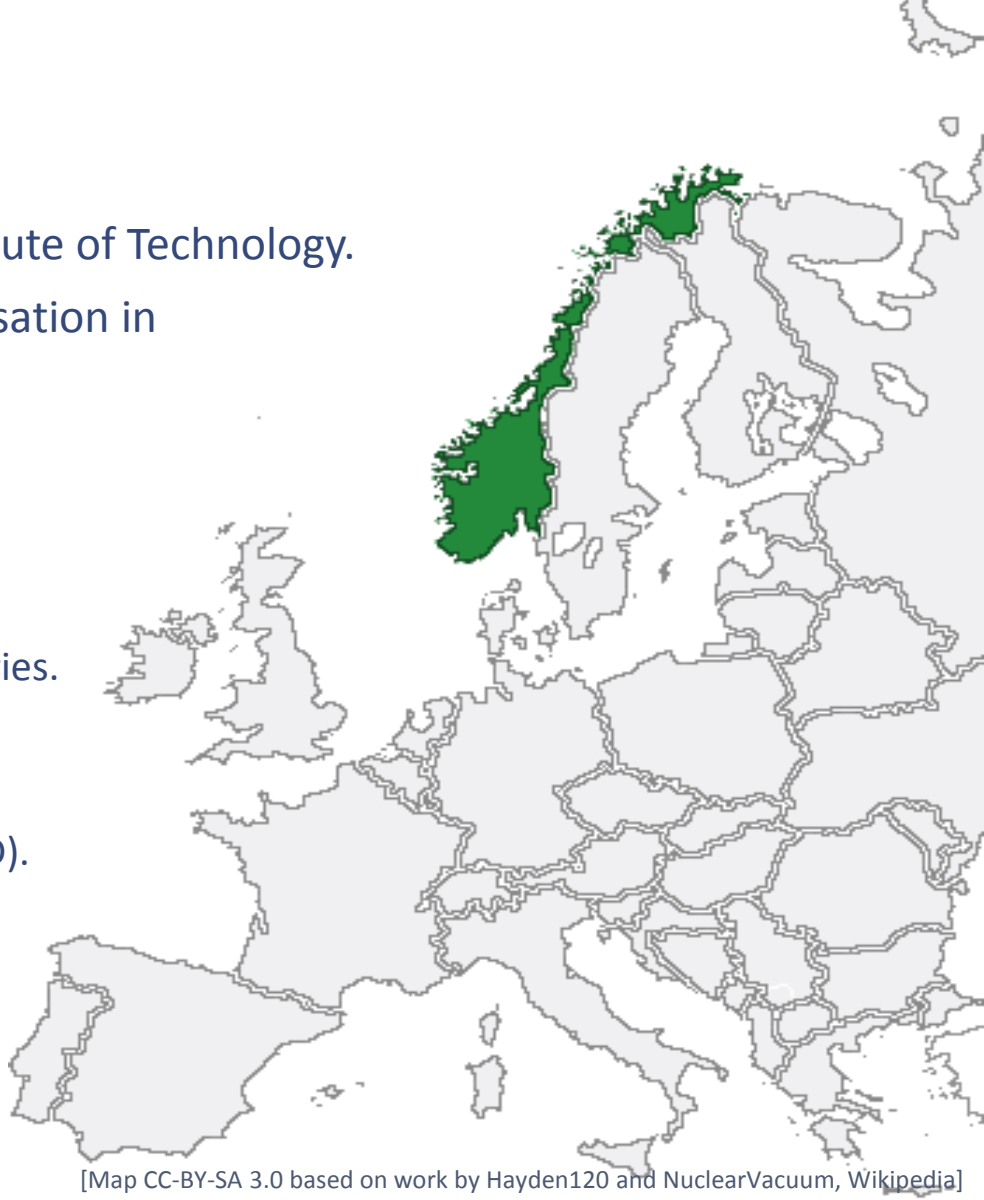
Short introduction to GPU and Heterogeneous Computing

University of Málaga, 2016-04-11

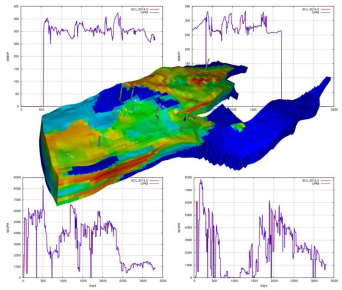
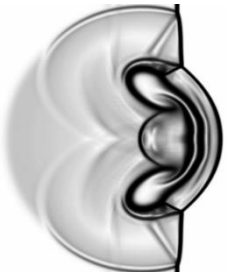
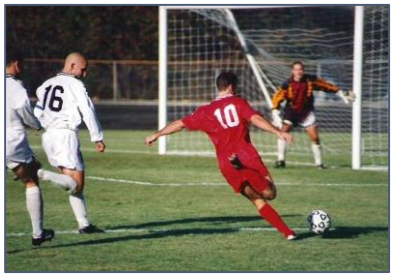
André R. Brodtkorb, SINTEF, Norway

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.



The Department of Applied Mathematics

- **Numerical simulation** of oil reservoirs, tsunamis, flooding, oil production, wind turbines, airport/aircraft turbulence, …
- **Discrete Optimization,** Scheduling of football matches (e.g., the national league in Norway), route planning, hospital planning, train planning, …
- **CAD Technology** and splines for geometry processing and design of bikes, airplanes, …
- **Computing**, Big data, cloud computing, GPU computing…

Applied Mathematics

$$\frac{\partial u}{\partial t} - fv = -g\frac{\partial h}{\partial x} - bu,$$
$$\frac{\partial v}{\partial t} + fu = -g\frac{\partial h}{\partial y} - bv,$$
$$\frac{\partial h}{\partial t} = -H\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)$$

- We "translate" a complex mathematical problem to a problem a computer can help us solve
- We must ensure the translation is both correct and fast
  - Tomorrows weather is no good in two days time!
  - The wrong answer is no good no matter how fast it arrives!

Computational Geosciences

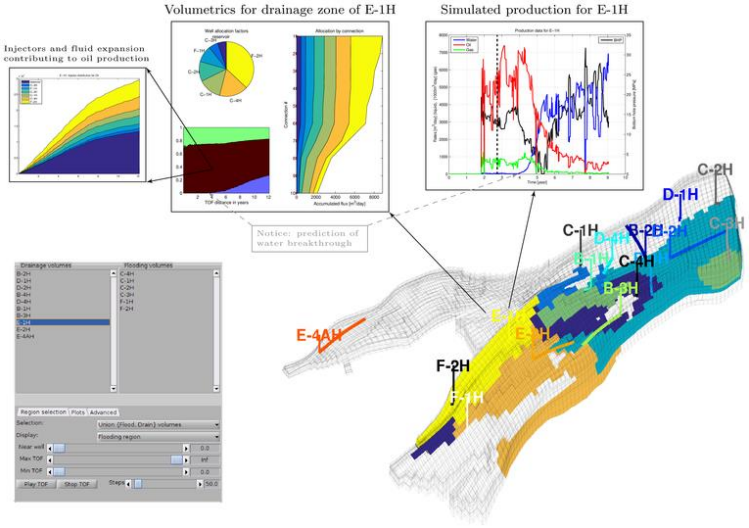
Part av **SINTEF IKT**,

Department of Applied Mathematics

Research Manager is Knut-Andreas Lie

We have research projects financed by

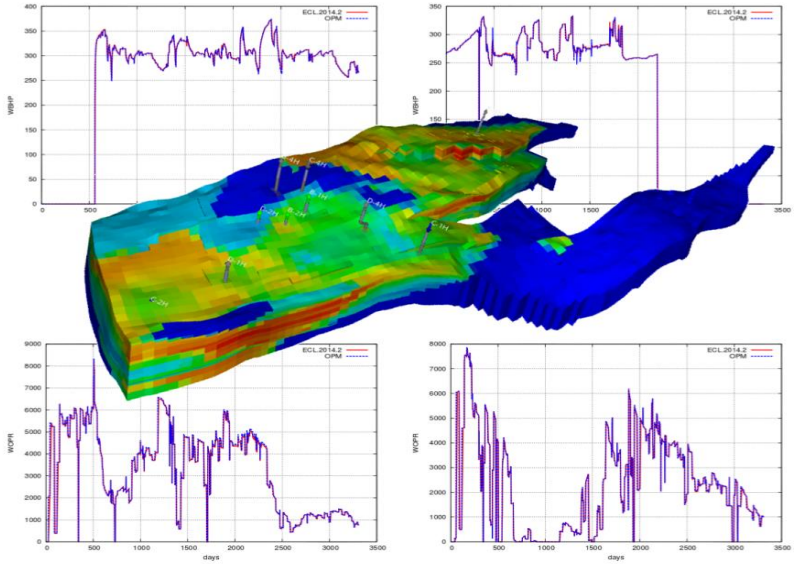
- Norwegian and foreign industry
  (E.g., Statoil, ExxonMobil, Schlumber
- The Research Council of Norway and GassNova



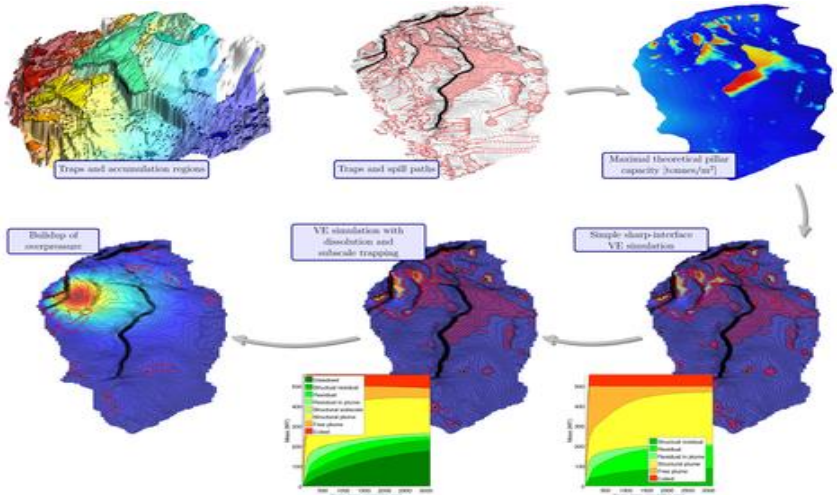
Most of what we create ends up in open software

- Open Porous Media (OPM)
- MATLAB Reservoir Simulation Toolbox (MRST)

Examples of current research topics

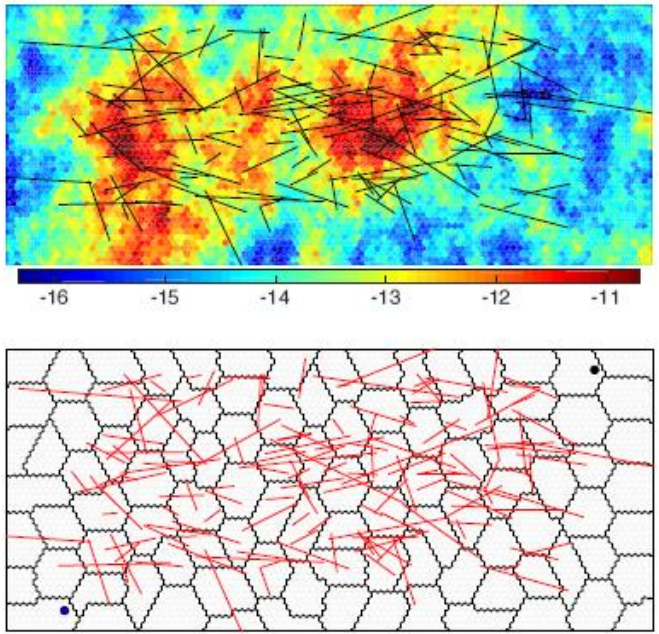


Simulation of oil and gas
in subsurface reservoirs

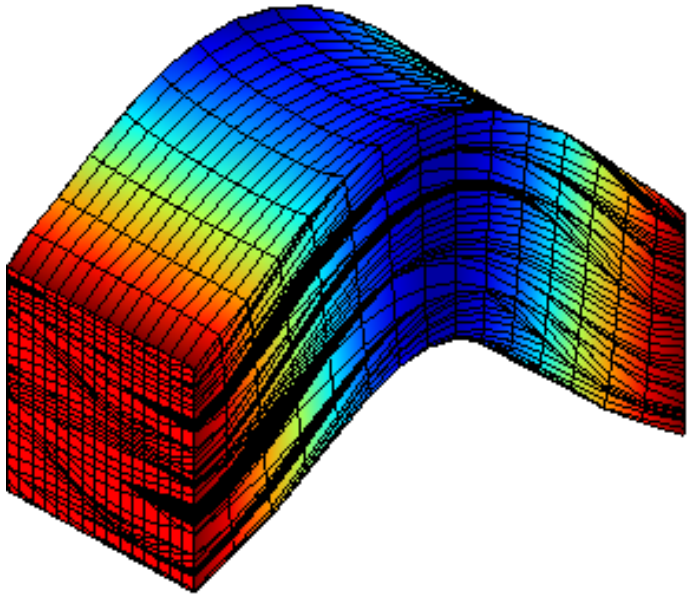


Injection of CO2 into
deep subsurface reservoirs

Examples of current research topics

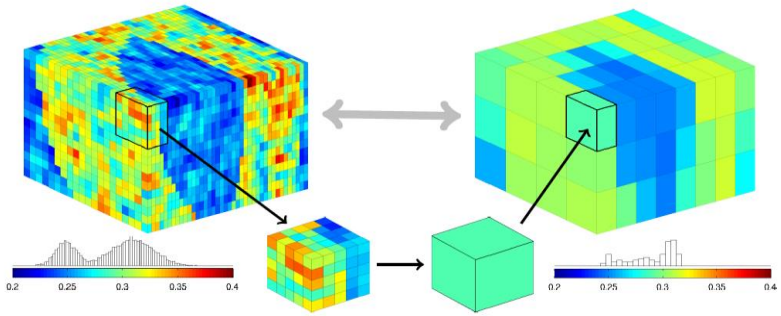


Flow in porous media with faults

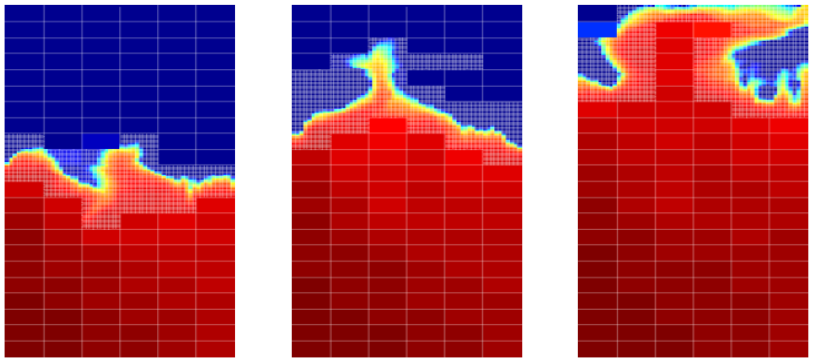


Combined geomechanics and flow physics

Examples of current research topics



Multiscale methods



Advanced methods for solving partial differential equations

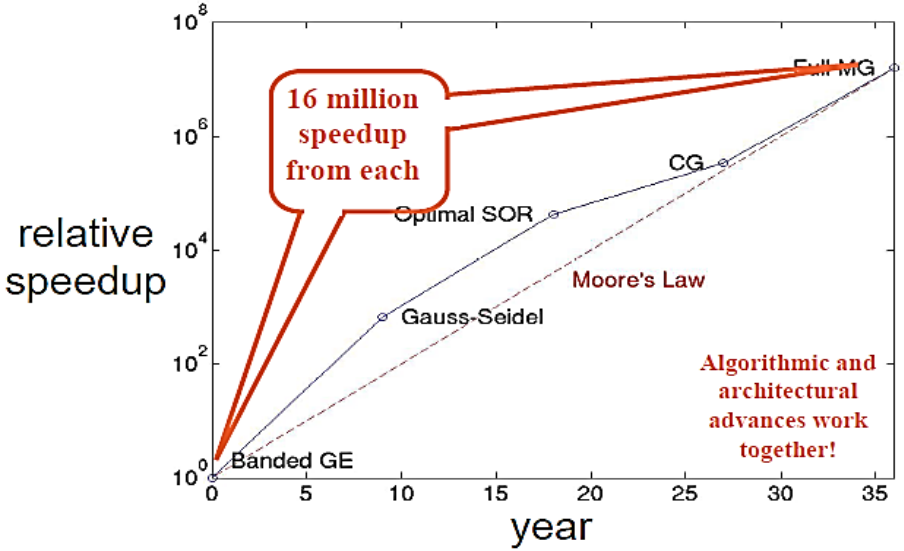
Outline

- Part 1:
  - Motivation for going parallel
  - Multi- and many-core architectures
  - Parallel algorithm design
- Part 2
  - Example: Computing π on the GPU
  - Optimizing memory access

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.
- Our aim is to equip you with some key insights on how to design algorithms for todays and tomorrows parallel architectures.



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

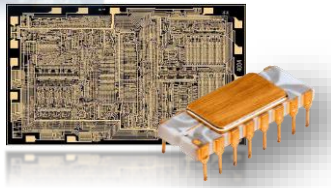
1947: Transistor

(Shockley, Bardeen, and Brattain)

2000

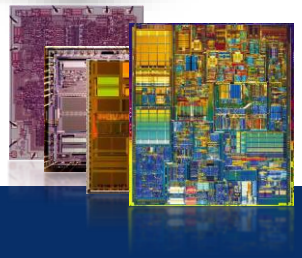
1958: Integrated Circuit

(Kilby)



1971: Microprocessor

(Hoff, Faggin, Mazor)



1971- Exponential growth

(Moore, 1965)

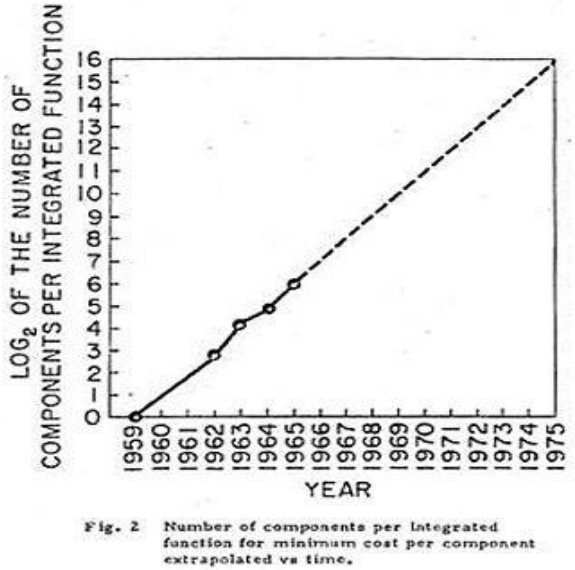
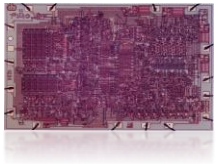


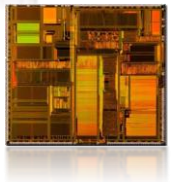
Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

History lesson: development of the microprocessor 2/2

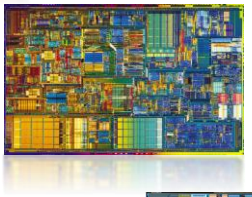


1971: 4004,
2300 trans, 740 KHz

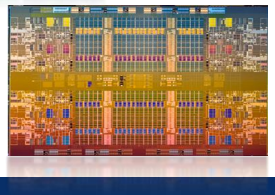
1982: 80286,
134 thousand trans, 8 MHz



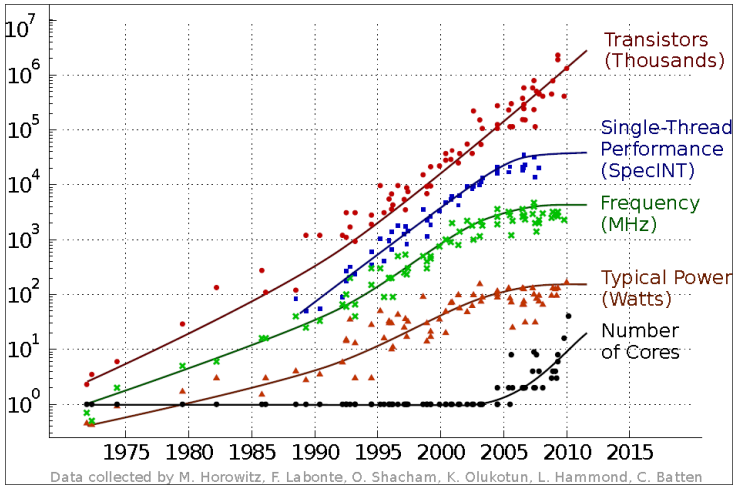
1993: Pentium P5,
1.18 mill. trans, 66 MHz



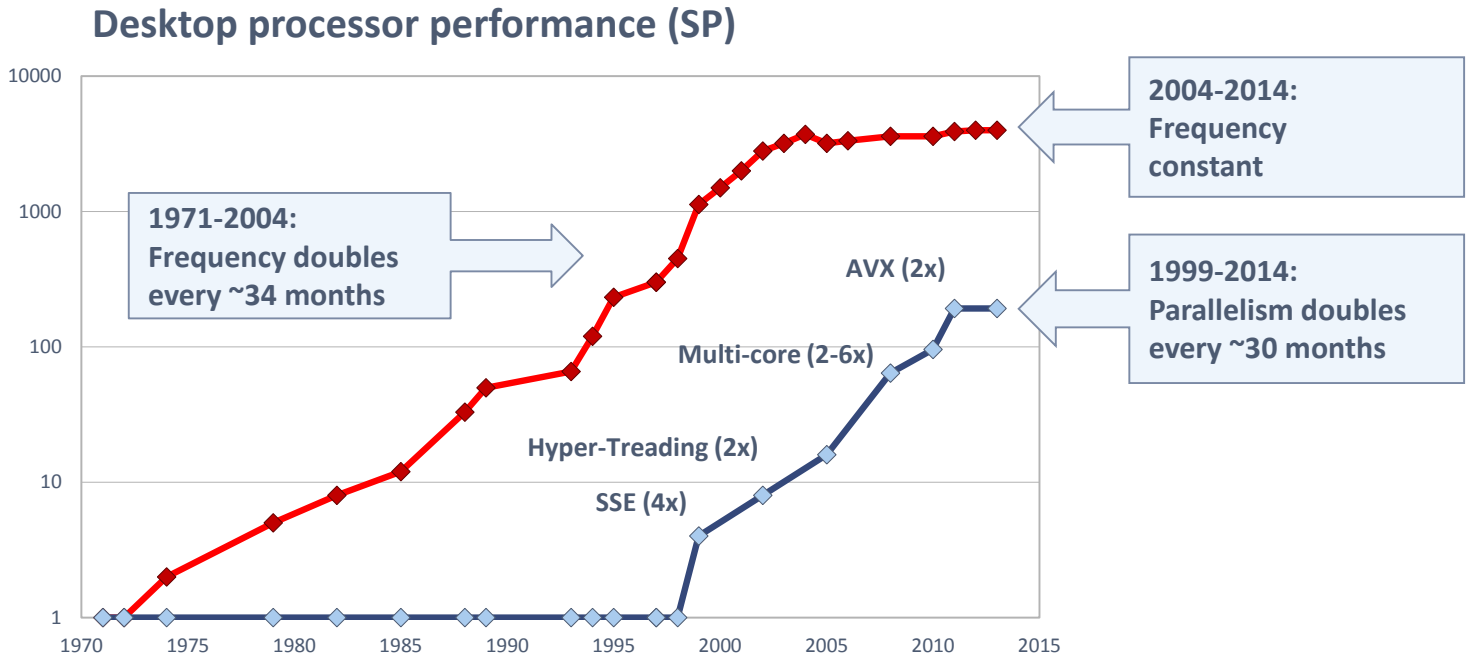
2000: Pentium 4,
42 mill. trans, 1.5 GHz



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



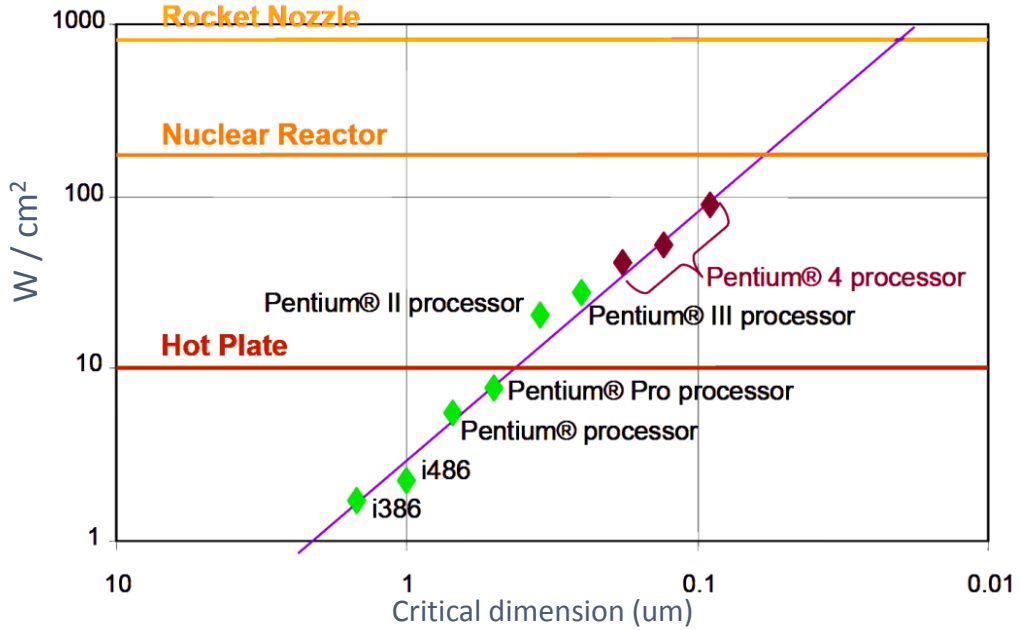
End of frequency scaling



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

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!

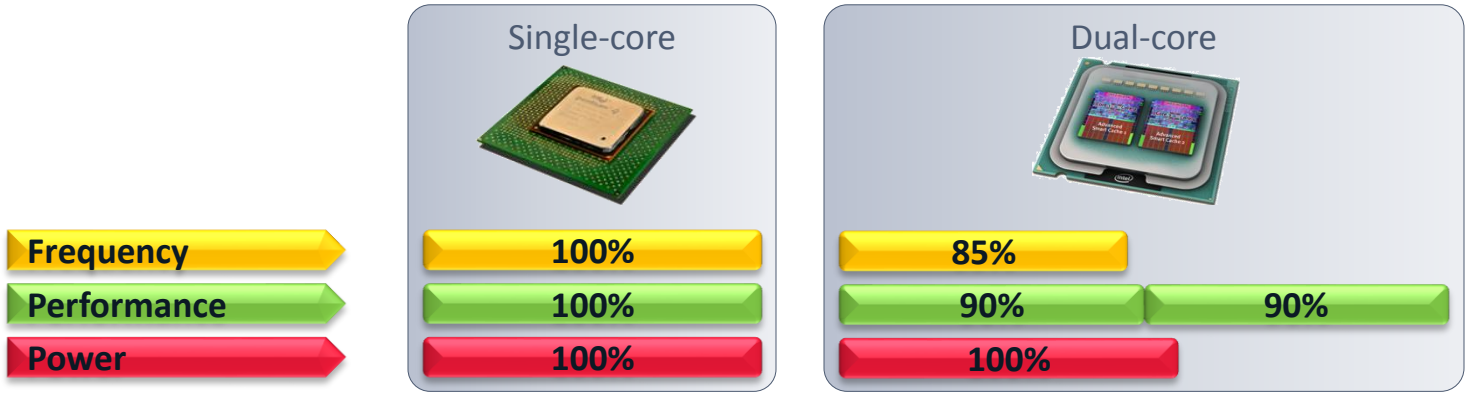


Original graph by G. Taylor, "Energy Efficient Circuit Design and the Future of Power Delivery" EPEPS'09

Why Parallelism?

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

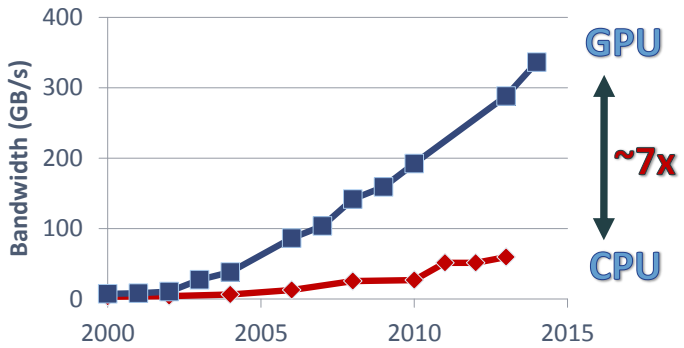
$$P_d \propto f^3$$

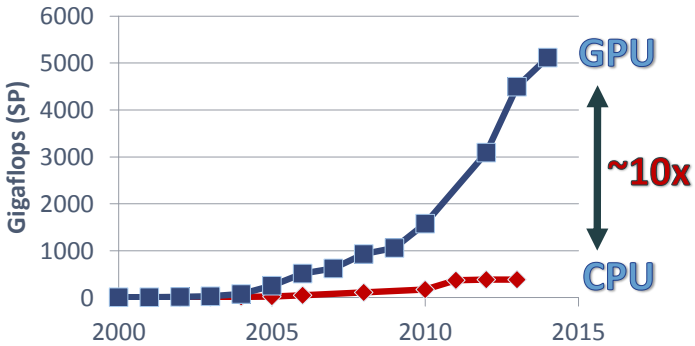


[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

Massive Parallelism: The Graphics Processing Unit

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

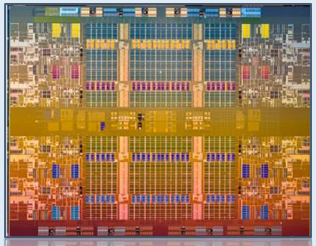




Multi- and many-core architectures

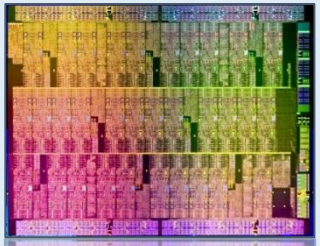
Multi- and many-core processor designs

- 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

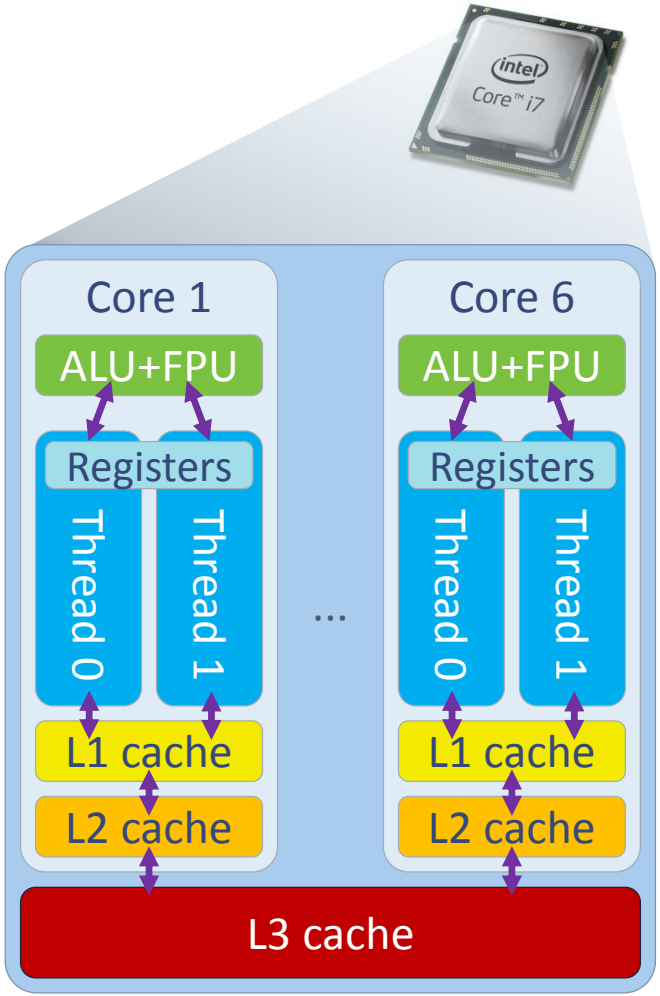


Accelerators:
GPUs, Xeon Phi

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!

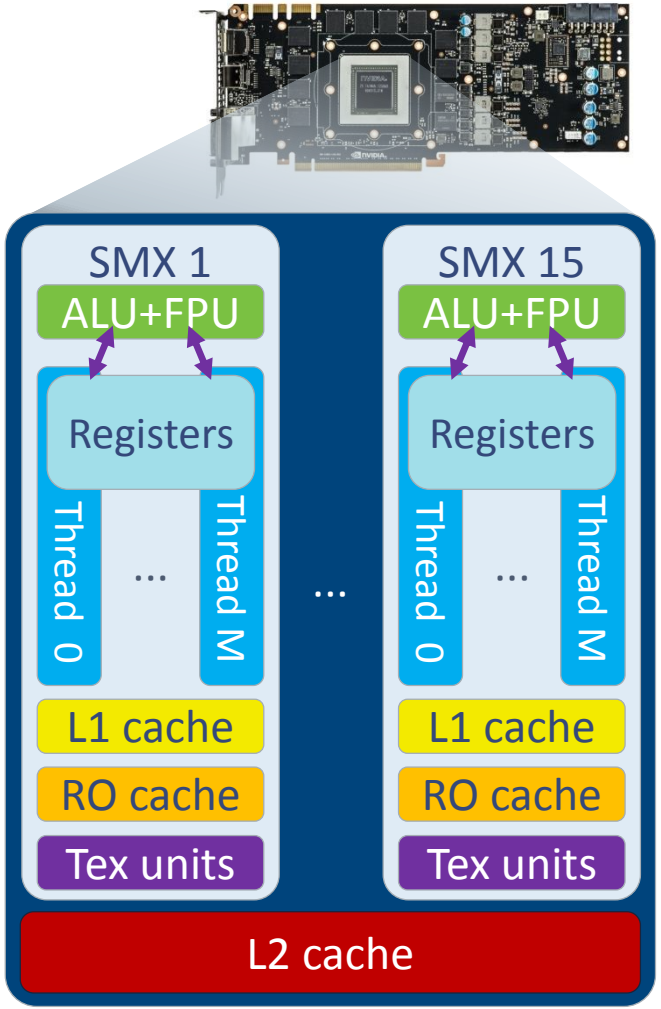


Simplified schematic of CPU design

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

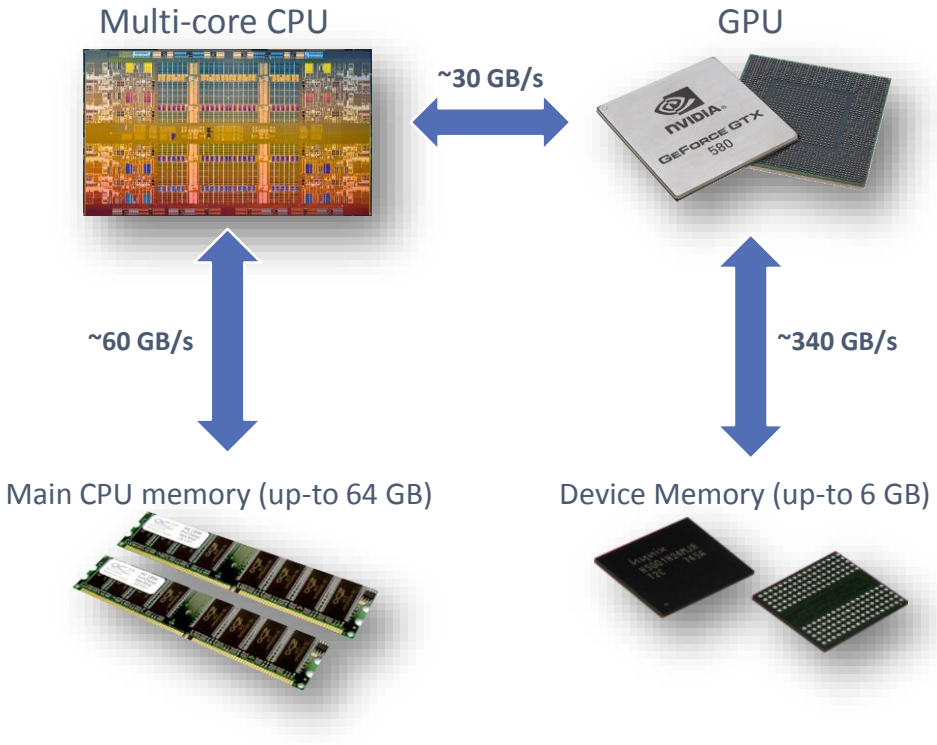
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: Six SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!



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

Parallel computing

- 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 chefs?
  - Let's look at one example: computing π

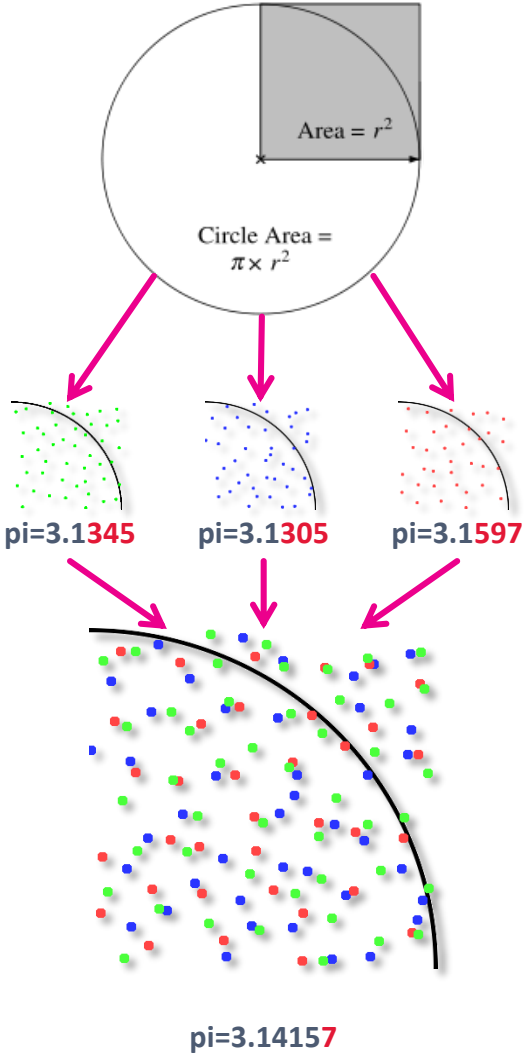
Picture: Daily Mail Reporter , www.dailymail.co.uk

Estimating π (3.14159...) in parallel

- 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

- This can be referred to as a data-parallel workload: All processors perform the same operation.



Disclaimer: this is a naïve way of calculating PI, only used as an example of parallel execution

Data parallel workloads

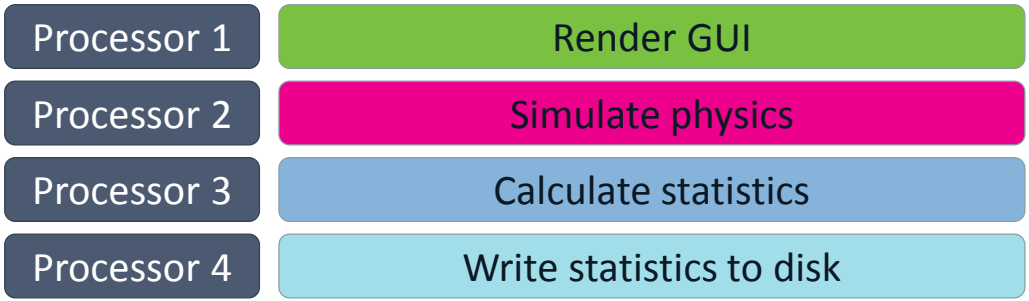
- Data parallelism performs the same operation
  for a set of different input data

- 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

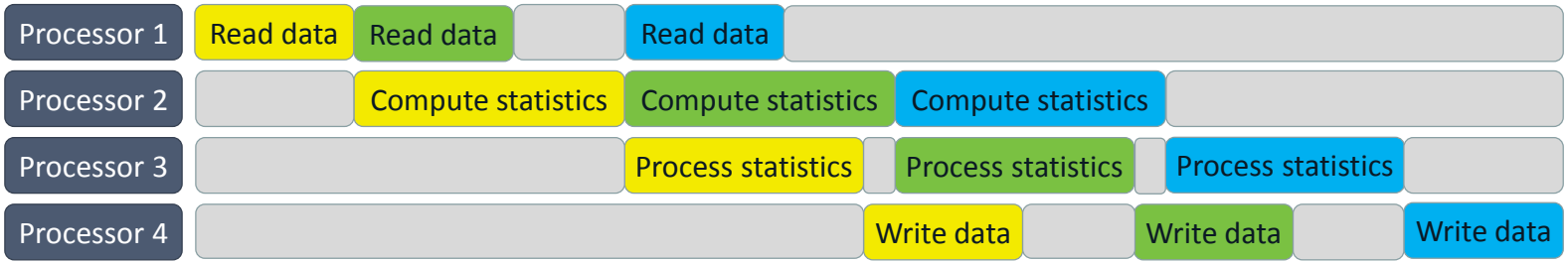
- Task parallelism divides a problem into subtasks which can be solved individually
- Scales well for a large number of tasks:
  The more parallel tasks, the more processors you can use
- Example: A simulation application:



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

Task parallel workloads 2/3

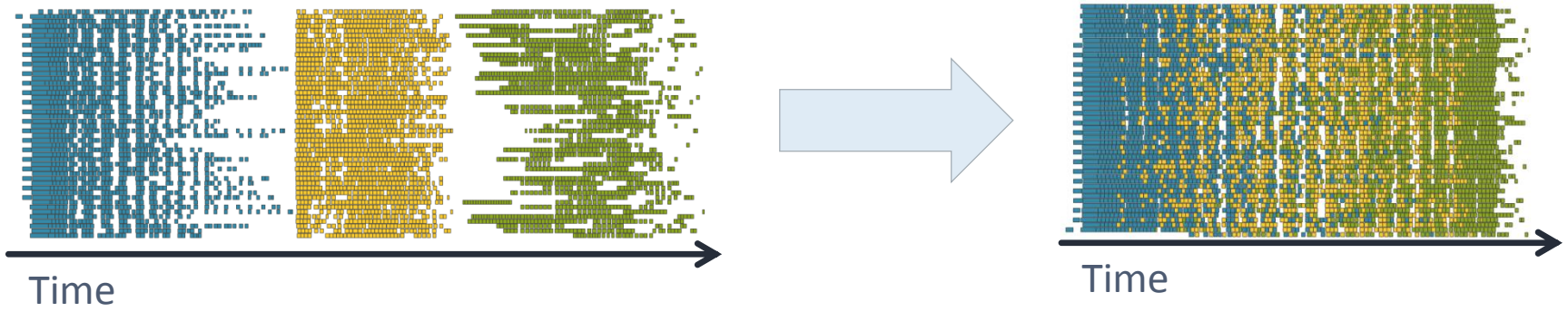
- Another way of using task parallelism is
  to execute dependent tasks on different processors
- Scales well with a large number of tasks, but performance limited by slowest stage
- Example: Pipelining dependent operations



- 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



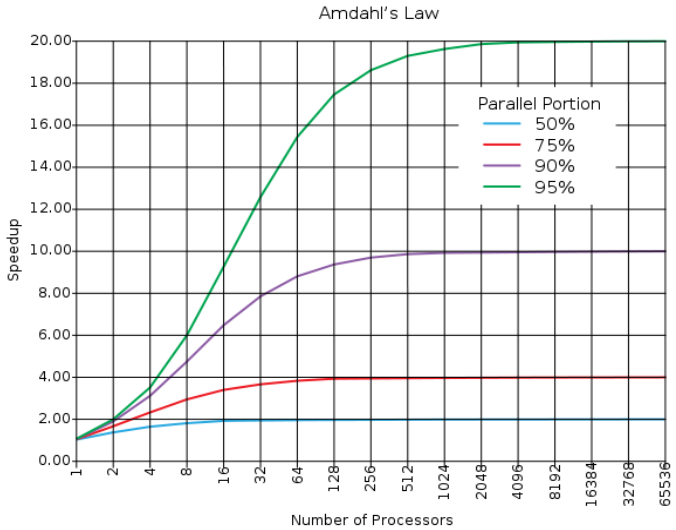
- "Gray boxes" are minimized

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!



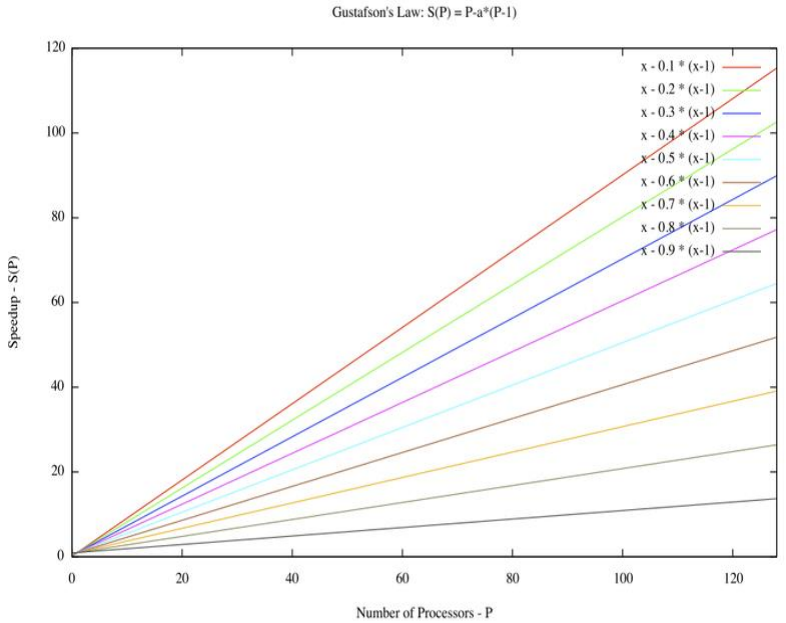
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

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

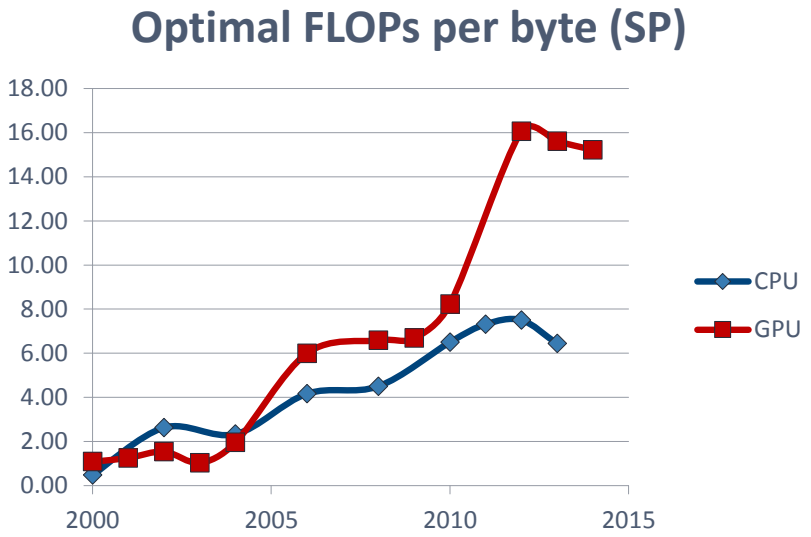
- Moving data has become the major bottleneck in computing.
- Downloading 1GB from Japan to Switzerland consumes roughly the energy of 1 charcoal briquette[1].
- A FLOP costs less than moving one byte[2].
- Key insight: flops are free, moving data is expensive

[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

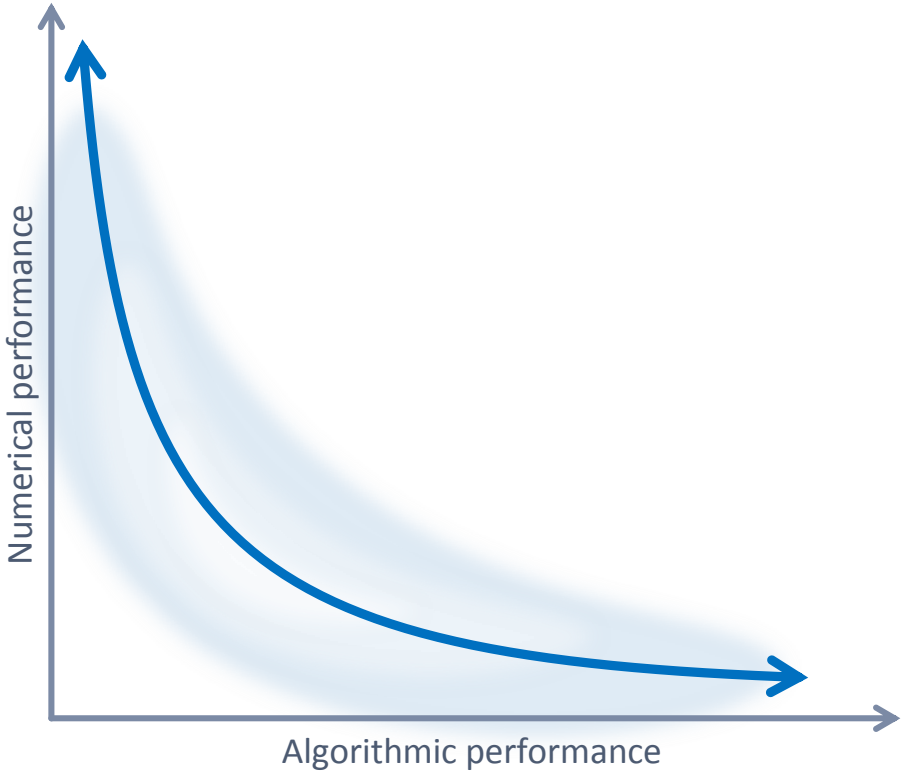
Limits on performance 4/4

- 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

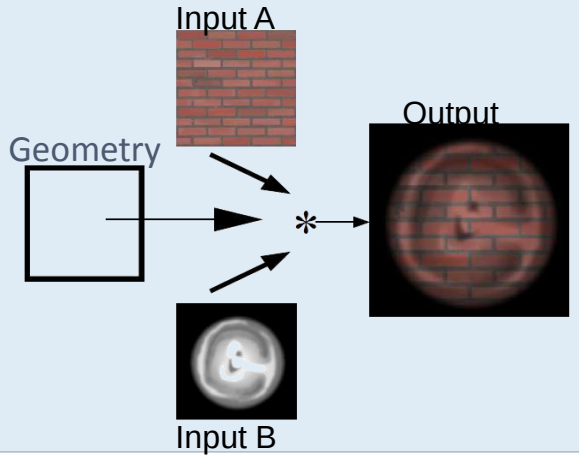


Programming GPUs

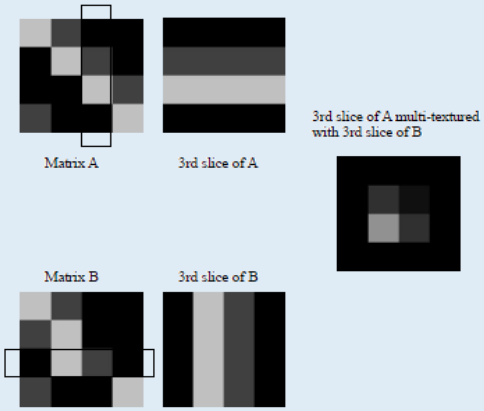
Early Programming of GPUs

- 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

Element-wise matrix multiplication

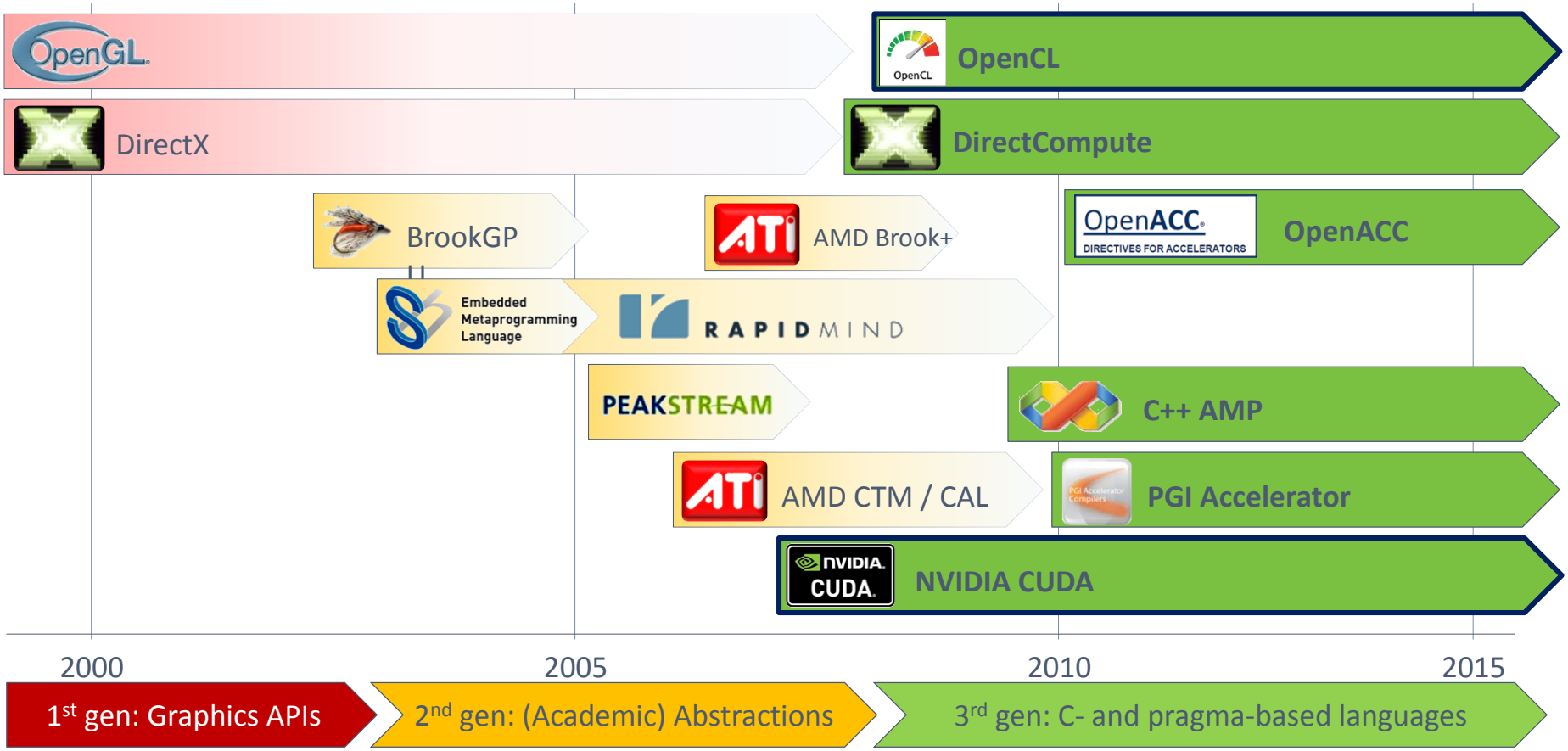


Matrix multiplication



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

GPU Programming Languages



Computing with CUDA

- We will focus on CUDA, as it 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.

- For high-level development, languages like OpenACC (pragma based) or C++ AMP (extension to C++) exist
  - Typicall works well for toy problems, and not so 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

```
__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 addFunctionGPU(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

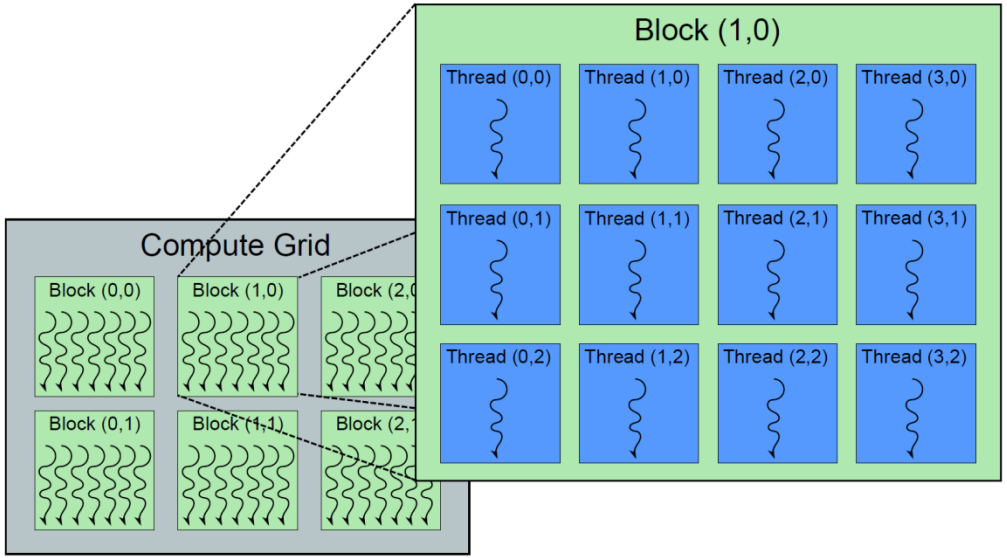
Implicit double for loop
for (int blockIdx.x = 0;
blockIdx.x < grid.x;
blockIdx.x) { …

Run on GPU

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

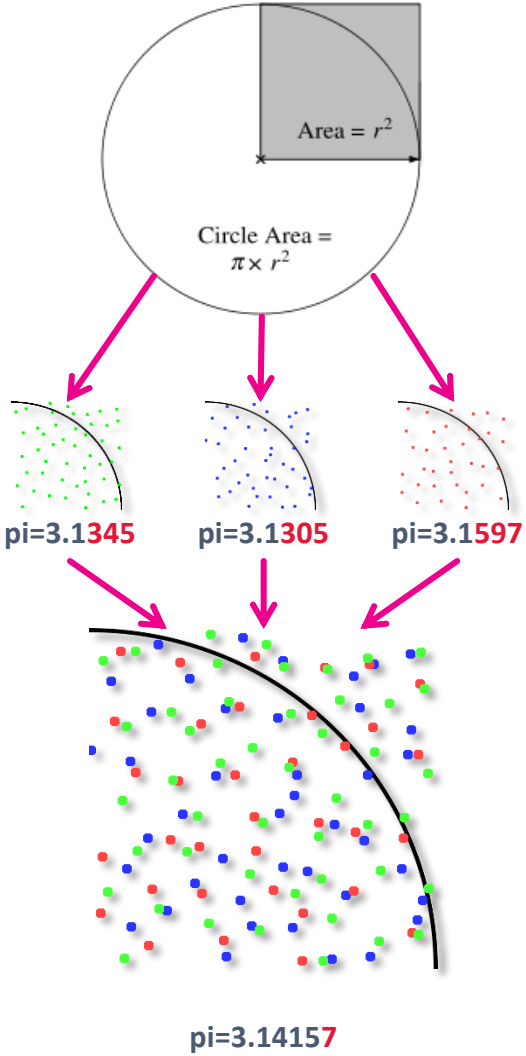
- Global synchronization and communication is only possible between kernel launches
  - Really expensive, and should be avoided if possible



Example: Computing π with CUDA

- 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☺



Serial CPU code (C/C++)

```
float computePi(int n_points) {
        int n_inside = 0;
        for (int i=0; i<n_points; ++i) {
            //Generate coordinate                                   1
            float x = generateRandomNumber();
            float y = generateRandomNumber();
            //Compute distance                                      2 & 3
            float r = sqrt(x*x + y*y);
            //Check if within circle
            if (r < 1.0f) { ++n_inside; }
        }
        //Estimate Pi                                               4
        float pi = 4.0f * n_inside / static_cast<float>(n_points);
        return pi;
}
```

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

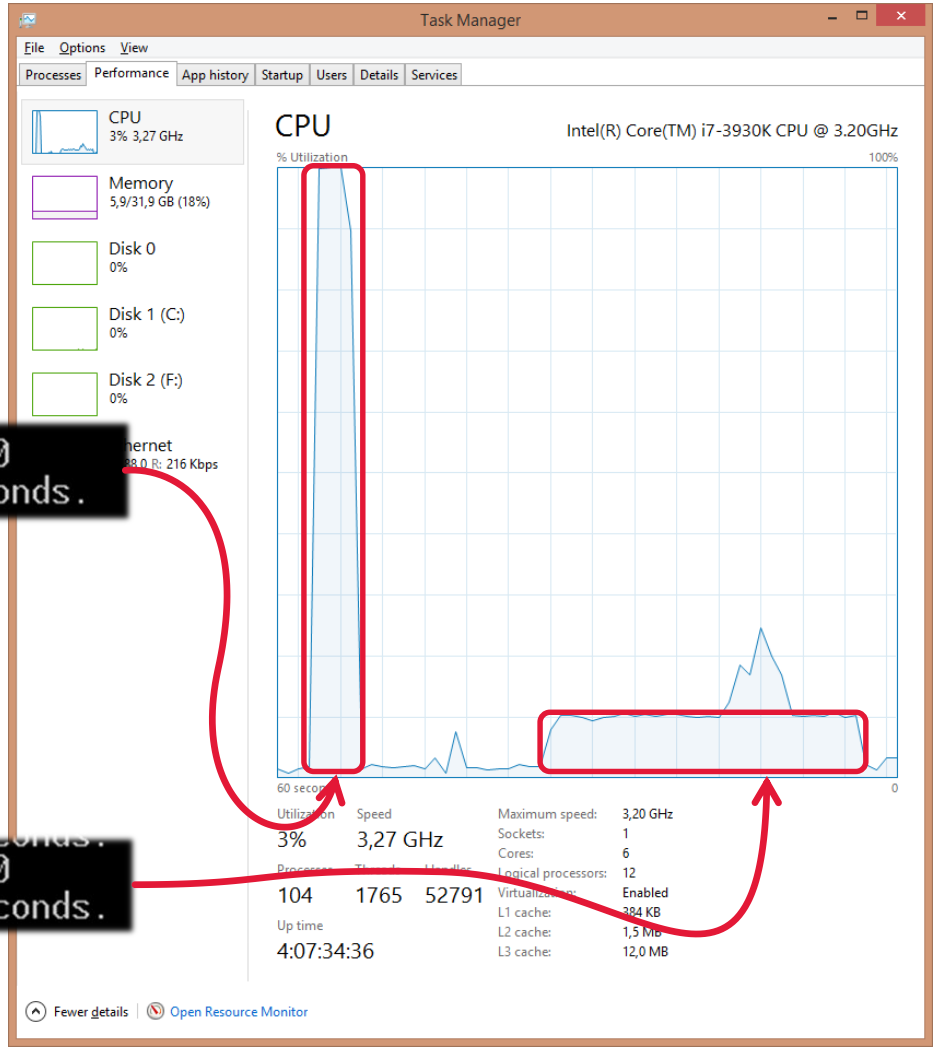
Performance

- 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

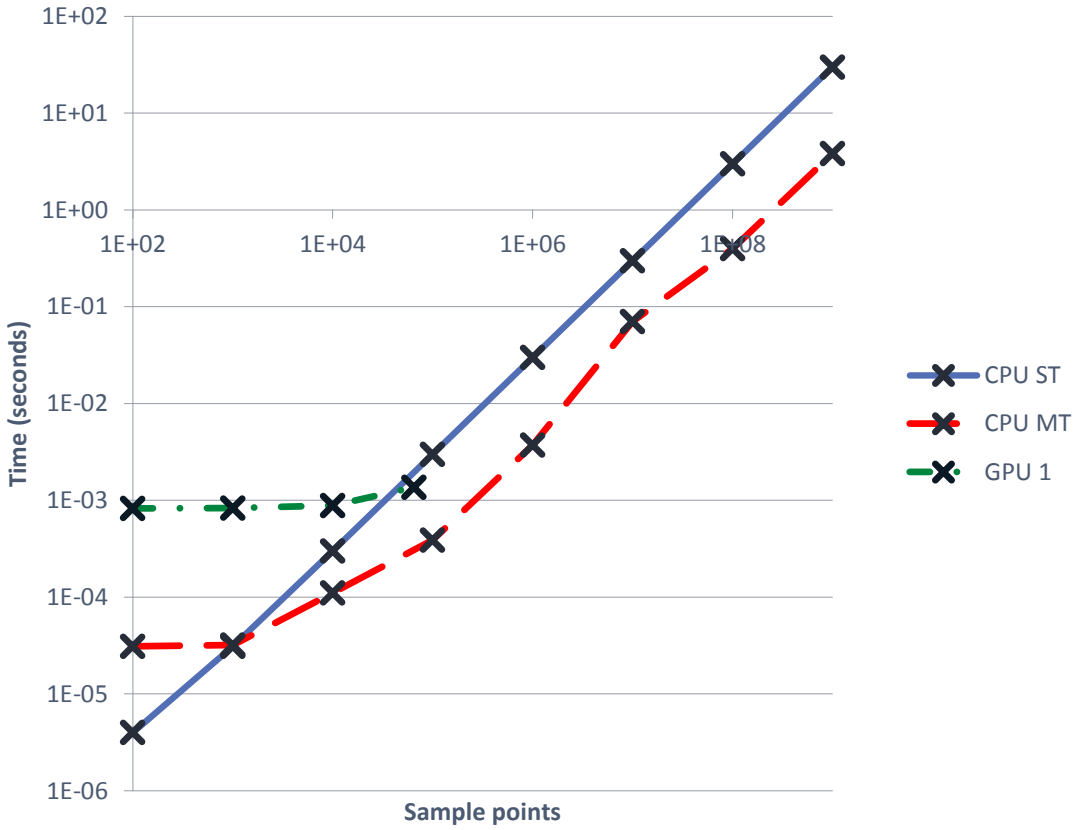
*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

Parallel GPU version 1 (CUDA) 2/3

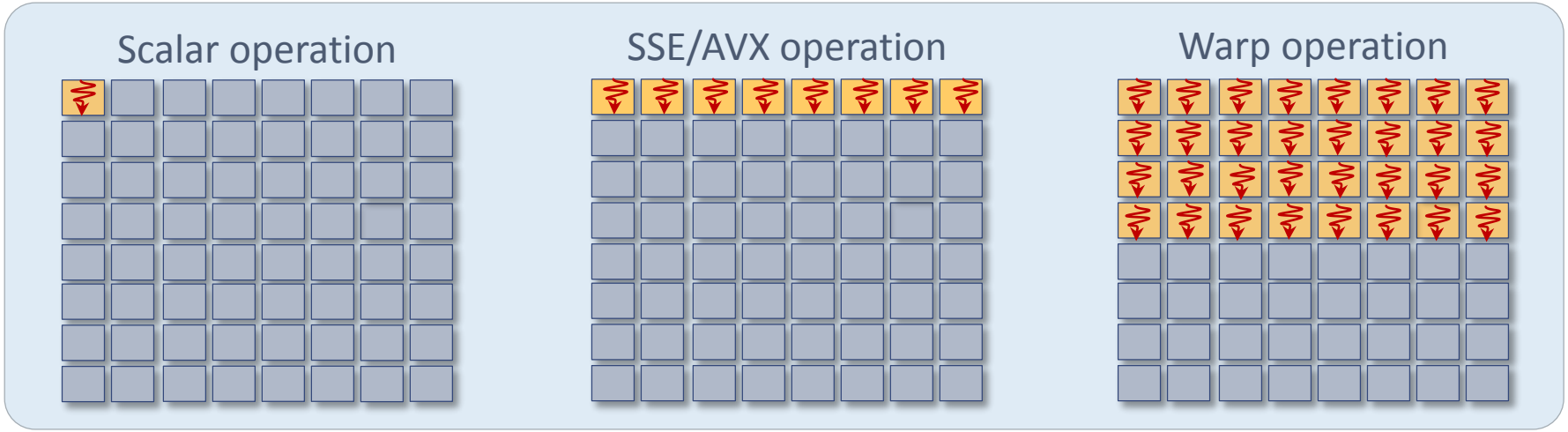
```
float computePi(int n_points) {
      dim3 grid = dim3(n_points, 1, 1);
      dim3 block = dim3(1, 1, 1);

      //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;
}
```

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.

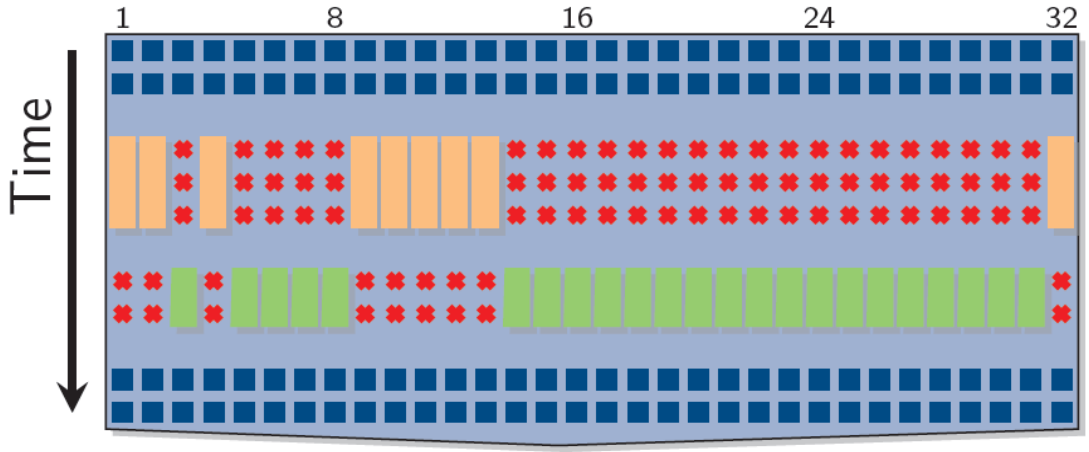


GPU Vector Execution Model



- **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;
        }
}
```

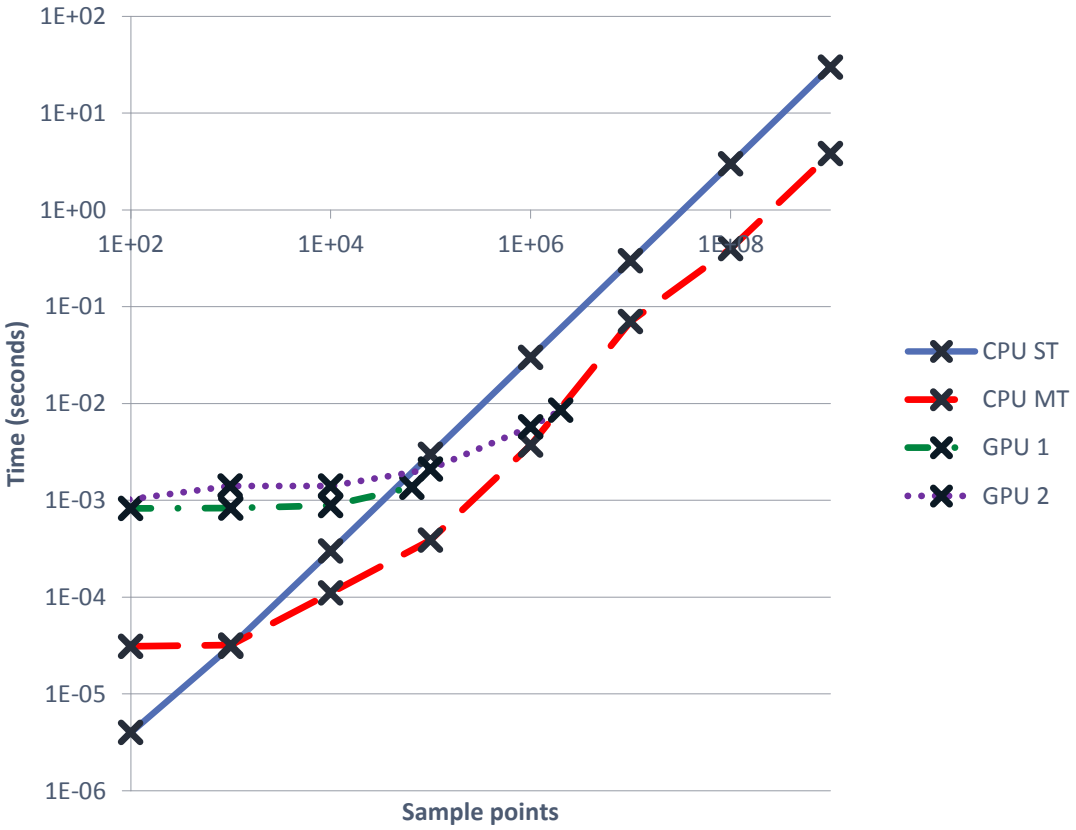
New indexing

```
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);
        …
}
```

32 threads per block

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!



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
```

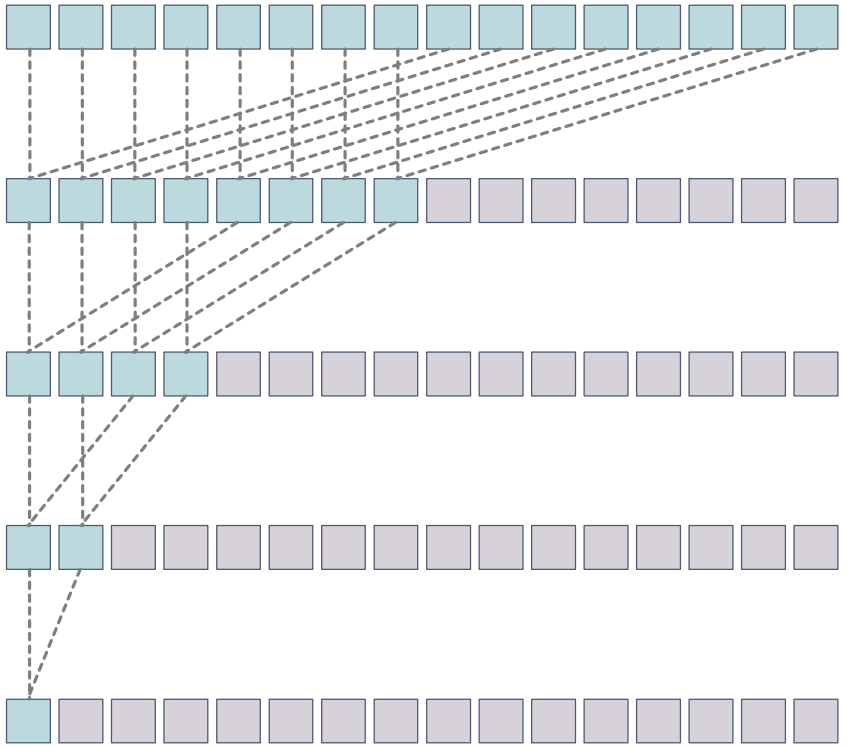
Shared memory: a kind of “programmable cache”
We have 32 threads: One entry per thread

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];
}
}
```

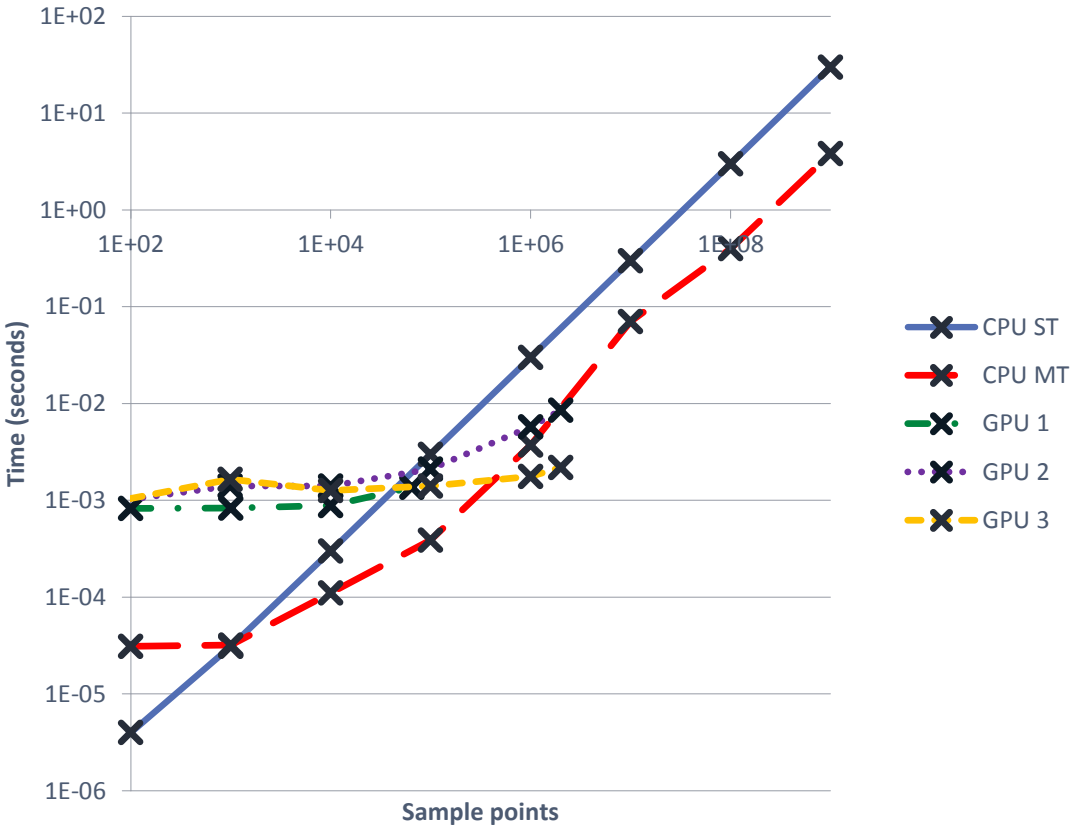
Parallel GPU version 3 (CUDA) 3/4

- 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

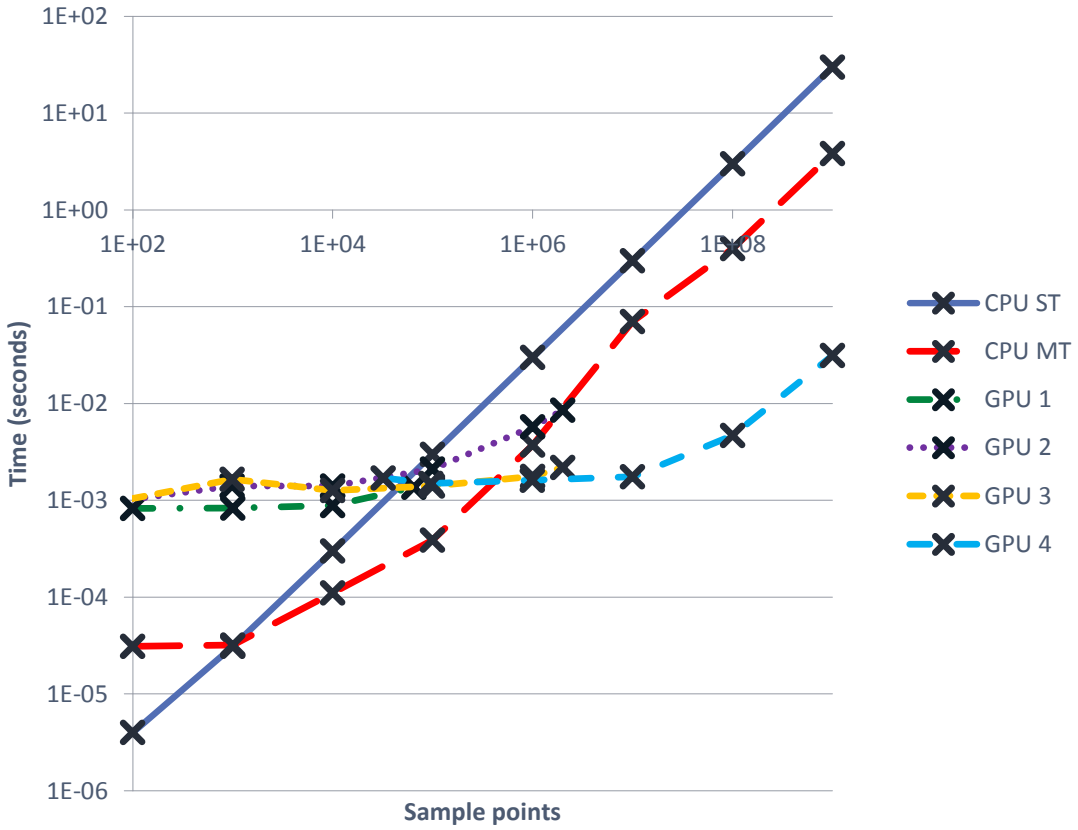


Parallel GPU version 4 (CUDA) 1/2

```
__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
}
```

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

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).

Recap of tutorial

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

- Getting started coding for GPUs has never been easier:
  - Nvidia CUDA tightly integrated into Visual Studio
  - Excellent profiling tools available with toolkit

- Low hanging fruit has been picked:
  - The challenge now is to devise new intelligent algorithms that take the architecture into consideration

Some references

- Code examples available online: http://github.com/babrodtk
- NVIDIA CUDA website: https://developer.nvidia.com/cuda-zone
- Brodtkorb, Hagen, Schulz and Hasle, **GPU Computing in Discrete Optimization Part I: Introduction to the GPU,** EURO Journal on Transportation and Logistics, 2013.
- Schulz, Hasle, Brodtkorb, and Hagen, **GPU Computing in Discrete Optimization Part II: Survey Focused on Routing Problems,** EURO Journal on Transportation and Logistics, 2013.
- Brodtkorb, Sætra and Hagen, **GPU Programming Strategies and Trends in GPU Computing**, Journal of Parallel and Distributed Computing, 2013.