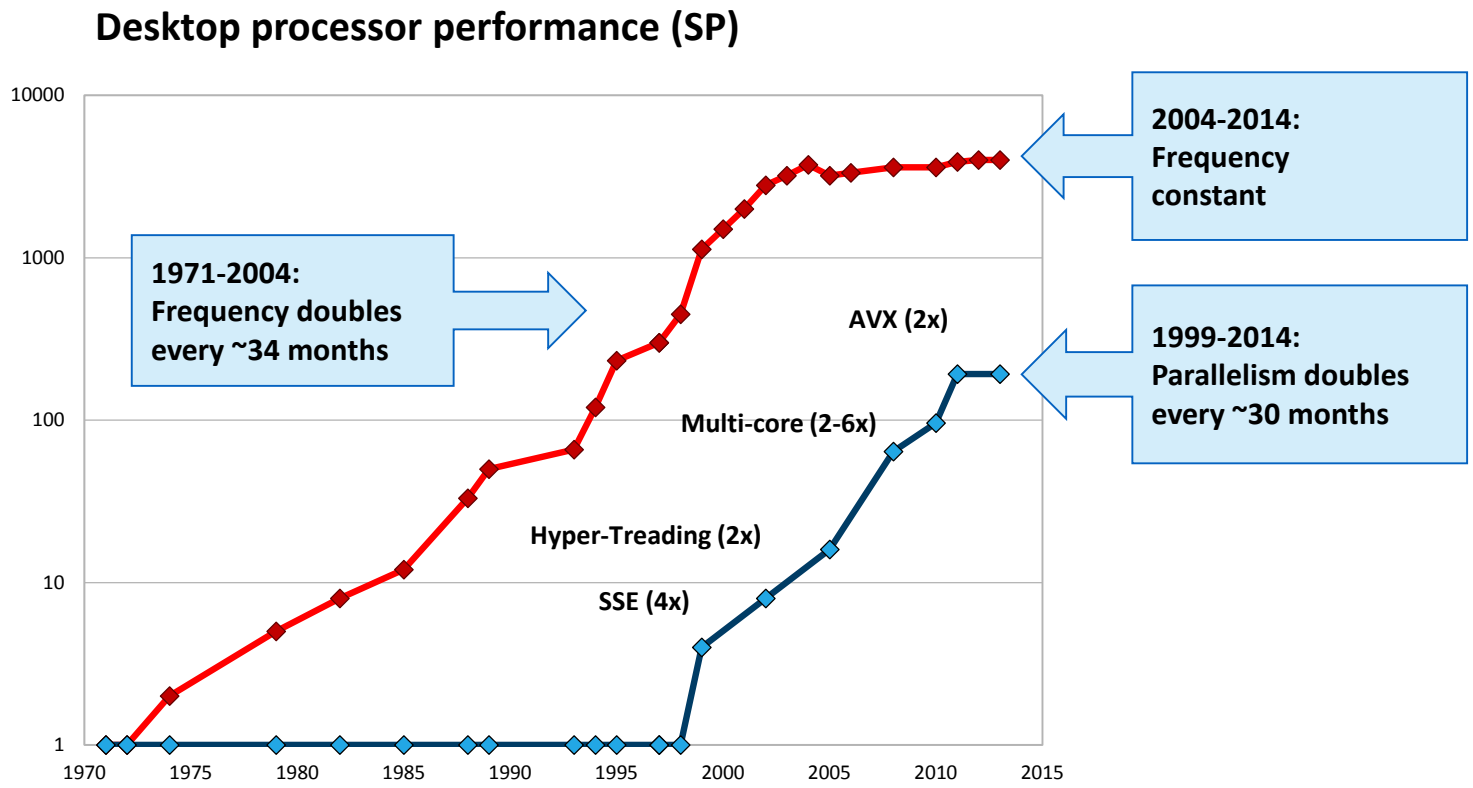


2010: Nehalem
2.3 bill. Trans, 8 cores, 2.66 GHz



SINTEF

End of frequency scaling

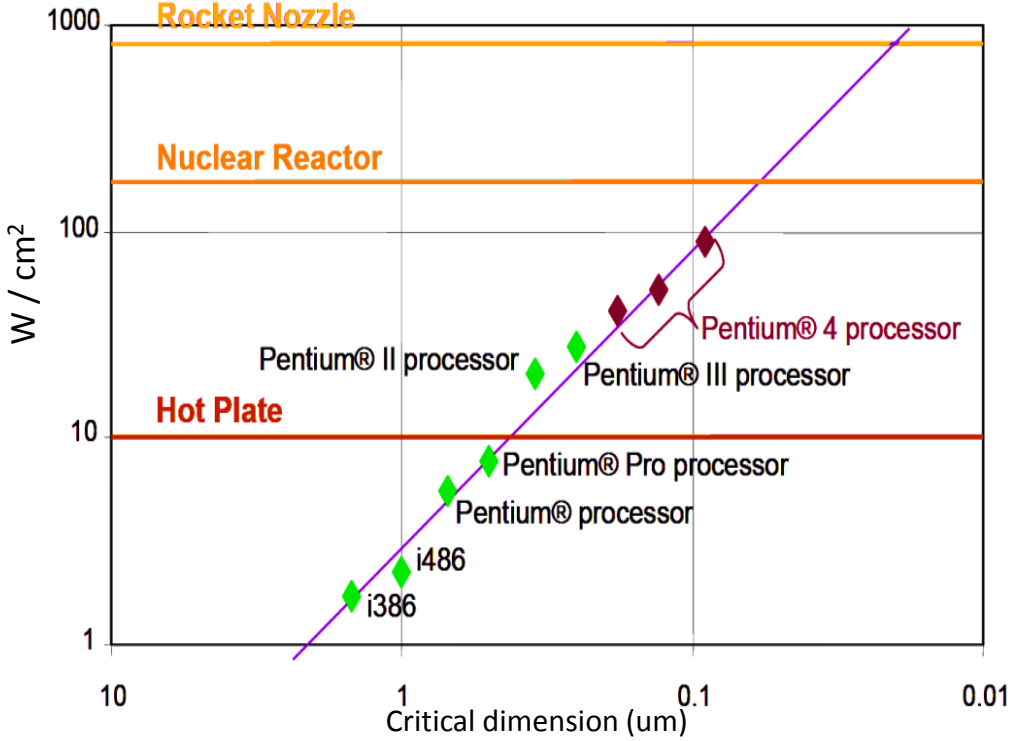


- 1970-2004: Frequency doubles every 34 months (Moore's law for performance)
- 1999-2014: Parallelism doubles every 30 months

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!



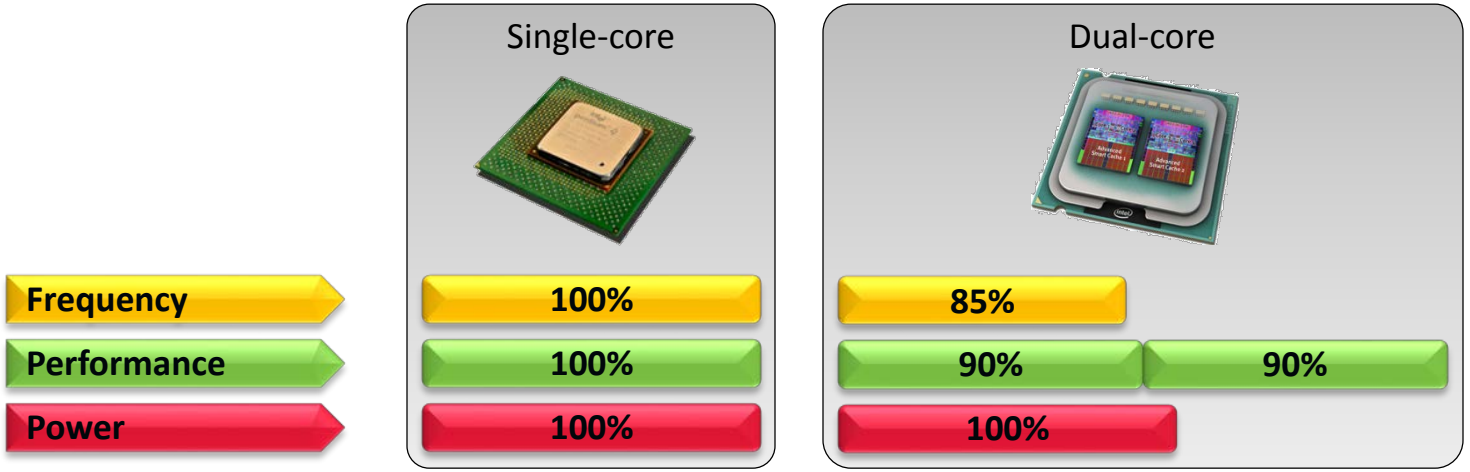
Graph taken from G. Taylor, "Energy Efficient Circuit Design and the Future of Power Delivery" EPEPS'09

SINTEF

Why Parallelism?

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

$$P_d \propto f^3$$

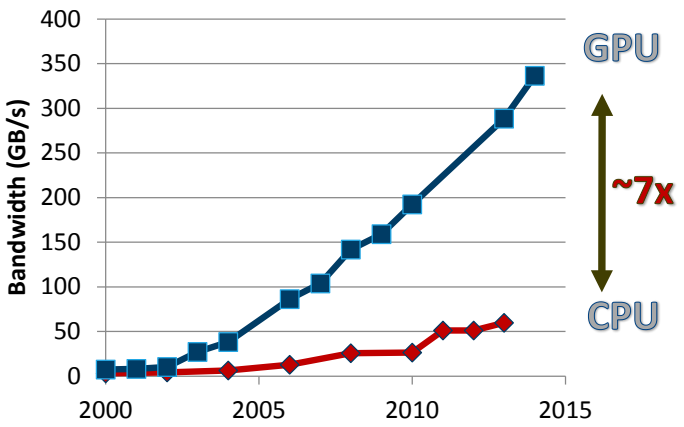


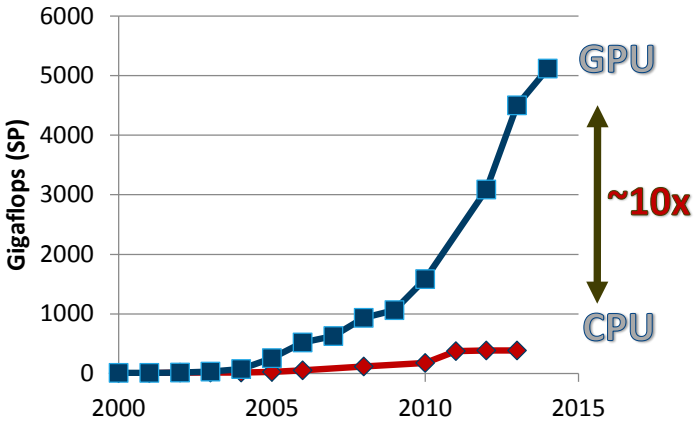
[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

SINTEF

Massive Parallelism: The Graphics Processing Unit

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



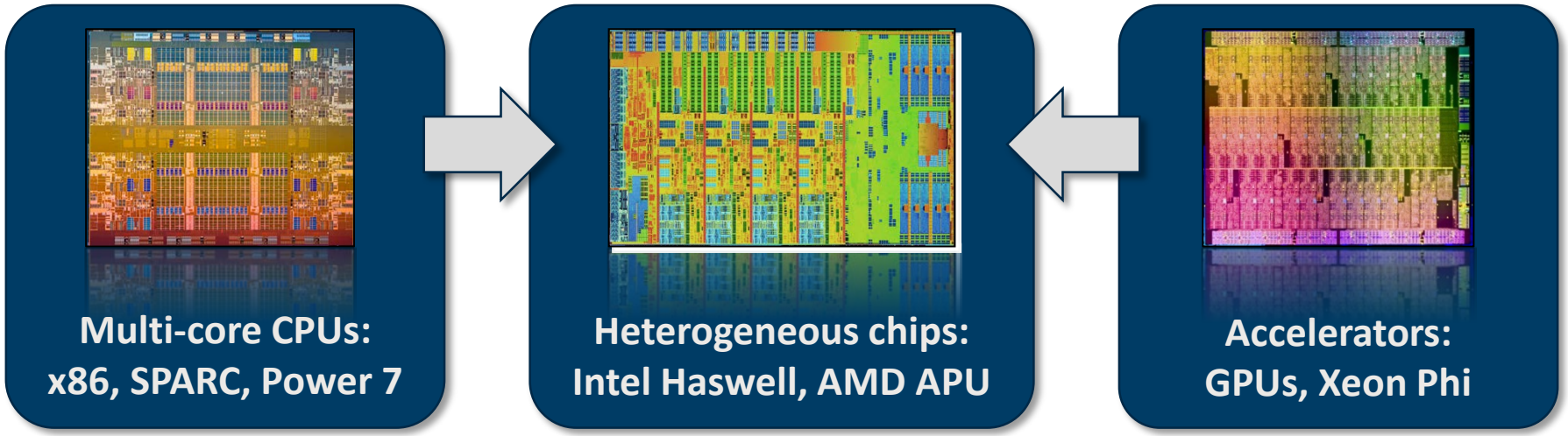


SINTEF

Multi- and many-core processors

Multi- and many-core processor designs

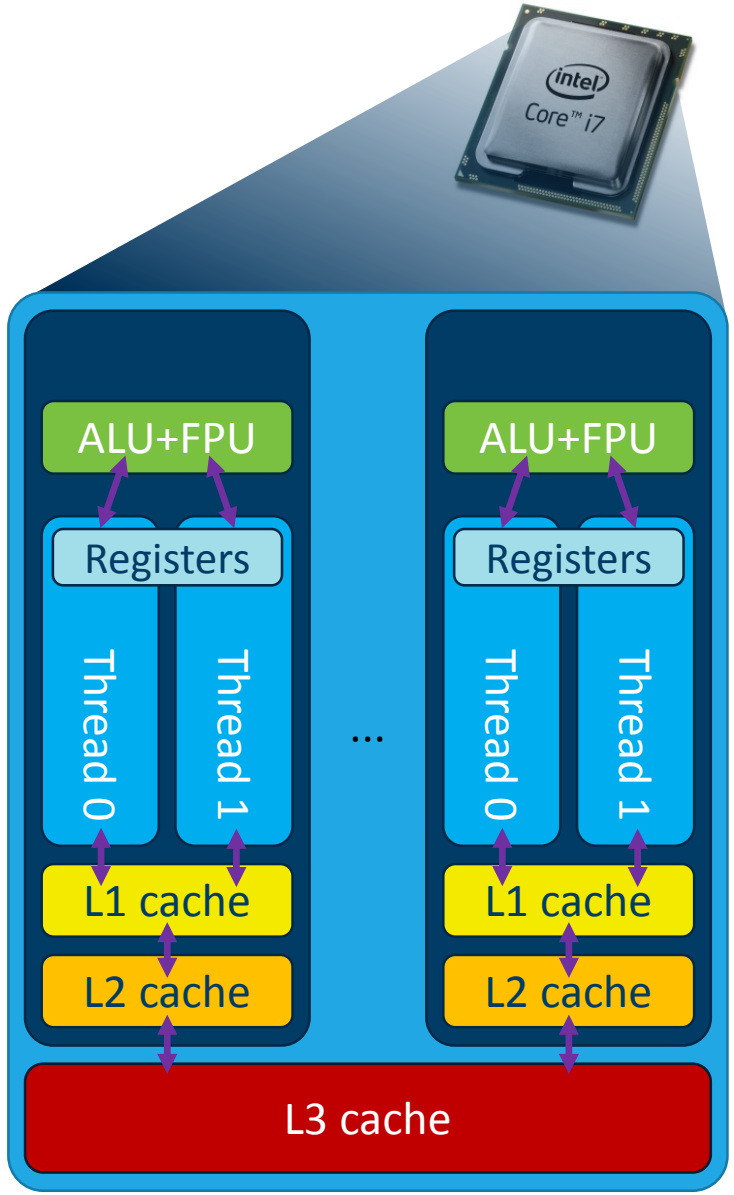
- Today, we have
  - 6-60 processors per chip
  - 8 to 32-wide SIMD instructions
  - Combines both SISD, SIMD, and MIMD on a single chip
  - Heterogeneous cores (e.g., CPU+GPU on single chip)



Multi-core 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!

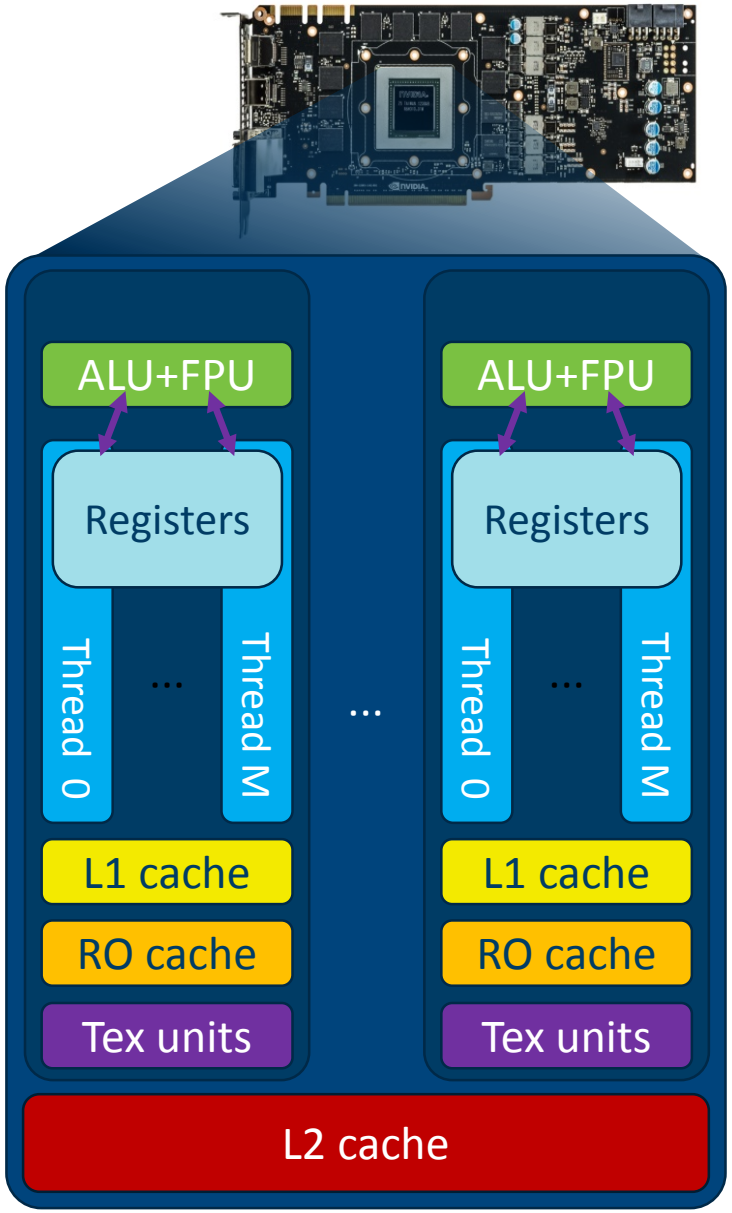


Simplified schematic of CPU design

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!

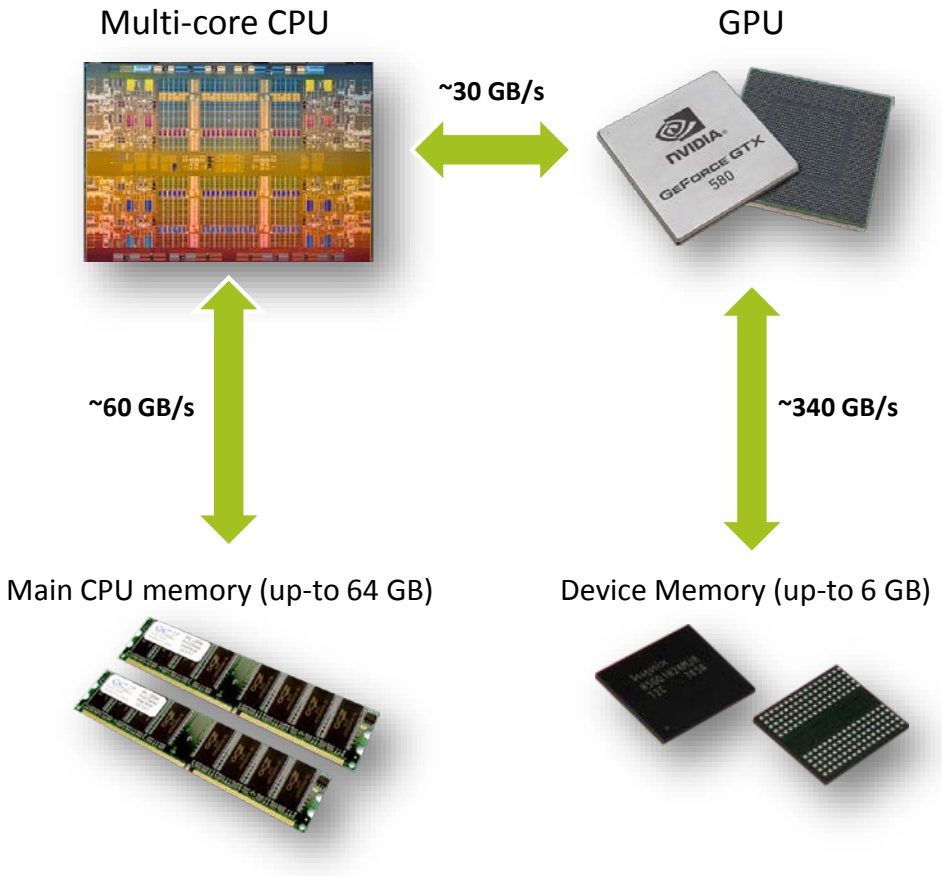


Simplified schematic of GPU design

SINTEF

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)



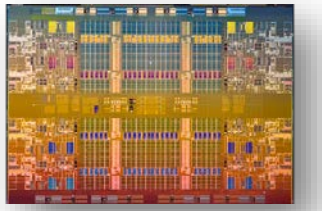
Parallel algorithm design

SINTEF

Type of parallel processing

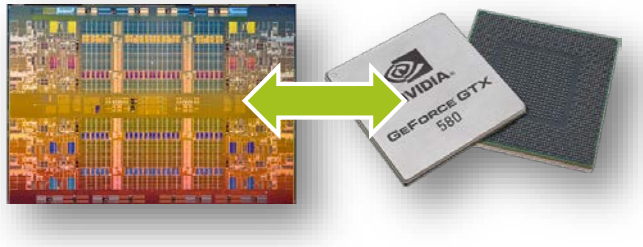
- When the processors are symmetric (identical), we tend to use symmetric multiprocessing.
  - Tasks will take the same amount of time independent of which processor it runs on.
  - All procesors can see everything in memory

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

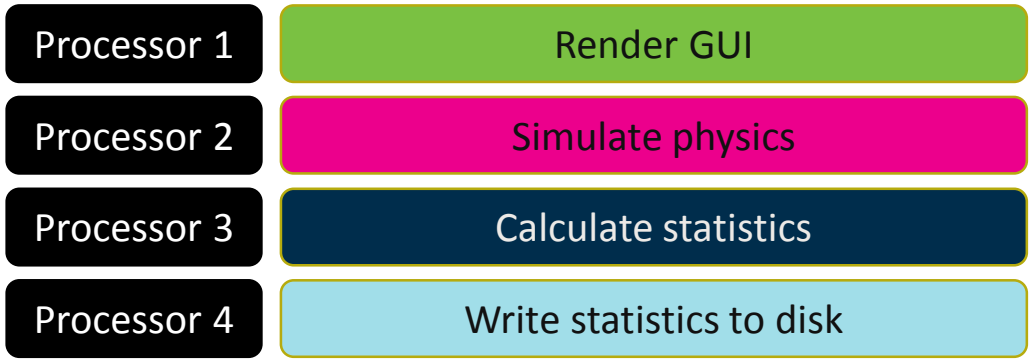
- Data parallelism performs the same operation
  for a set of different input data

- 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:



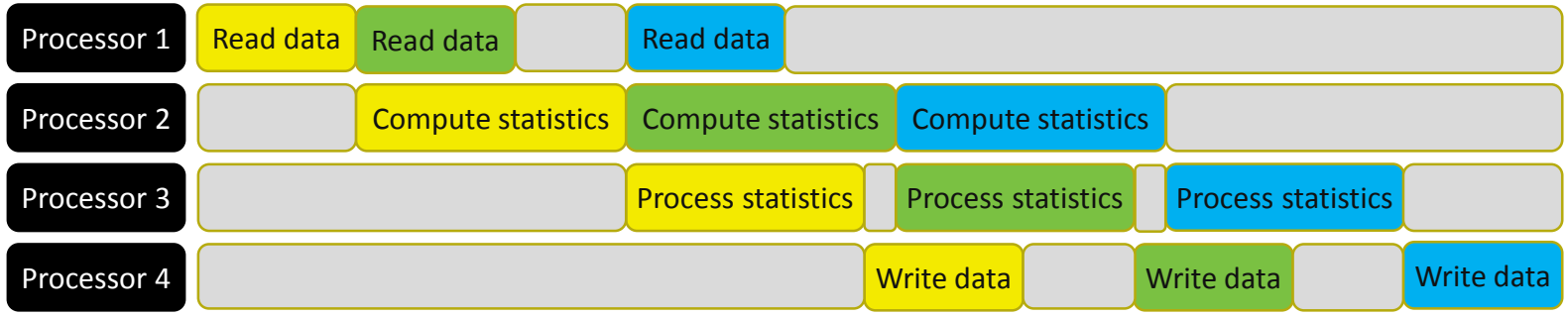
- Note that not all tasks will be able to fully utilize the processor

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!

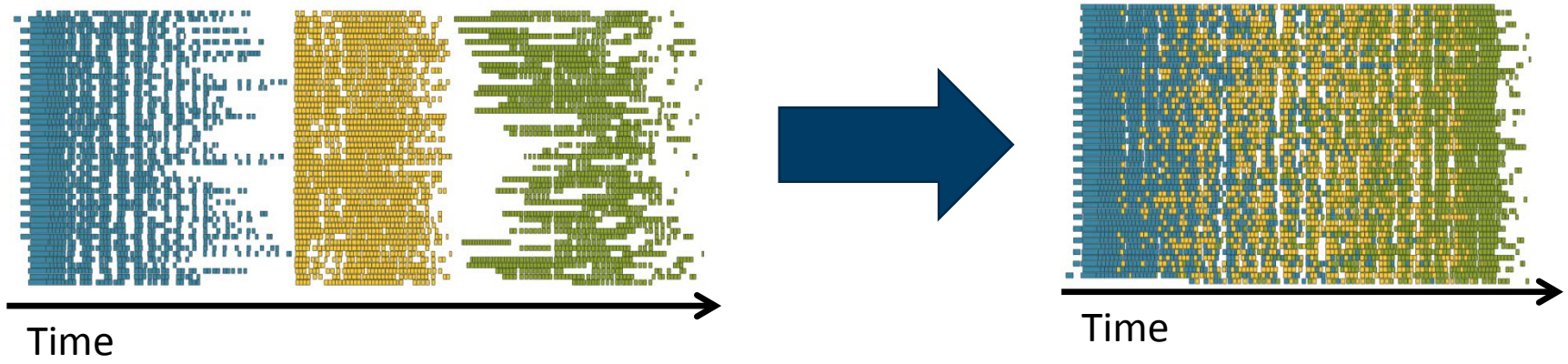
SINTEF

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

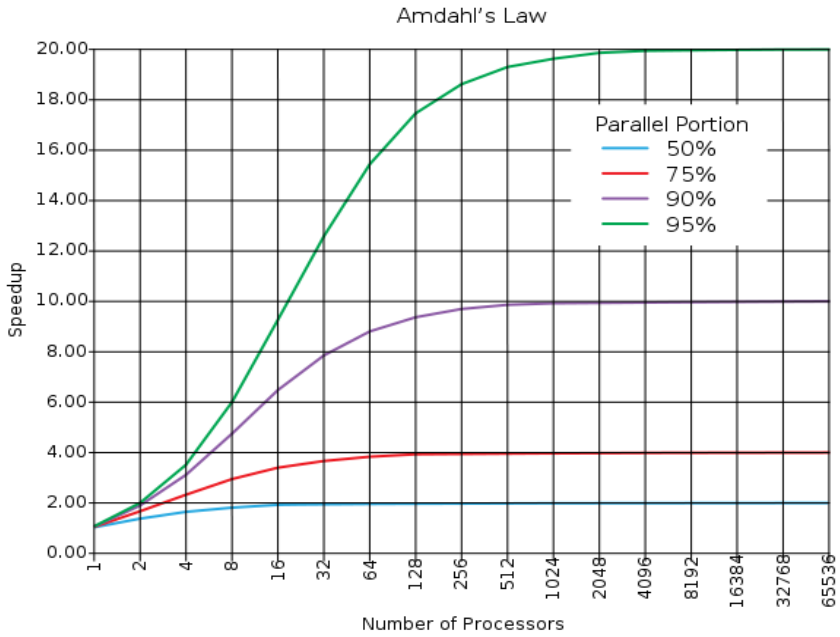
Example from Dongarra, On the Future of High Performance Computing: How to Think for Peta and Exascale Computing, 2012

SINTEF

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!



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

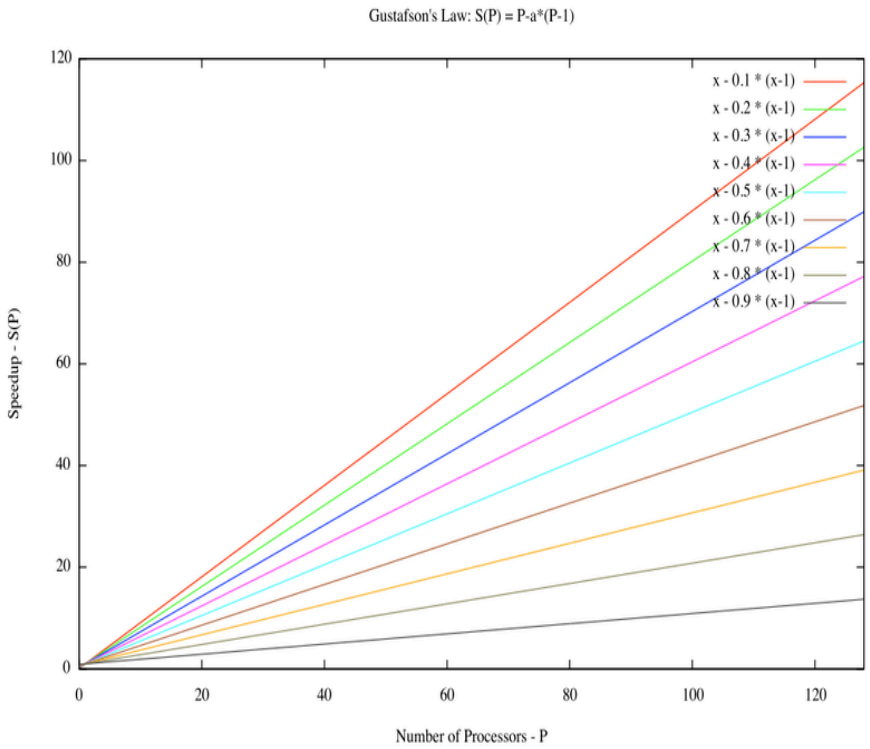
S: Speedup
P: Parallel portion of code
N: Number of processors

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

SINTEF

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

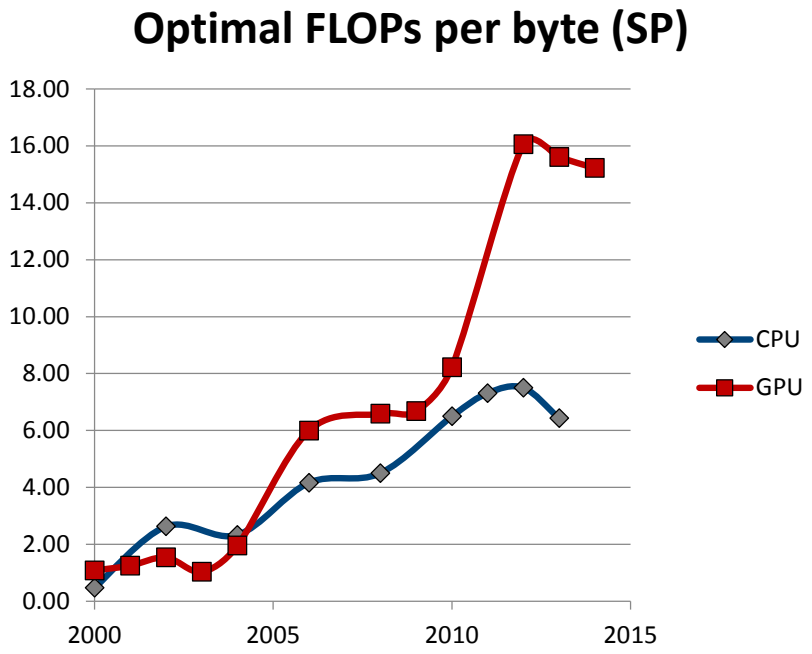
[1] Energy content charcoal: 10 MJ / kg, kWh per GB: 0.2 (Coroama et al., 2013), Weight charcoal briquette: ~25 grams

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

SINTEF

Limits on performance 4/4

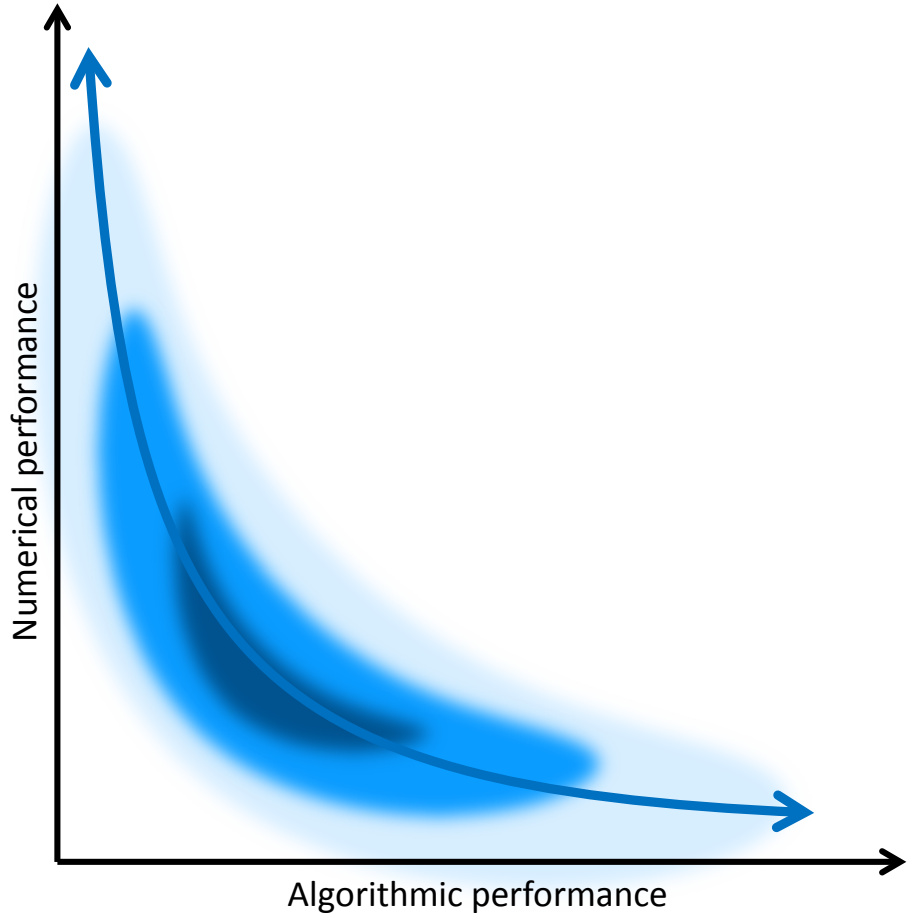
- A single precision number is four bytes
  - You must perform over 60 operations for each float read on a GPU!
  - Over 25 operations on a CPU!
- This groups algorithms into two classes:
  - Memory bound
    Example: Matrix multiplication
  - Compute bound
    Example: Computing π
- The third limiting factor is latencies
  - Waiting for data
  - Waiting for floating point units
  - Waiting for ...



SINTEF

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

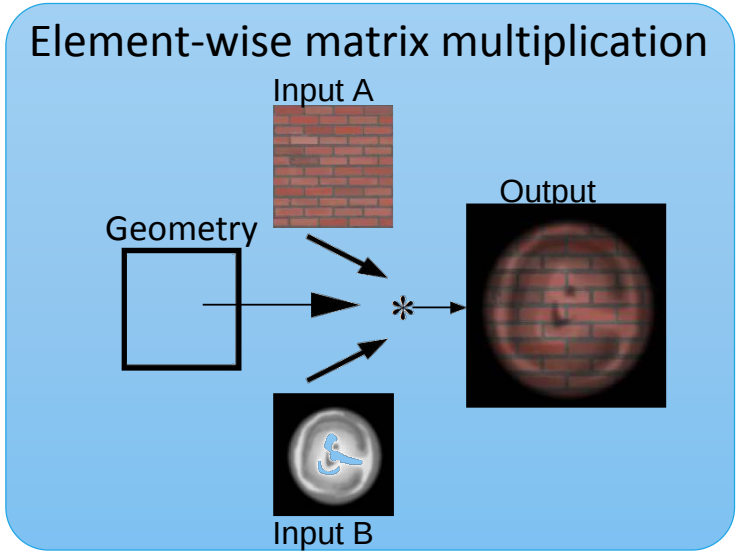


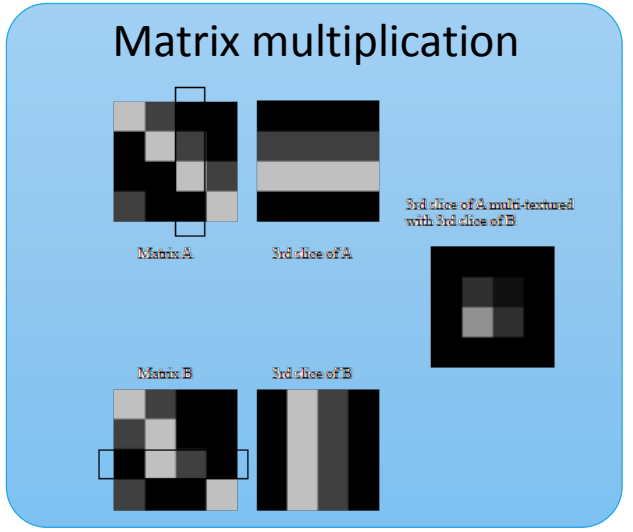
Programming GPUs

SINTEF

Early Programming of GPUs

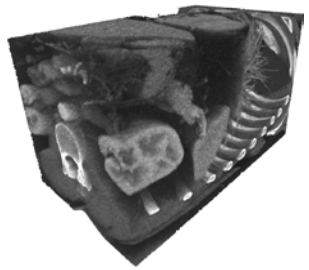
- GPUs were first programmed using OpenGL and other graphics languages
  - Mathematics were written as operations on graphical primitives
  - Extremely cumbersome and error prone
  - Showed that the GPU was capable of outperforming the CPU



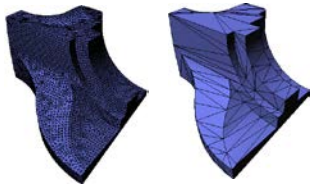


[1] Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

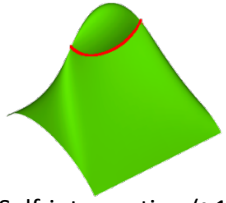
Examples of Early GPU Research at SINTEF



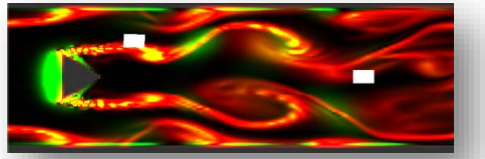
Registration of medical data (~20x)



Preparation for FEM (~5x)

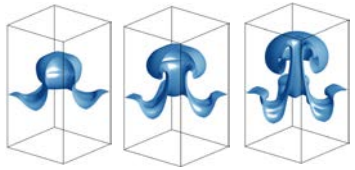


Self-intersection (~10x)

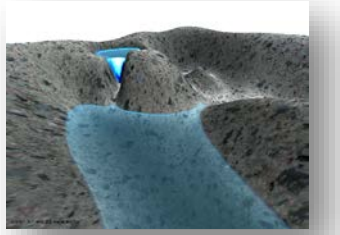


Fluid dynamics and FSI (Navier-Stokes)

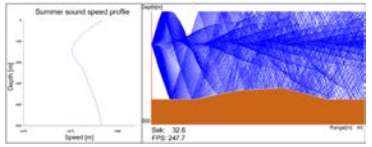
Inpainting (~400x matlab code)



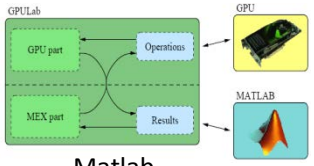
Euler Equations (~25x)



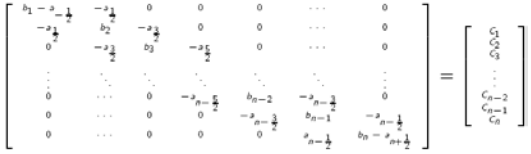
SW Equations (~25x)



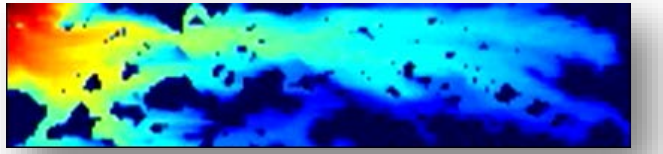
Marine aqoustics (~20x)



Matlab Interface

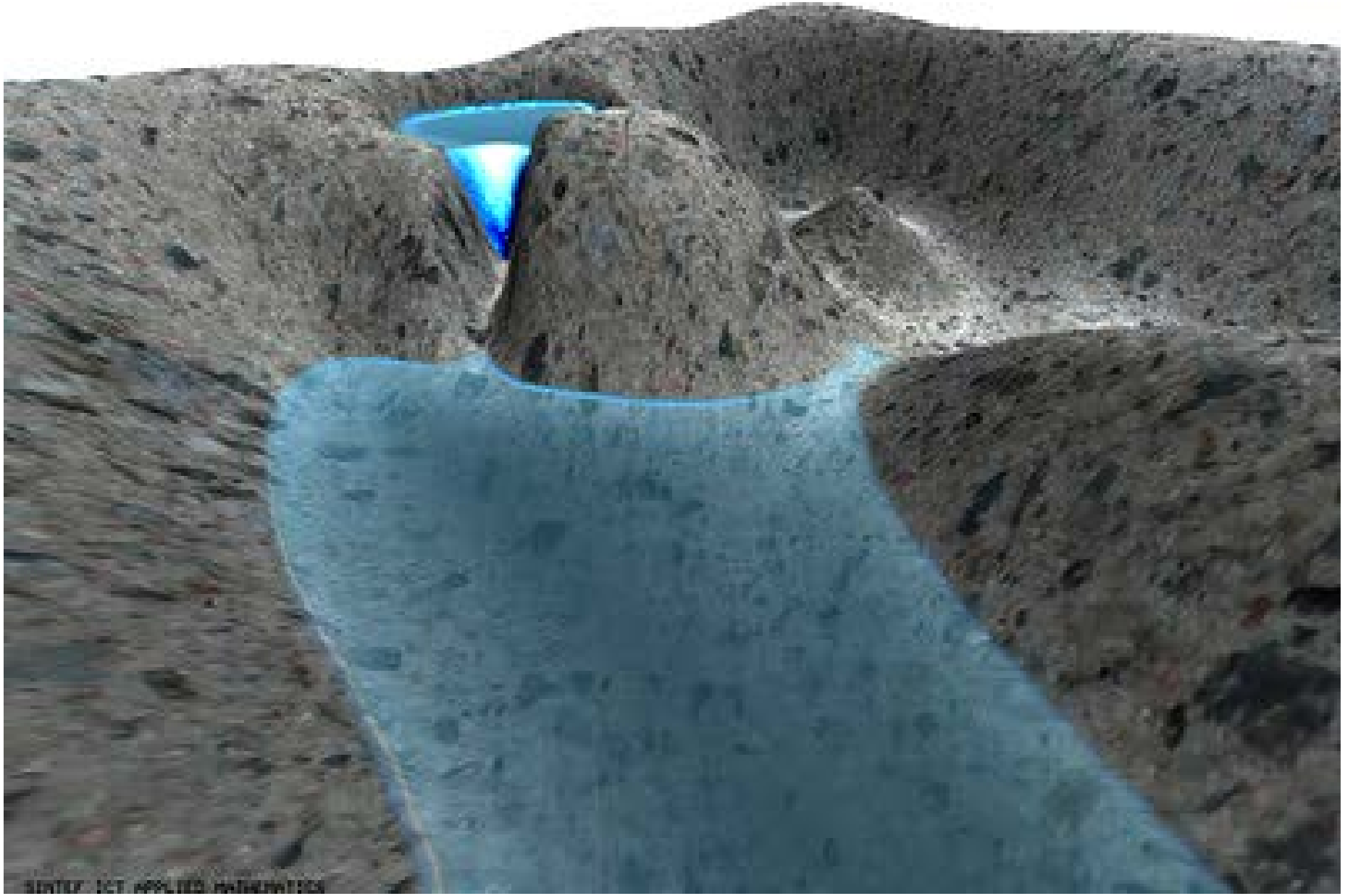


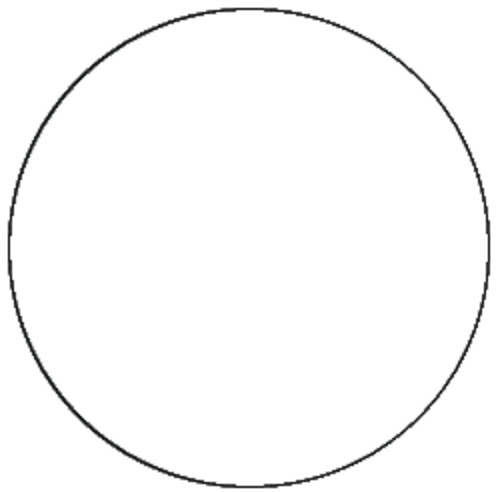
Linear algebra

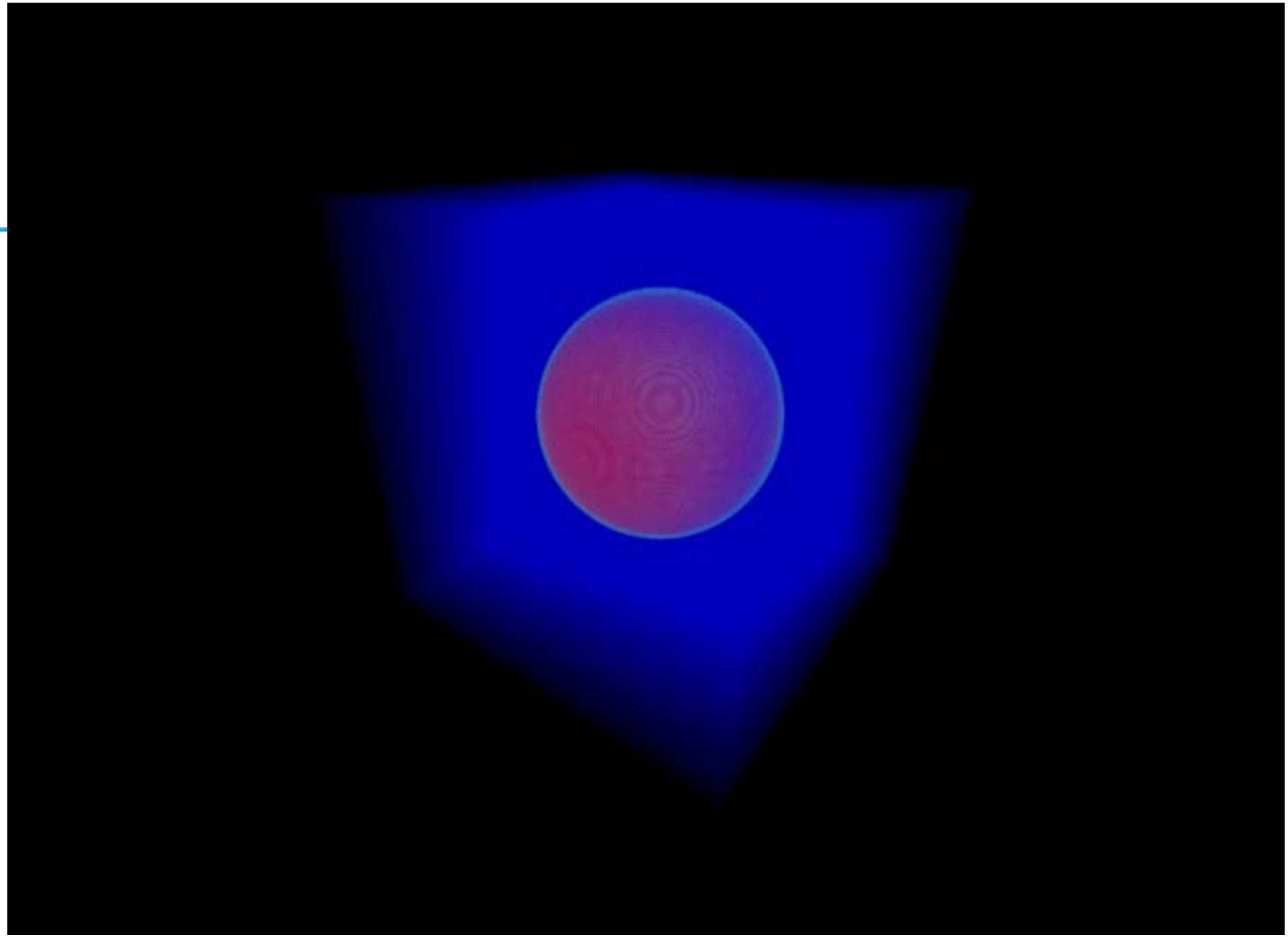


Water injection in a fluvial reservoir (20x)

SINTEF



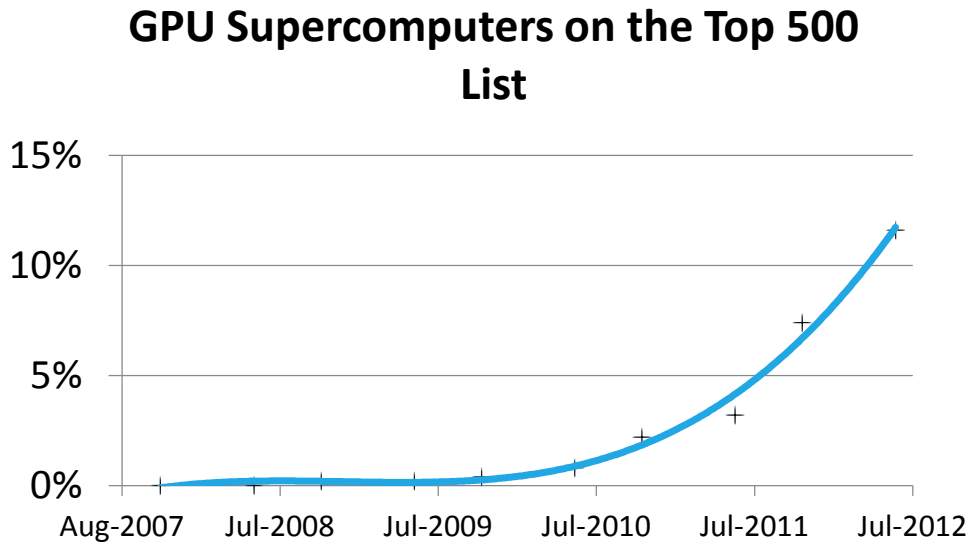




SINTEF

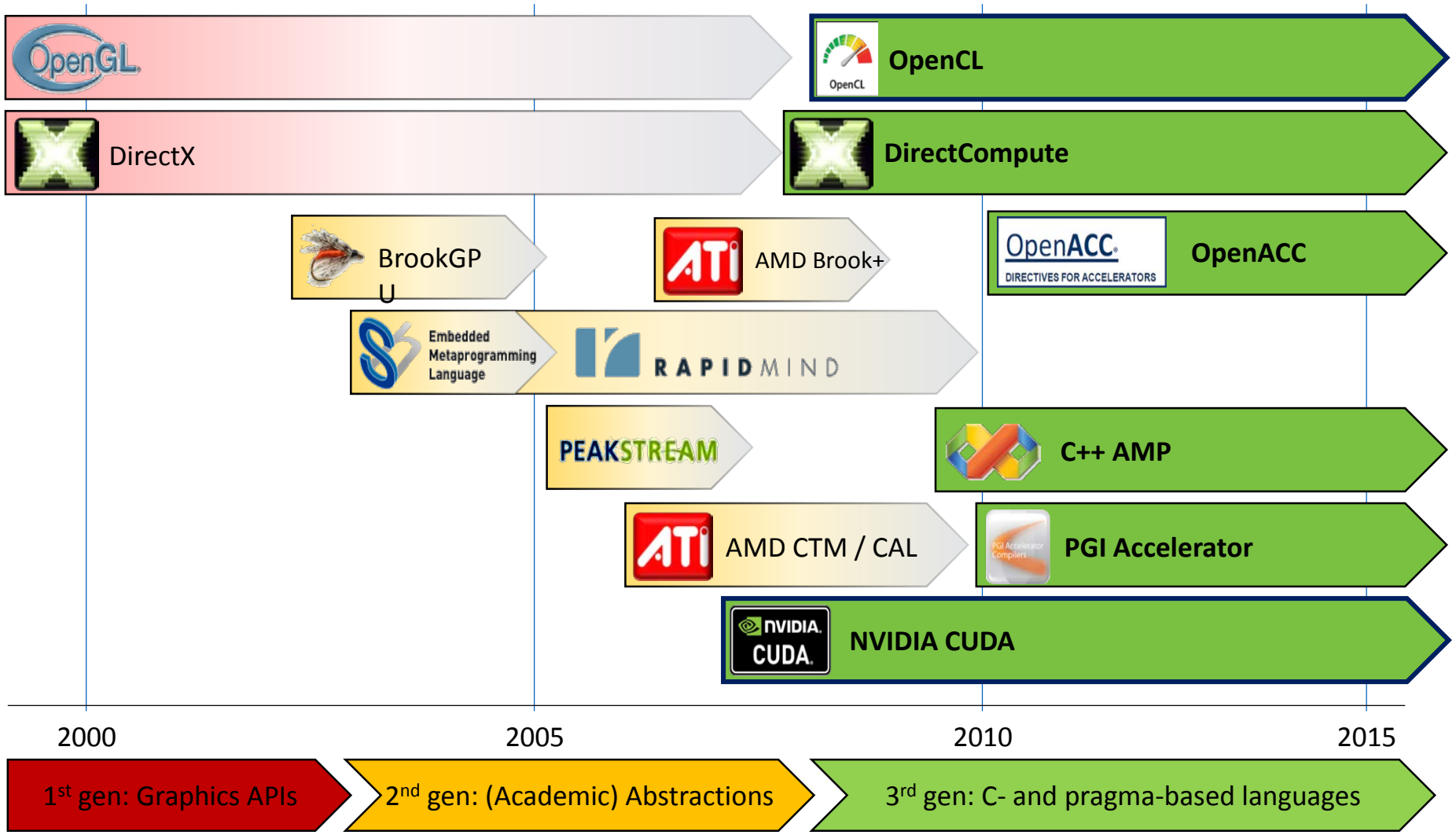
Examples of GPU Use Today

- Thousands of academic papers
- Big investment by large software companies
- Growing use in supercomputers



SINTEF

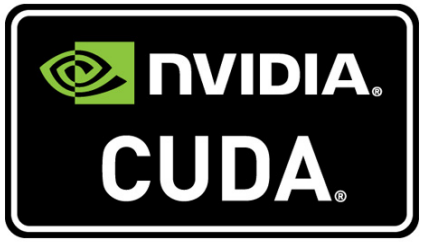
GPU Programming Languages



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)

C++ on CPU

```
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];
        }
    }
}
```

Matrix from Wikipedia: Matrix addition

Example: Adding two matrices in CUDA 2/2

```
__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];
}


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
}
```

GPU function

Indices

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

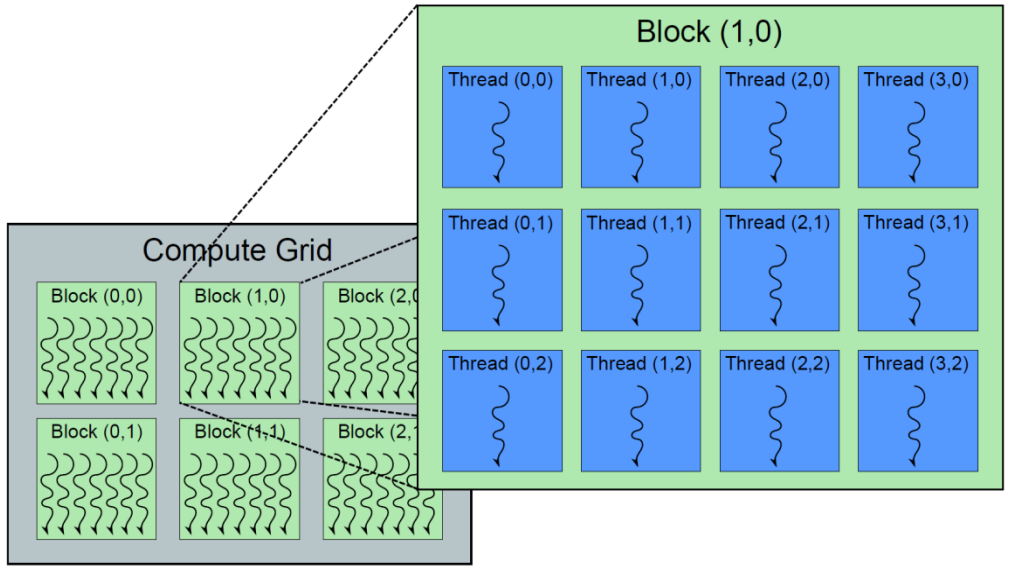
Calls GPU function

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



SINTEF

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

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

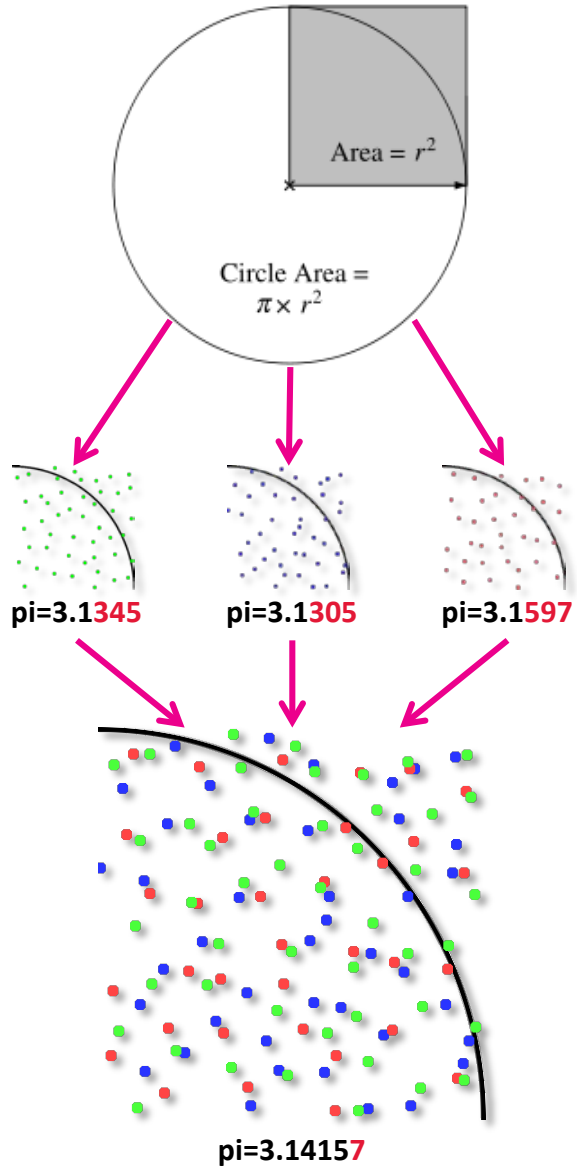
Computing π with CUDA

Computing π with CUDA

- There are many ways of estimating Pi. One way is to estimate the area of a circle.

- Sample random points within one quadrant
- Find the ratio of points inside to outside the circle
  - Area of quarter circle: $A_c = \pi r^2/4$
    Area of square: $A_s = r^2$
  - $\pi = 4\ A_c/A_s \approx 4$ #points inside / #points outside
- Increase accuracy by sampling more points
- Increase speed by using more nodes

- Algorithm:
  1. Sample random points within a quadrant
  2. Compute distance from point to origin
  3. If distance less than r, point is inside circle
  4. Estimate π as 4 #points inside / #points outside



Remember: The algorithms serves as an example:
it's far more efficient to estimate π as 22/7, or 355/113☺

SINTEF

Serial CPU code (C/C++)

```
float computePi(int n_points) {
    int n_inside = 0;
    for (int i=0; i<n_points; ++i) {
     //Generate coordinate
     float x = generateRandomNumber();
     float y = generateRandomNumber();
     //Compute distance
     float r = sqrt(x*x + y*y);
     //Check if within circle
     if (r < 1.0f) { ++n_inside; }
    }
    //Estimate Pi
    float pi = 4.0f * n_inside / static_cast<float>(n_points);
    return pi;
}
```

1

2 & 3

4

SINTEF

Parallel CPU code (C/C++ with OpenMP)

```
float computePi(int n_points) {
    int n_inside = 0;
    #pragma omp parallel for reduction(+:n_inside)
    for (int i=0; i<n_points; ++i) {
     //Generate coordinate
     float x = generateRandomNumber();
     float y = generateRandomNumber();
     //Compute distance
     float r = sqrt(x*x + y*y);
     //Check if within circle
     if (r <= 1.0f) { ++n_inside; }
    }
    //Estimate Pi
    float pi = 4.0f * n_inside / static_cast<float>(n_points);
    return pi;
}
```

Run for loop in parallel using multiple threads

Make sure that every expression involving **n_inside** modifies the global variable using the + operator

SINTEF

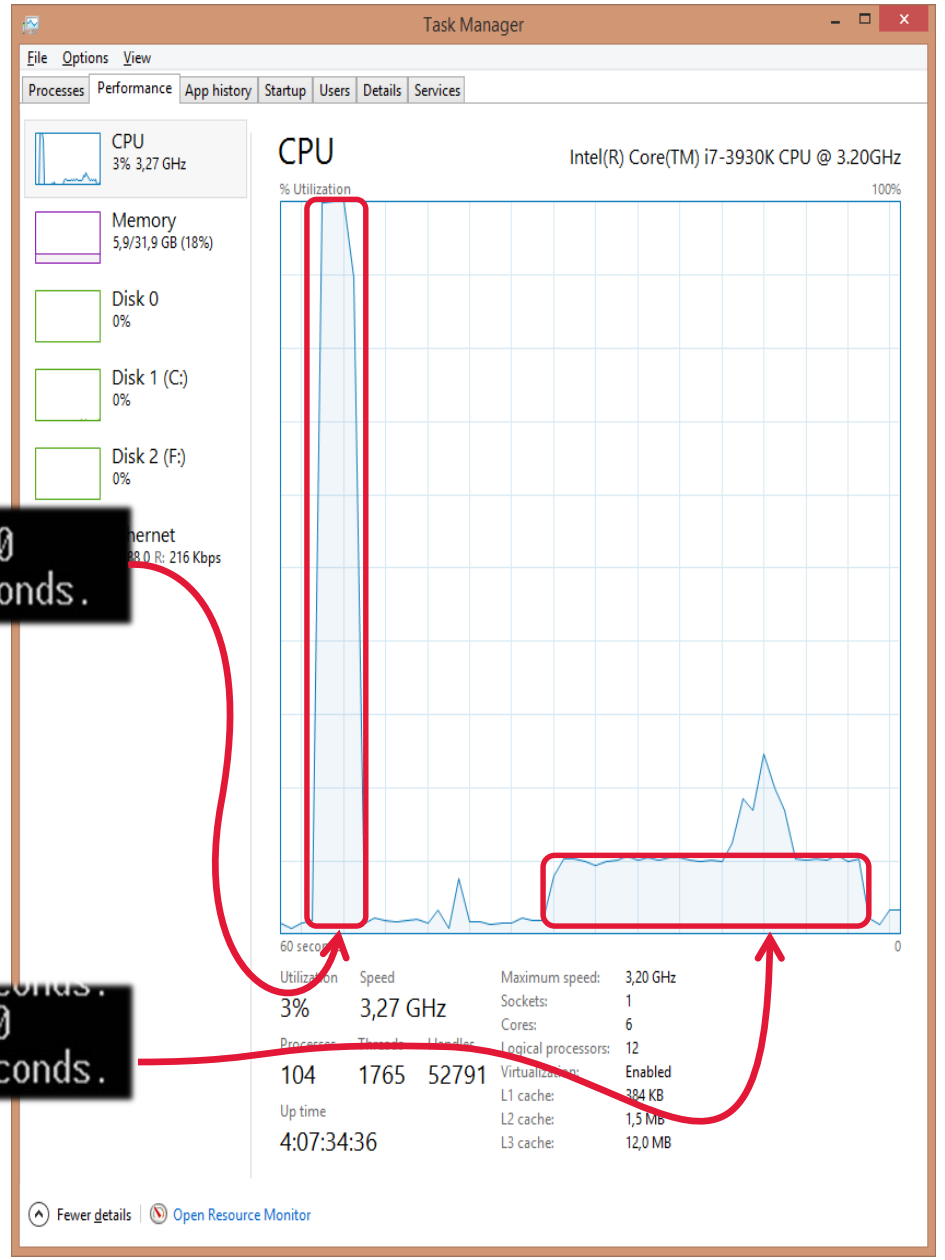
Performance

- Parallel: 3.8 seconds @ 100% CPU

```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141476 in 3.799772 seconds.
```

- Serial: 30 seconds @ 10% CPU

```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141495 in 29.883573 seconds.
```



Parallel GPU version 1 (CUDA) 1/3

```
__global__ void computePiKernel1(unsigned int* output) {
    //Generate coordinate
    float x = generateRandomNumber();
    float y = generateRandomNumber();

    //Compute radius
    float r = sqrt(x*x + y*y);

    //Check if within circle
    if (r <= 1.0f) {
        output[blockIdx.x] = 1;
    } else {
        output[blockIdx.x] = 0;
    }
}
```

GPU function

*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

SINTEF

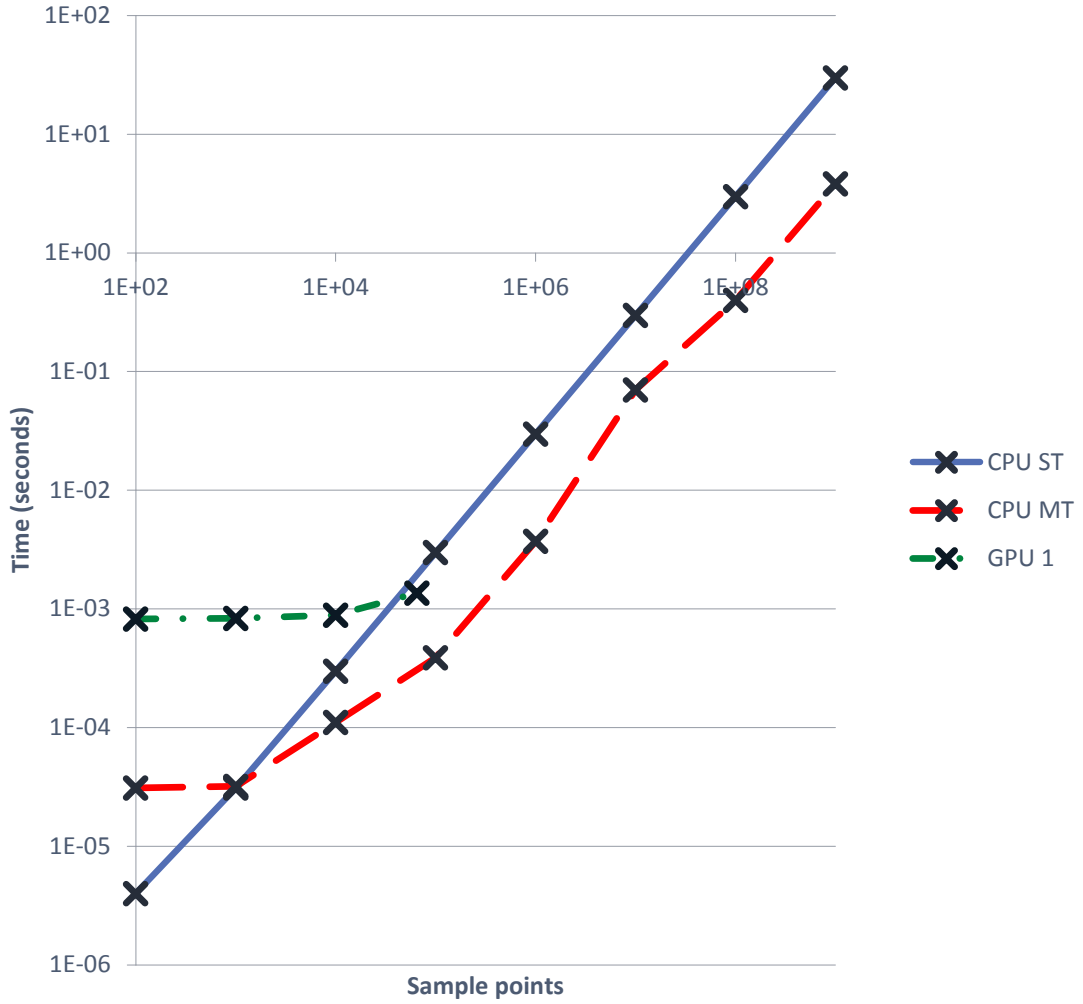
Parallel GPU version 1 (CUDA) 2/3

```
float computePi(int n_points) {
    dim3 grid = dim3(n_points, 1, 1);
    dim3 block = dim3(1, 1, 1);

    //Allocate data on graphics card for output
    cudaMalloc((void**)&gpu_data, gpu_data_size);

    //Execute function on GPU ("lauch the kernel")
    computePiKernel1<<<grid, block>>>(gpu_data);

    //Copy results from GPU to CPU
    cudaMemcpy(&cpu_data[0], gpu_data, gpu_data_size,
               cudaMemcpyDeviceToHost);

    //Estimate Pi
    for (int i=0; i<cpu_data.size(); ++i) {
        n_inside += cpu_data[i];
    }
    return pi = 4.0f * n_inside / n_points;
}
```

SINTEF

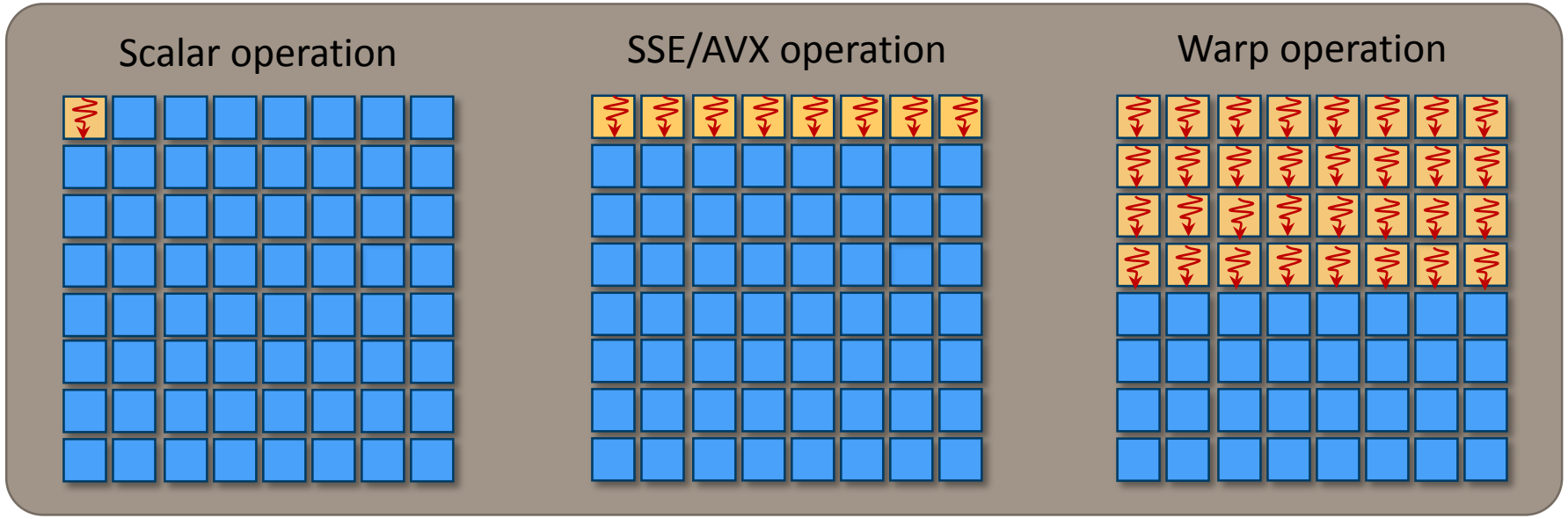
Parallel GPU version 1 (CUDA) 3/3

- Unable to run more than 65535 sample points
- Barely faster than single threaded CPU version for largest size!
- Kernel launch overhead appears to dominate runtime
- The fit between algorithm and architecture is poor:
  - 1 thread per block: Utilizes at most 1/32 of computational power.



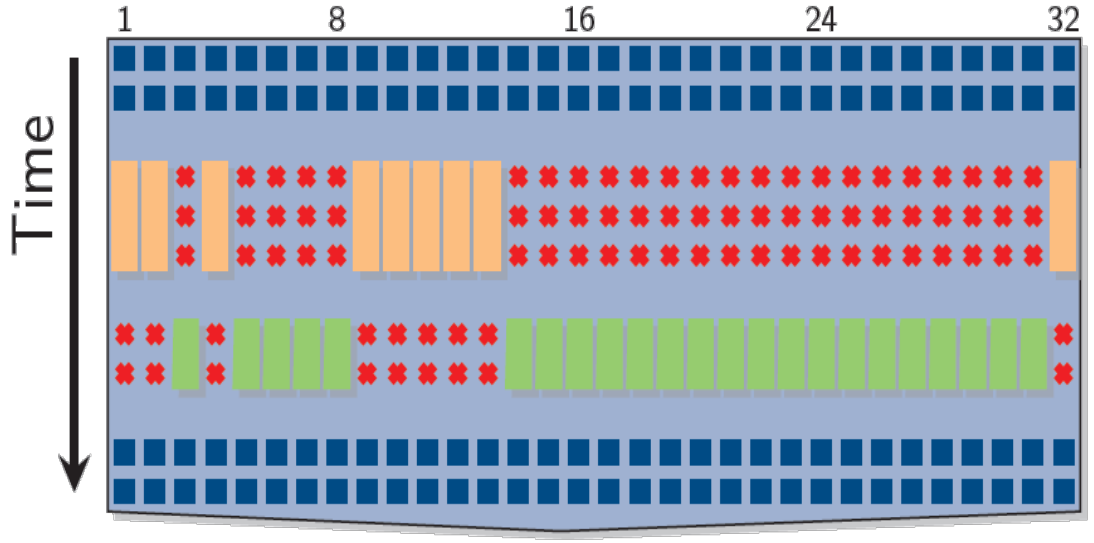
SINTEF

GPU Vector Execution Model



- **CPU scalar:** 1 thread, 1 operand on 1 data element
- **CPU SSE/AVX:** 1 thread, 1 operand on 2-8 data elements
- **GPU Warp:** 32 threads, 32 operands on 32 data elements
  - Exposed as **individual threads**
  - Actually runs the **same instruction**
  - Divergence implies **serialization and masking**

Serialization and masking




```
// Non-divergent code
if( x > 0 ) {
    y = pow(x, exp);
    y *= Ks;
    z = y + Ka;
} else {
    x = 0;
    z = Ka;
}
// Non-divergent code
```

Hardware automatically serializes and masks divergent code flow:

- Execution time is the sum of all branches taken
- Programmer is relieved of fiddling with element masks (which is necessary for SSE/AVX)
- Worst case 1/32 performance
- Important to **minimize divergent code flow within warps**!
  - Move conditionals into data, use min, max, conditional moves.

Parallel GPU version 2 (CUDA) 1/2

```
__global__ void computePiKernel2(unsigned int* output) {
        //Generate coordinate
        float x = generateRandomNumber();
        float y = generateRandomNumber();

        //Compute radius
        float r = sqrt(x*x + y*y);

        //Check if within circle
        if (r <= 1.0f) {
                output[blockIdx.x*blockDim.x + threadIdx.x] = 1;
        } else {
                output[blockIdx.x*blockDim.x + threadIdx.x] = 0;
        }
}

float computePi(int n_points) {
        dim3 grid = dim3(n_points/32, 1, 1);
        dim3 block = dim3(32, 1, 1);
        …
        //Execute function on GPU (“lauch the kernel”)
        computePiKernel1<<<grid, block>>>(gpu_data);
        …
}
```

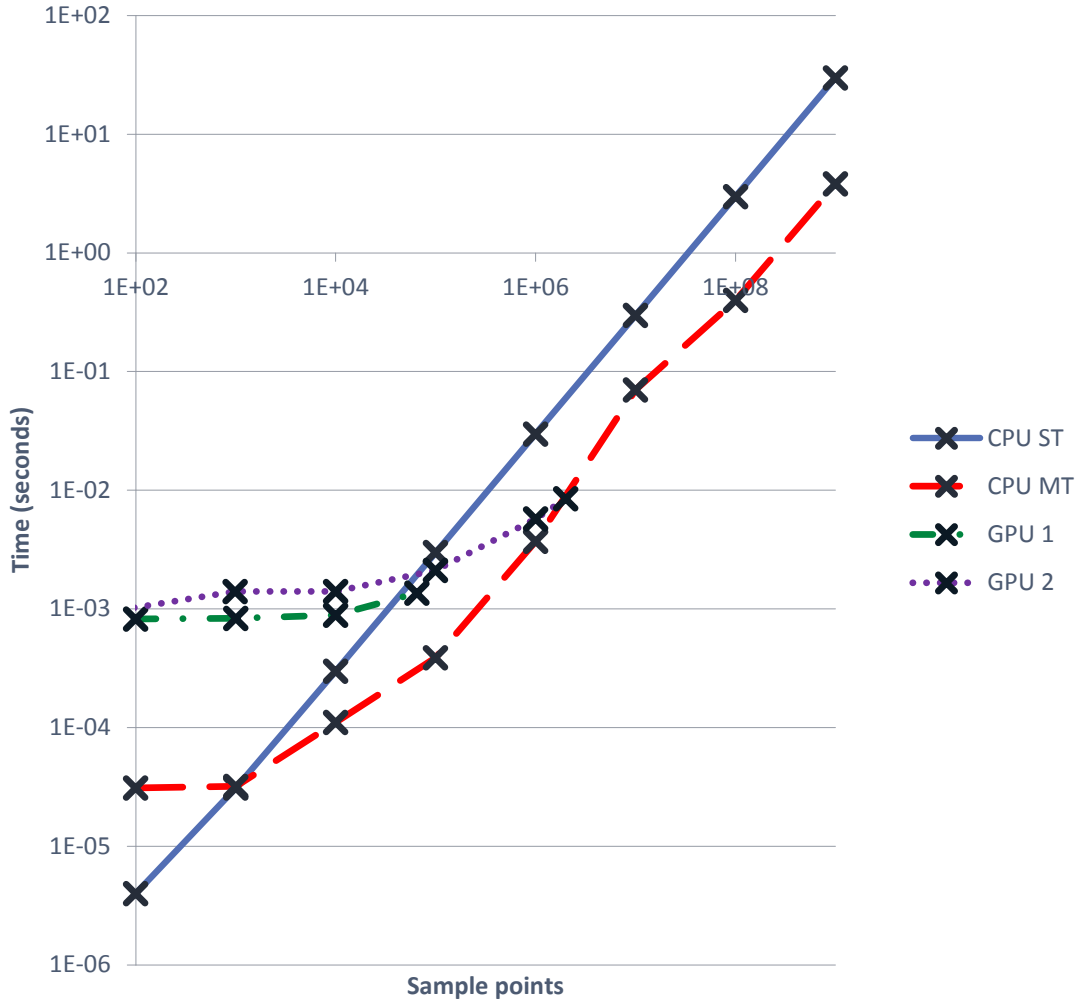
New indexing

32 threads per block

SINTEF

Parallel GPU version 2 (CUDA) 2/2

- Unable to run more than 32*65535 sample points
- Works well with 32-wide SIMD
- Able to keep up with multi-threaded version at maximum size!
- We perform roughly 16 operations per 4 bytes written (1 int): memory bound kernel!
  Optimal is 60 operations!



SINTEF

Parallel GPU version 3 (CUDA) 1/4

```
__global__ void computePiKernel3(unsigned int* output, unsigned int seed) {
    __shared__ int inside[32];

    //Generate coordinate
    //Compute radius
    …

    //Check if within circle
    if (r <= 1.0f) {
        inside[threadIdx.x] = 1;
    } else {
        inside[threadIdx.x] = 0;
    }

    … //Use shared memory reduction to find number of inside per block
```

Shared memory: a kind of “programmable cache”
We have 32 threads: One entry per thread

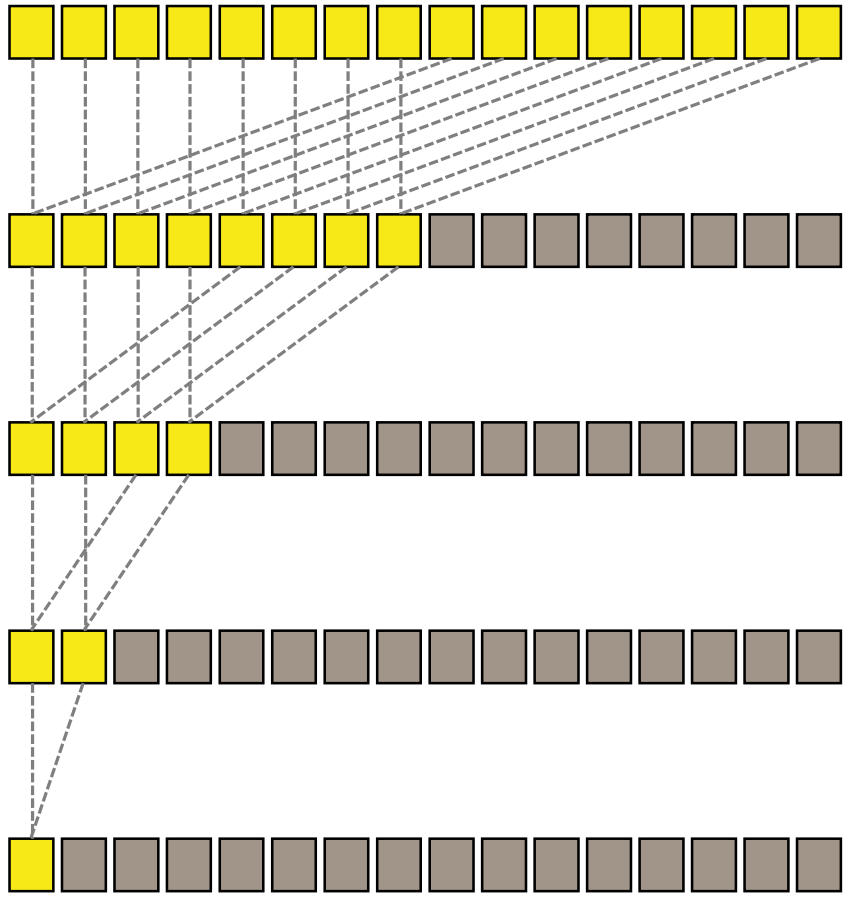
Parallel GPU version 3 (CUDA) 2/4

```
    … //Continued from previous slide

    //Use shared memory reduction to find number of inside per block
    //Remember: 32 threads is one warp, which execute synchronously
    if (threadIdx.x < 16) {
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+16];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+8];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+4];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+2];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+1];
    }

    if (threadIdx.x == 0) {
        output[blockIdx.x] = inside[threadIdx.x];
    }
}
```

Parallel GPU version 3 (CUDA) 3/4

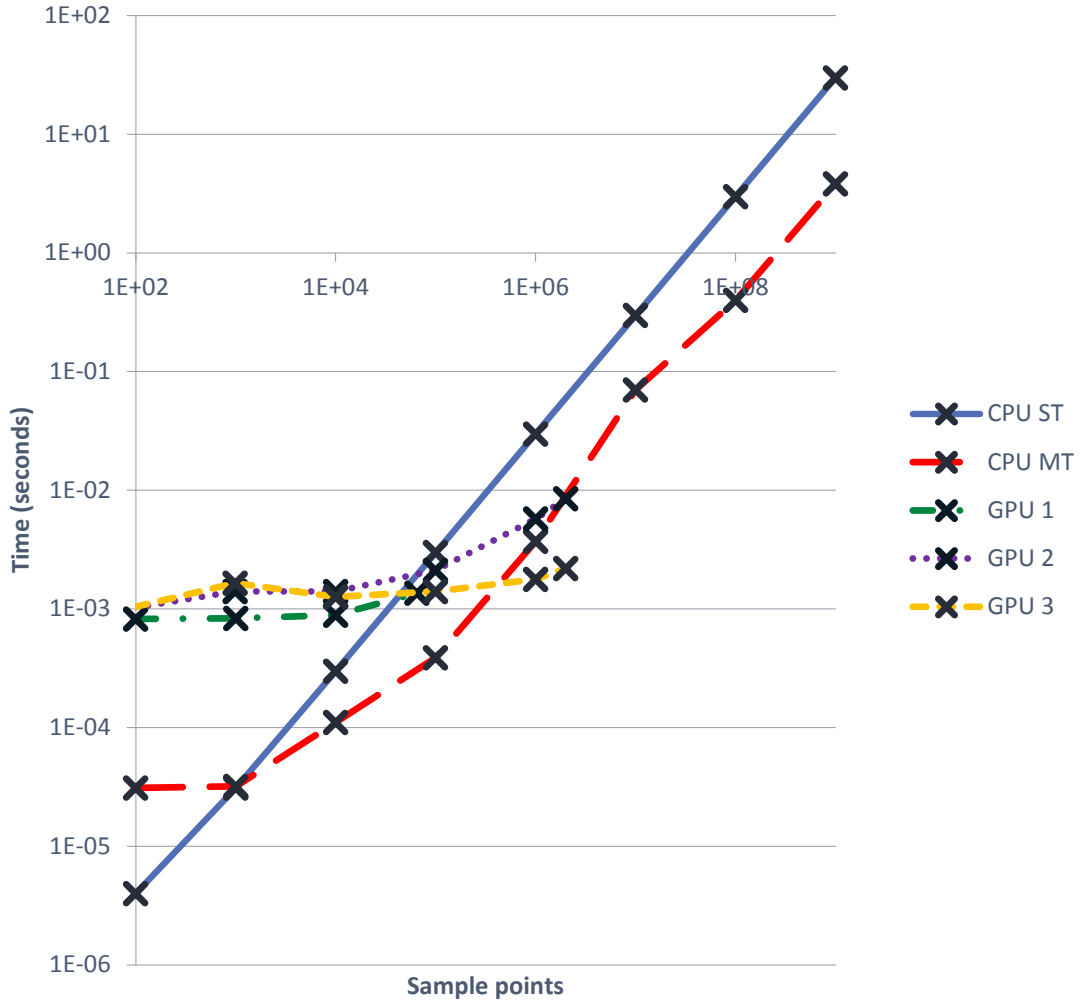
- Shared memory is a kind of programmable cache
  - Fast to access (just slightly slower than registers)
  - Programmers responsibility to move data into shared memory
  - All threads in one block can see the same shared memory
  - Often used for communication between threads

- Sum all elements in shared memory using shared memory reduction



Parallel GPU version 3 (CUDA) 4/4

- Memory bandwidth use reduced by factor 32!
- Good speed-up over multithreaded CPU!
- Maximum size is still limited to 65535*32.
- Two ways of increasing size:
  - Increase number of threads
  - Make each thread do more work



SINTEF

Parallel GPU version 4 (CUDA) 1/2

```
__global__ void computePiKernel4(unsigned int* output) {
        int n_inside = 0;

         //Shared memory: All threads can access this
        __shared__ int inside[32];
        inside[threadIdx.x] = 0;

        for (unsigned int i=0; i<iters_per_thread; ++i) {
                //Generate coordinate
                //Compute radius
                //Check if within circle
                if (r <= 1.0f) { ++inside[threadIdx.x]; }
        }

        //Communicate with other threads to find sum per block
        //Write out to main GPU memory
}
```

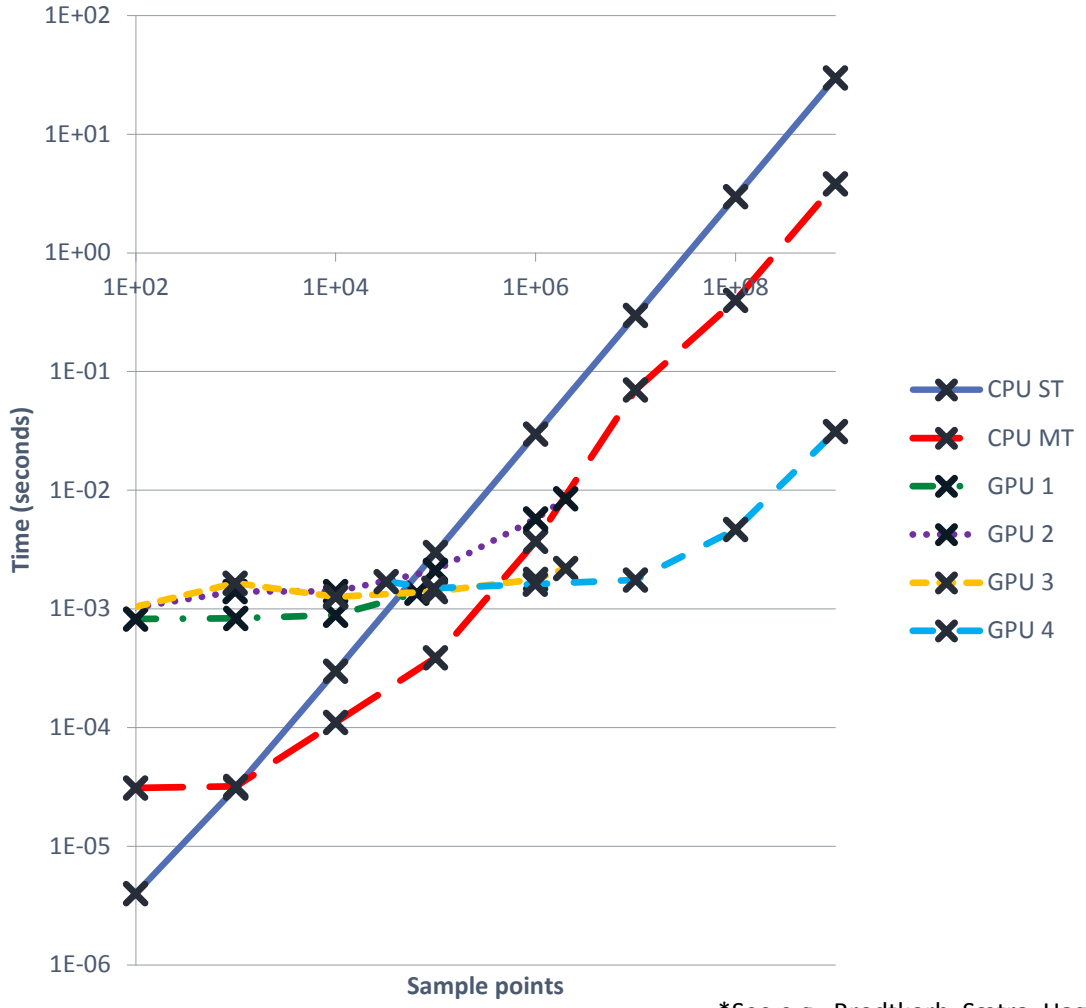
SINTEF

Parallel GPU version 4 (CUDA) 2/2

- Overheads appears to dominate runtime up-to 10.000.000 points:
  - Memory allocation
  - Kernel launch
  - Memory copy

- Estimated GFLOPS: ~450
  Thoretical peak: ~4000

- Things to investigate further:
  - Profile-driven development*!
  - Check number of threads, memory access patterns, instruction stalls, bank conflicts, ...



*See e.g., Brodtkorb, Sætra, Hagen,
GPU Programming Strategies and Trends in GPU Computing, JPDC, 2013

SINTEF

Comparing performance

- Previous slide indicates speedup of
  - 100x versus OpenMP version
  - 1000x versus single threaded version
  - Theoretical performance gap is 10x: why so fast?

- Reasons why the comparison is fair:
  - Same generation CPU (Core i7 3930K) and GPU (GTX 780)
  - Code available on Github: you can test it yourself!

- Reasons why the comparison is unfair:
  - Optimized GPU code, unoptimized CPU code.
  - I do not show how much of CPU/GPU resources I actually use (profiling)
  - I cheat with the random function (I use a simple linear congruential generator).

Summary

Summary part 1a

- All current processors are parallel:
  - You cannot ignore parallelization and expect high performance
  - Serial programs utilize 1% of potential!

- We need to design our algorithms with a specific architecture in mind
  - Data parallel, task parallel
  - Symmetric multiprocessing, heterogeneous computing

- GPUs can be programmed using many different languages
  - Cuda is the most mature
  - OpenCL is portable across hardware platforms

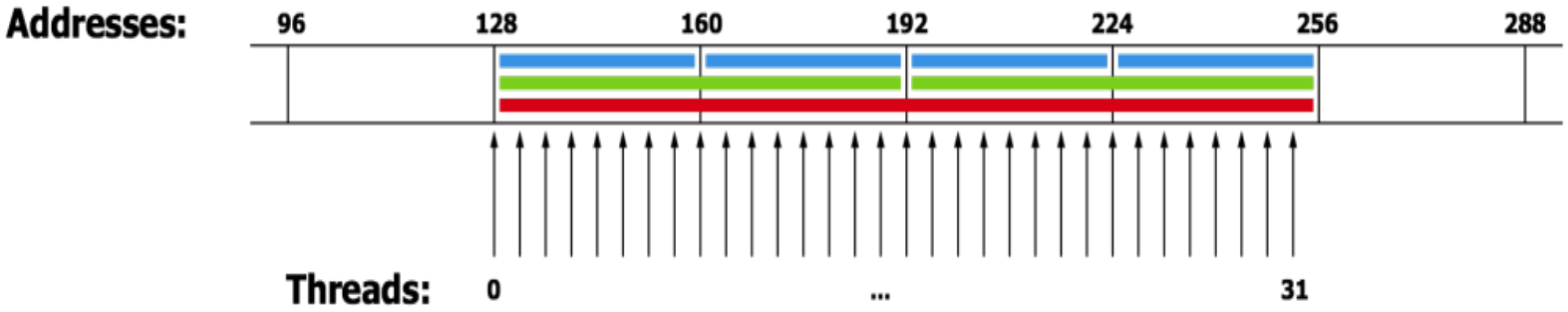
- Need to consider the hardware
  - Even for "simple" data-parallel workloads such as computing π

Bonus slides: Optimizing Memory Access

Memory access 1/2

- Accessing a single memory address triggers transfer of a full *cache line* (128 bytes)
  - The smallest unit transferrable over the memory bus
  - Identical to how CPUs transfer data

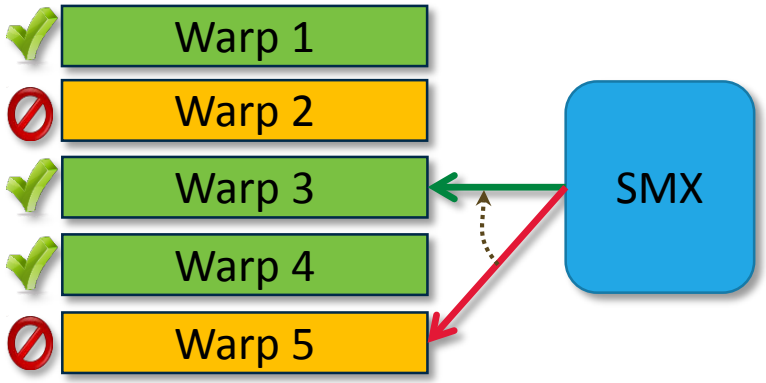
- For peak performance, 32 threads should use 32 consecutive integers/floats
  - This is referred to as coalesced reads



- On modern GPUs: Possible to transfer 32 byte segments: Better fit for random access!
- Slightly more complex in reality: see CUDA Programming Guide for full set of rules

Memory access 2/2

- GPUs have high bandwidth, and high latency
  - Latencies are on the order of hundreds to thousands of clock cycles
- Massive multithreading hides latencies
  - When one warp stalls on memory request, another warp steps in and uses execution units



- Effect: Latencies are completely hidden as long as you have enough memory parallelism:
  - More than 100 simultaneous requests for full cache lines per SM (Kepler).
  - Far more for random access!

For more details, see Paulius Micikevicius, GPU Performance Analysis and Optimization, 2013

Example: Parallel reduction

- Reduction is the operation of finding a single number from a series of numbers
  - Frequently used parallel building block in parallel computing
  - We've already used it to compute π

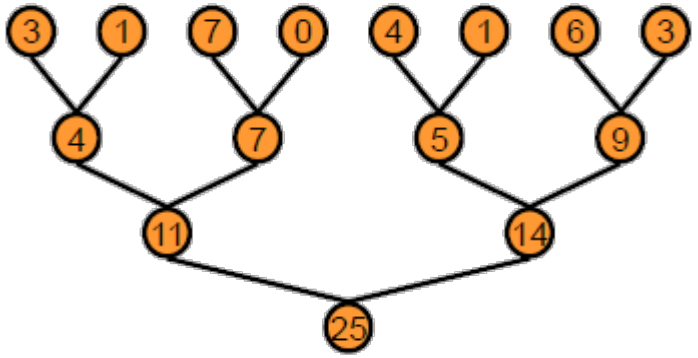
- Examples:
  - Find minimum, maximum, average, sum
  - In general: Perform a binary operation on a set data

- CPU example:

```
//Initialize to first element
T result = data[0];

//Loop through the rest of the elements
for (int i=1; i<data.size(); ++i) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

Parallel considerations

- This is a completely memory bound application
  - O(1) operation per element read and written.
  - Need to optimize for memory access!

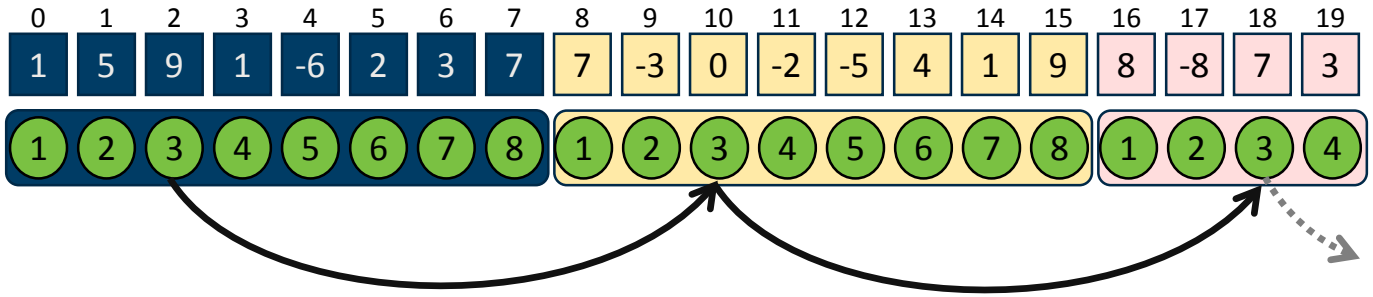


- Classical approach: represent as a binary tree
  - log2(n) passes required to reduce n elements
  - Example: 10 passes to find maximum of 1024 elements

- General idea:
  - Use few blocks with maximum number of threads (i.e., 512 in this example)
  - *Stride* through memory until all items are read
  - Perform shared memory reduction to find single largest

Example based on Mark Harris, Optimizing parallel reduction in CUDA

Striding through data




```
for (int i=threadIdx.x; i<size; i += blockDim.x) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

- Striding ensures perfect coalesced memory reads
- Thread 2 operates on elements 2, 10, 18, etc. for a block size of 8
- We have block size of 512: Thread 2 operates on elements 2, 514, 1026, …
- Perform "two-in-one" or "three-in-one" strides for more parallel memory requests

SINTEF

Shared memory reduction 1/2

- By striding through data, we efficiently reduce N/num_blocks elements to 512.

- Now the problem becomes reducing 512 elements to 1:
  lets continue the striding, but now in shared memory

- Start by reducing from 512 to 64 (notice use of __syncthreads()):

```
__syncthreads(); // Ensure all threads have reached this point

// Reduce from 512 to 256
if(tid < 256) { sdata[tid] = sdata[tid] + sdata[tid + 256]; }
__syncthreads();

// Reduce from 256 to 128
if(tid < 128) { sdata[tid] = sdata[tid] + sdata[tid + 128]; }
__syncthreads();

// Reduce from 128 to 64
if(tid < 64) { sdata[tid] = sdata[tid] + sdata[tid +  64]; }
__syncthreads();
```

Shared memory reduction 2/2

- When we have 64 elements, we can use 32 threads to perform the final reductions
- Remember that 32 threads is one warp, and execute instructions in SIMD fashion
- This means we do not need the syncthreads:

```
if (tid < 32) {
        volatile T *smem = sdata;
        smem[tid] = smem[tid] + smem[tid + 32];
        smem[tid] = smem[tid] + smem[tid + 16];
        smem[tid] = smem[tid] + smem[tid + 8];
        smem[tid] = smem[tid] + smem[tid + 4];
        smem[tid] = smem[tid] + smem[tid + 2];
        smem[tid] = smem[tid] + smem[tid + 1];
}

if (tid == 0) {
        global_data[blockIdx.x] = sdata[0];
}
```

- Volatile basically tells the optimizer "off-limits!"
- Enables us to safely skip __syncthreads()

SINTEF