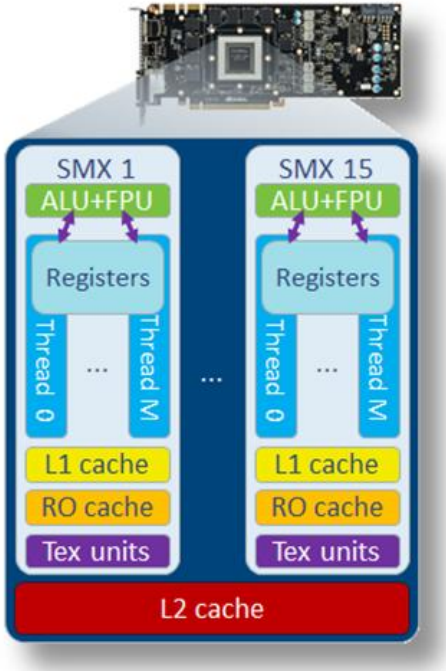
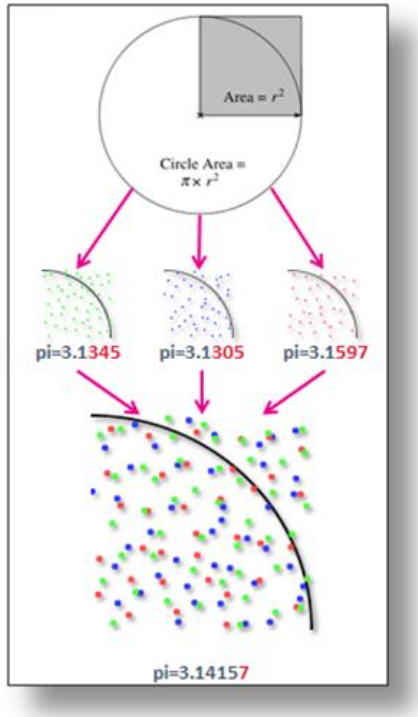
Parallel Computing Towards Exascale

André R. Brodtkorb

Visual Computing Forum #30

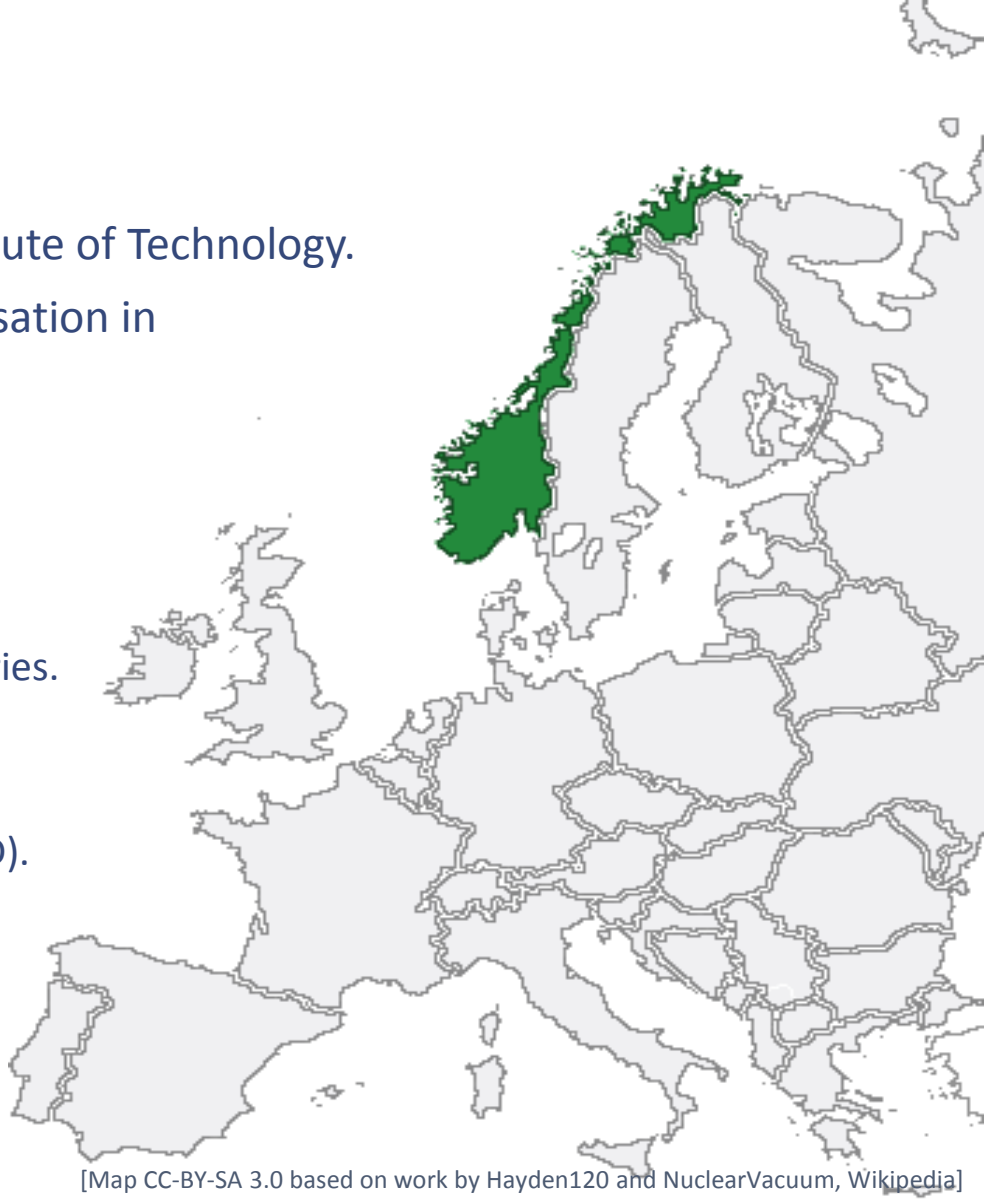
September 19, 2014





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.



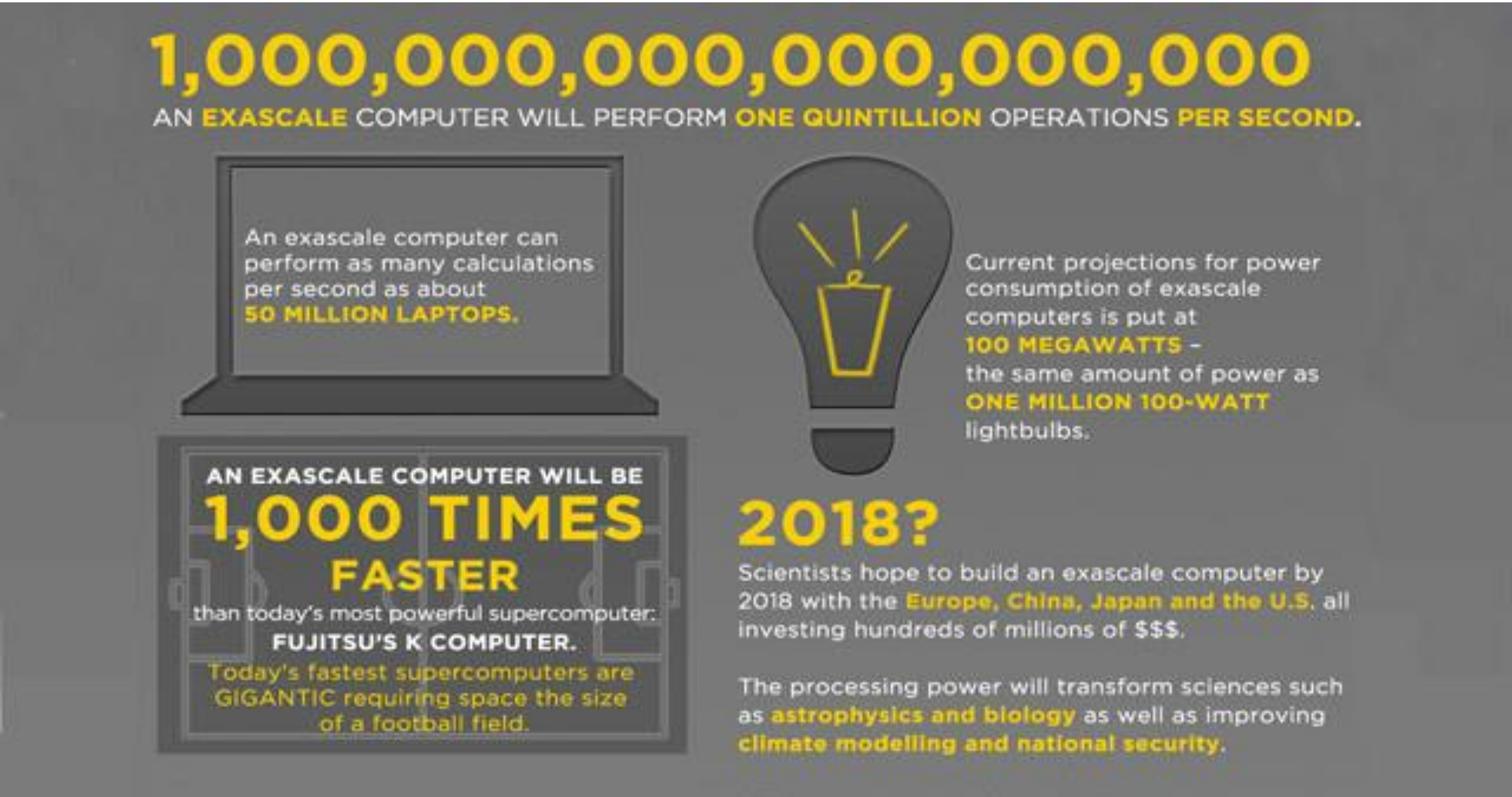
Outline

- Motivation for exascale
- Multi- and many-core architectures
- Computing π on a massively parallel machine
- Leveraging domain specific languages
- Summary

Motivation exascale

What is Exascale?

Graphic: http://edition.cnn.com/2012/03/29/tech/super-computer-exa-flop/

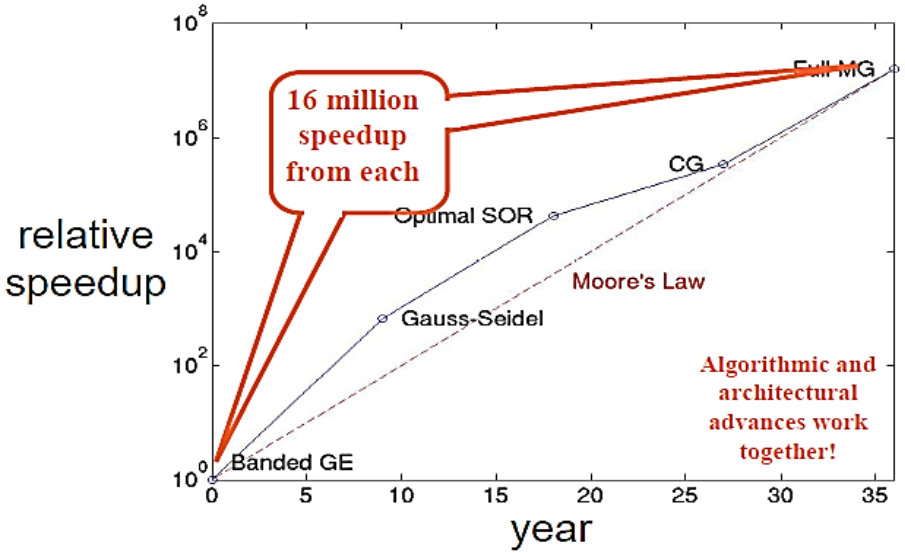


Why do we need Exascale?

- More accurate weather forecasts.
  - Extreme weather we can predict today, was impossible to foresee as little as ten years ago.
- Simulation of a human brain.
  - Is estimated to run on the order of one Exaflop.
- More accurate CFD simulations.
  - Design of supersonic aircraft, missiles, and space shuttles.
- New unforeseen simulation strategies and application areas enabled by Exascale.

Why care about "exascale hardware"?

- Same type of hardware for supercomputers and laptops.
- 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

History lesson: development of the microprocessor 1/2

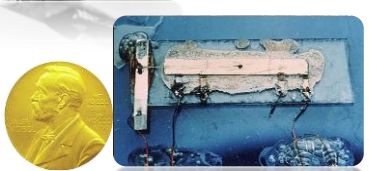
1942: Digital Electric Computer

(Atanasoff and Berry)

1956

1947: Transistor

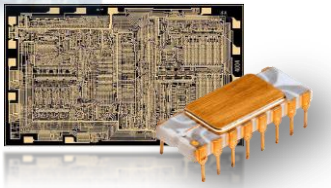
(Shockley, Bardeen, and Brattain)



2000

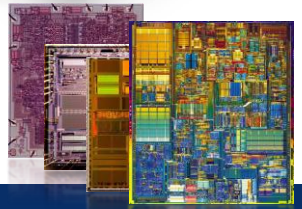
1958: Integrated Circuit

(Kilby)



1971: Microprocessor

(Hoff, Faggin, Mazor)



1971- Exponential growth

(Moore, 1965)

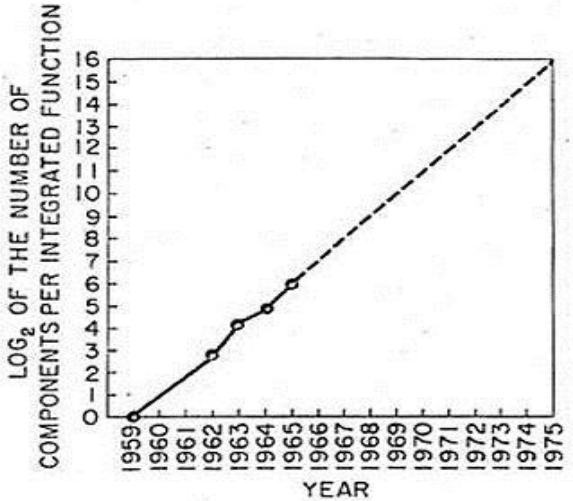
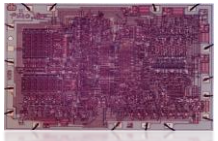
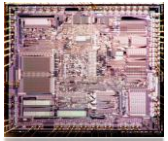


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

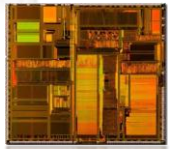
History lesson: development of the microprocessor 2/2



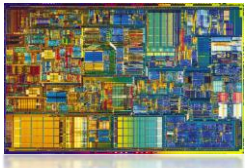
1971: 4004,
2300 trans, 740 KHz



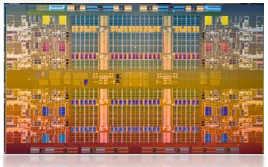
1982: 80286,
134 thousand trans, 8 MHz



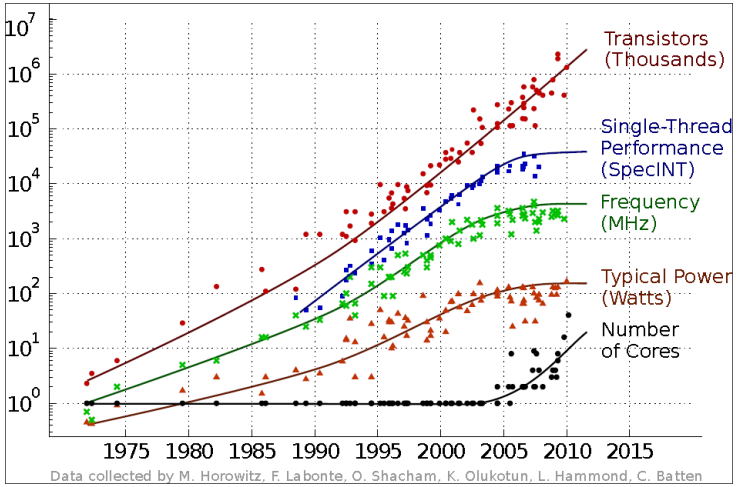
1993: Pentium P5,
1.18 mill. trans, 66 MHz



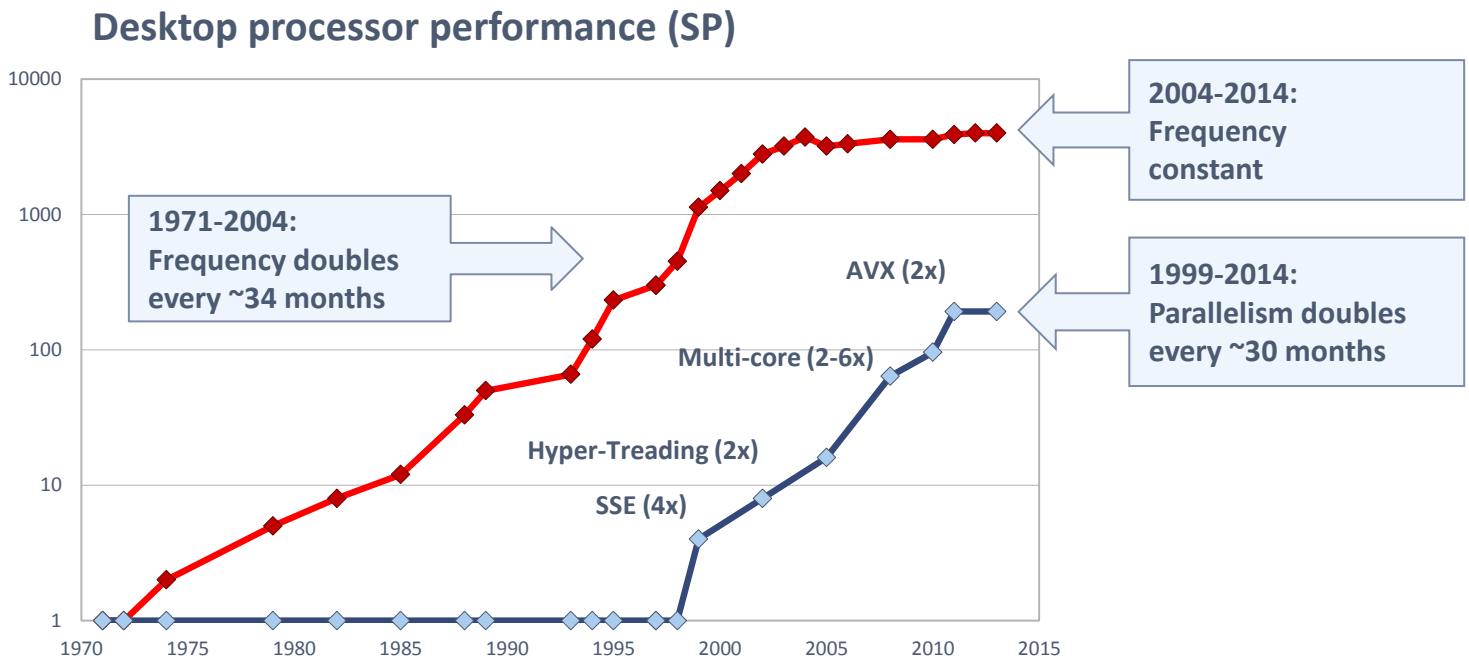
2000: Pentium 4,
42 mill. trans, 1.5 GHz



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



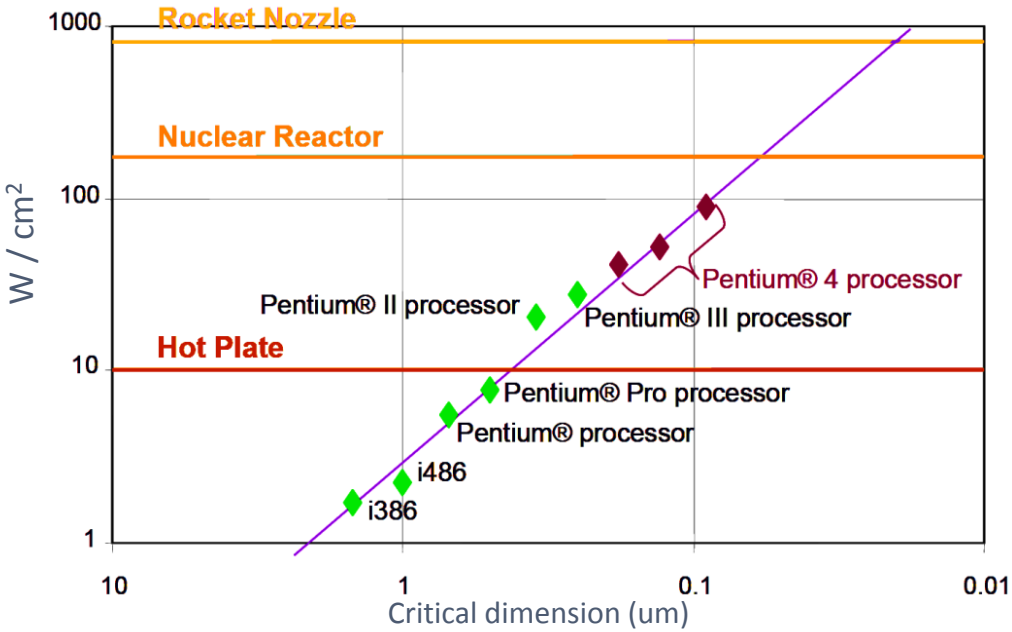
End of frequency scaling



- 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

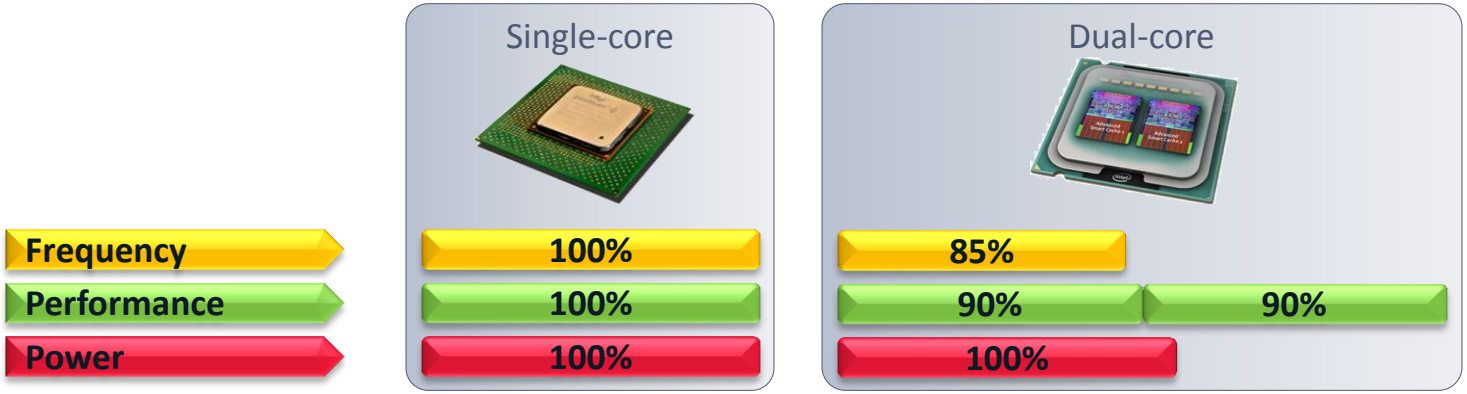


Original graph by G. Taylor, "Energy Efficient Circuit Design and the Future of Power Delivery" EPEPS'09

Why Parallelism?

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

$$P_d \propto f^3$$

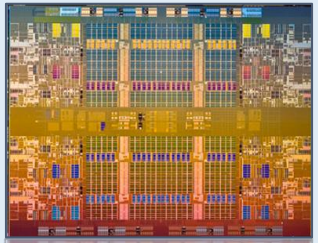


[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

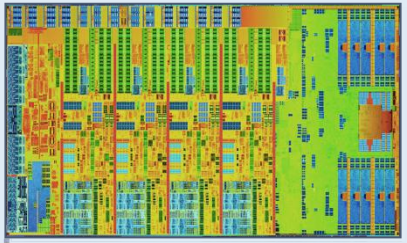
Multi- and many-core architectures

Multi- and many-core processor designs

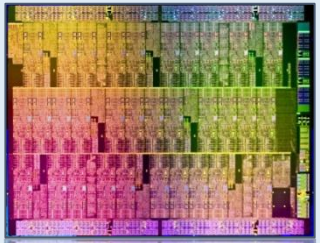
- 6-60 processors per chip
- 8 to 32-wide SIMD instructions
- Heterogeneous cores (e.g., CPU+GPU on single chip)



Multi-core CPUs:
x86, SPARC, Power 7



Heterogeneous chips:
Intel Haswell, AMD APU

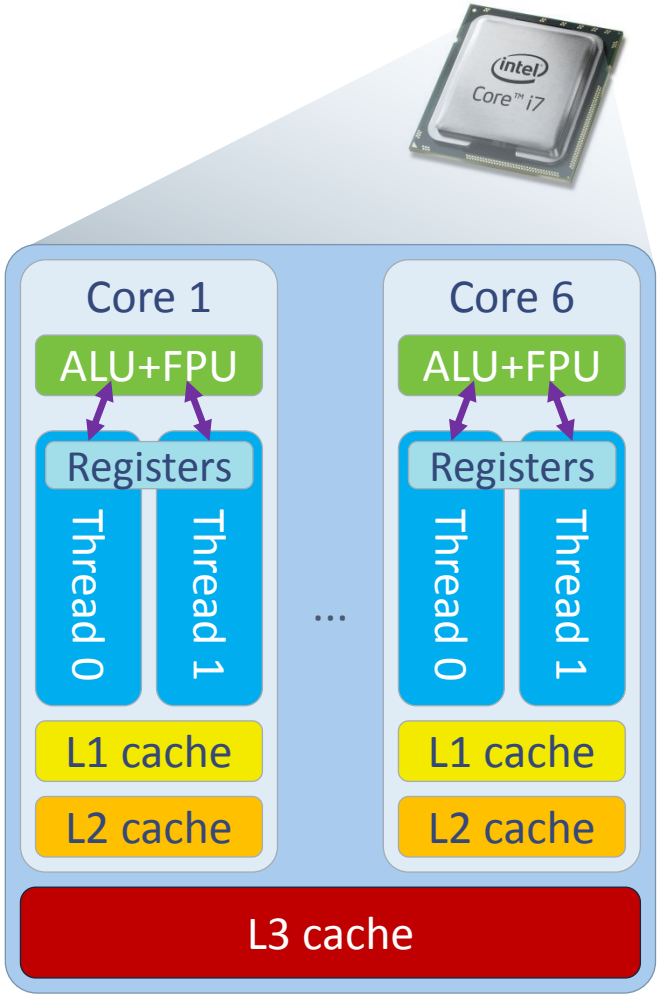


Accelerators:
GPUs, Xeon Phi

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!

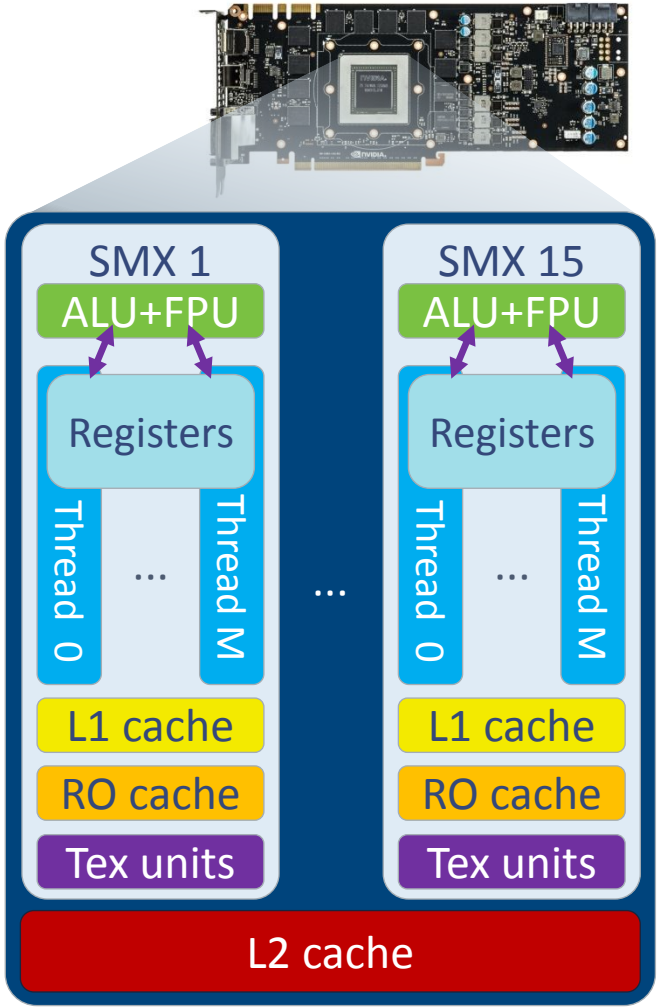


Simplified schematic of CPU design

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

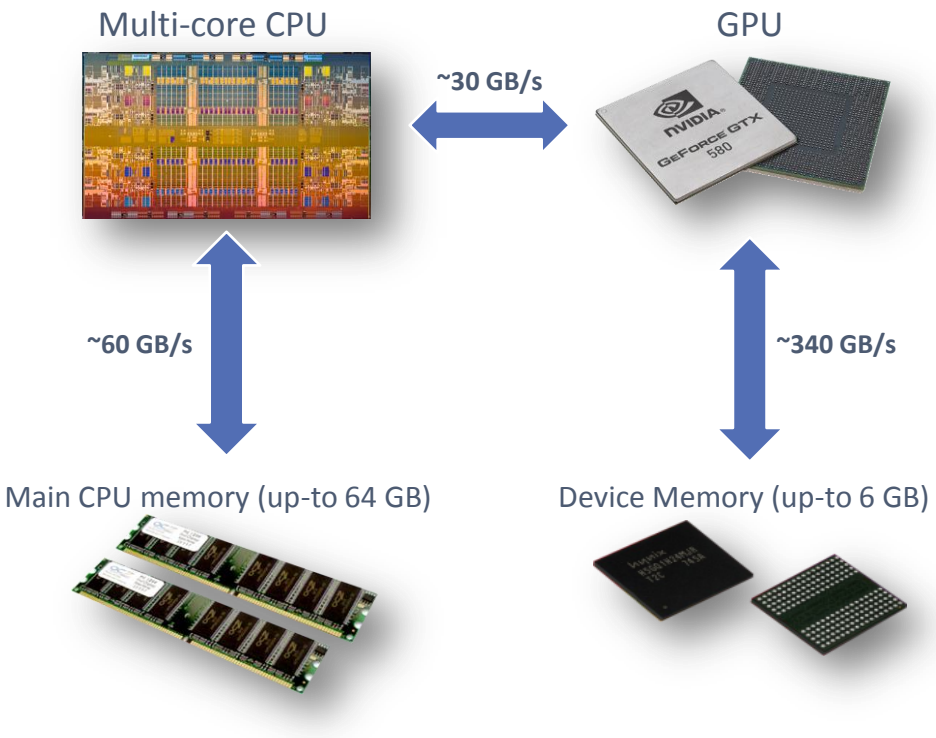
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: Six SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!



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

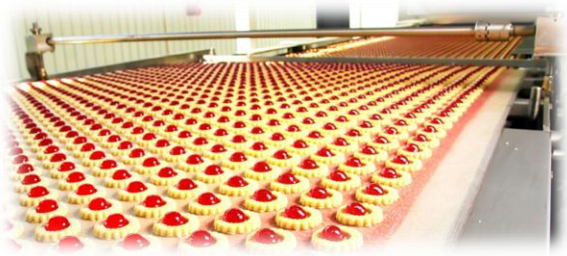
Parallel computing

- 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?

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

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

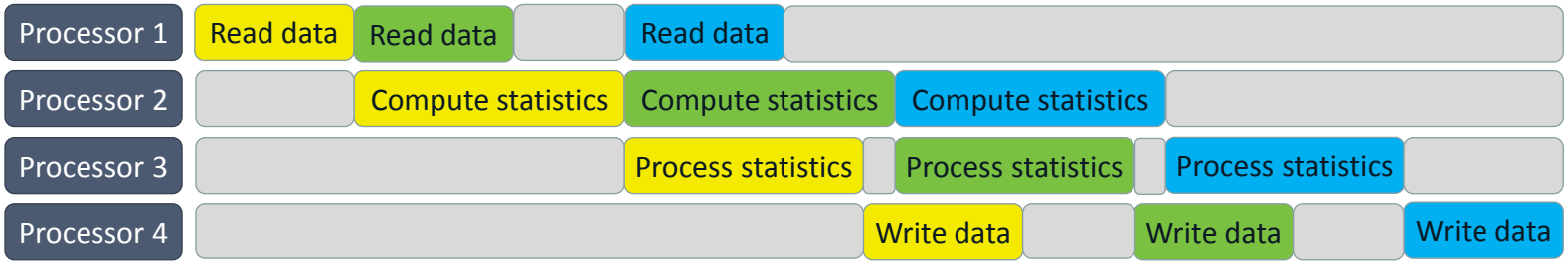
- Task parallelism divides a problem into subtasks which can be solved individually
- Scales well for a large number of tasks:
  The more parallel tasks, the more processors you can use
- Example: A simulation application:

Processor 1	Render GUI
Processor 2	Simulate physics
Processor 3	Calculate statistics
Processor 4	Write statistics to disk

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

Task parallel workloads 2/3

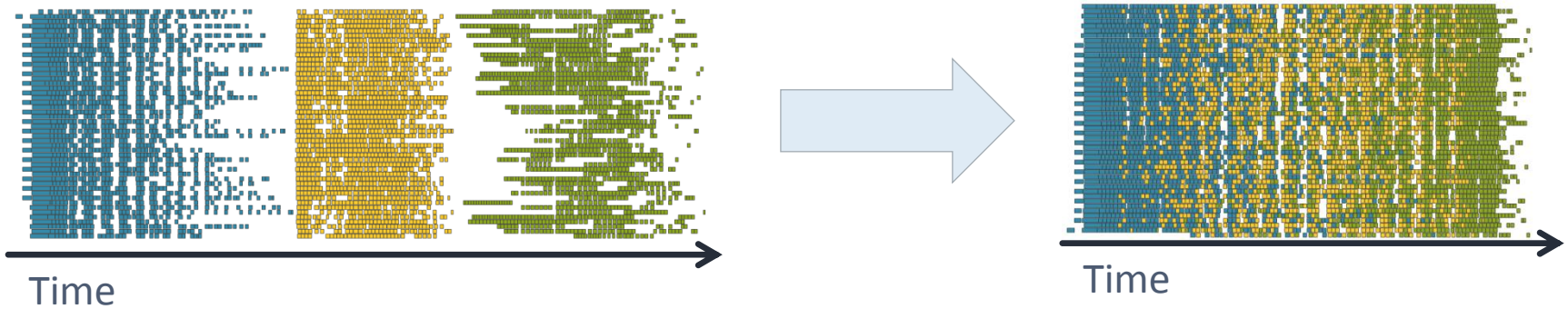
- Another way of using task parallelism is to execute dependent tasks on different processors
- Scales well with a large number of tasks, but performance limited by slowest stage
- Example: Pipelining dependent operations



- 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

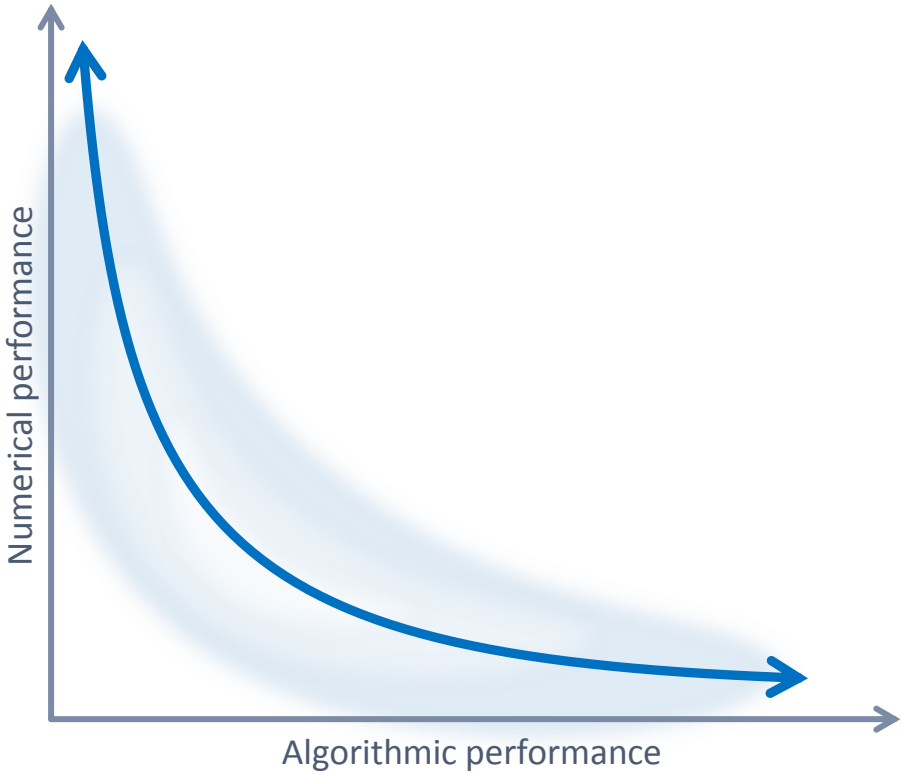


- "Gray boxes" are minimized

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

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



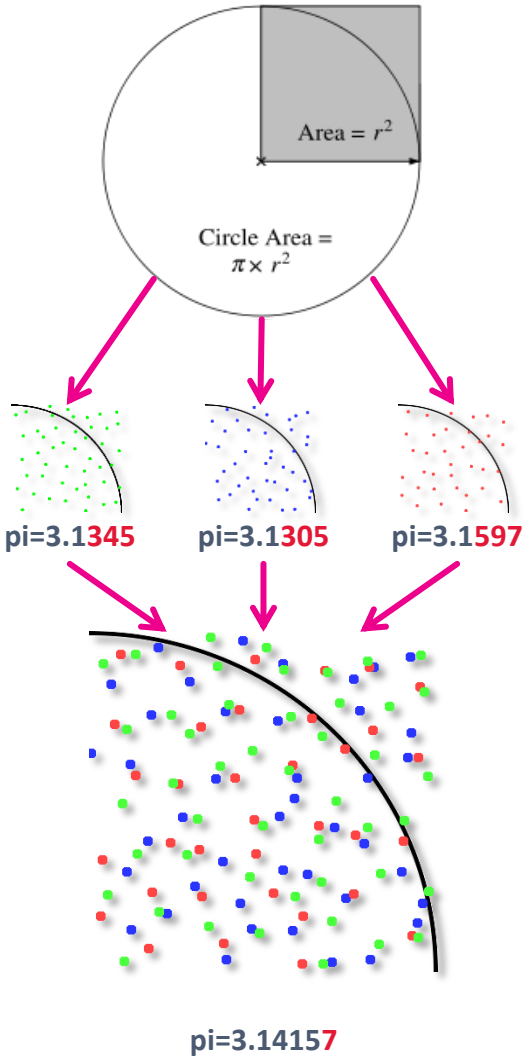
Computing PI in parallel

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 is 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

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 = n_inside + 1; }
        }
        //Estimate Pi
        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 = n_inside + 1; }
        }
        //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

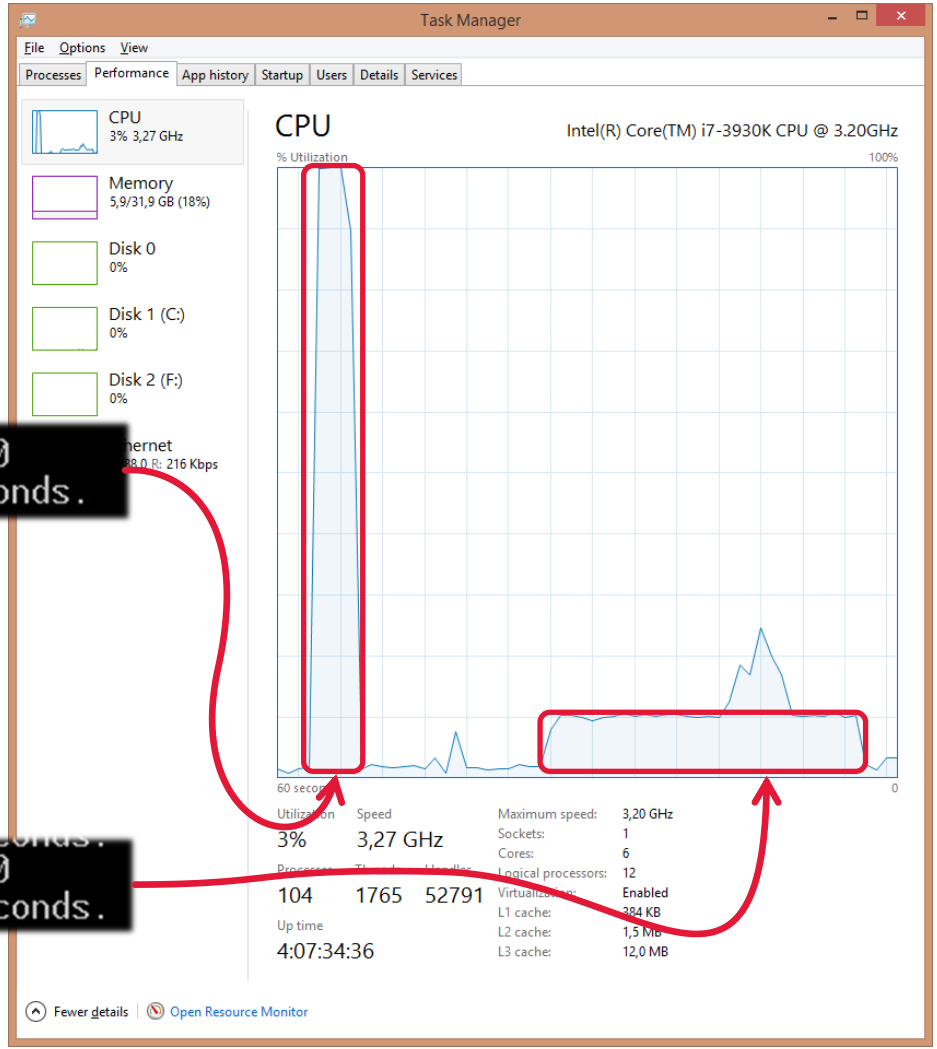
Performance

- 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

GPU function

```
__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;
        }
}
```

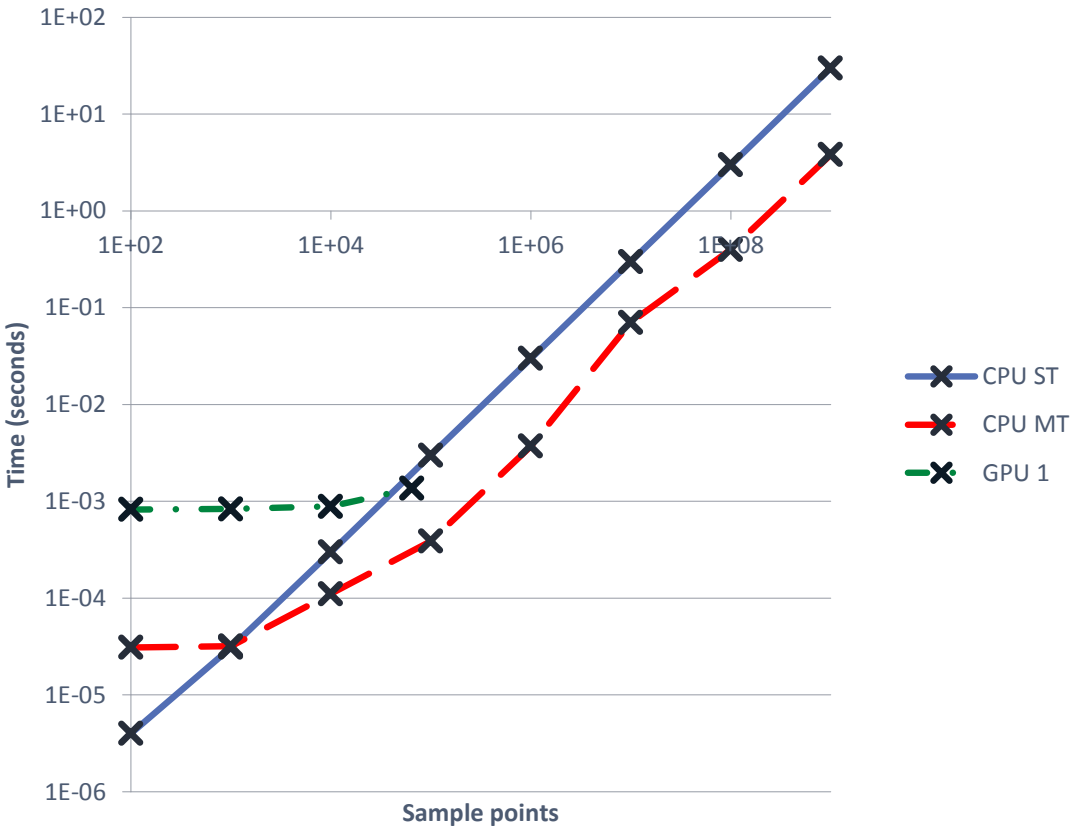
*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

Parallel GPU version 1 (CUDA) 2/3

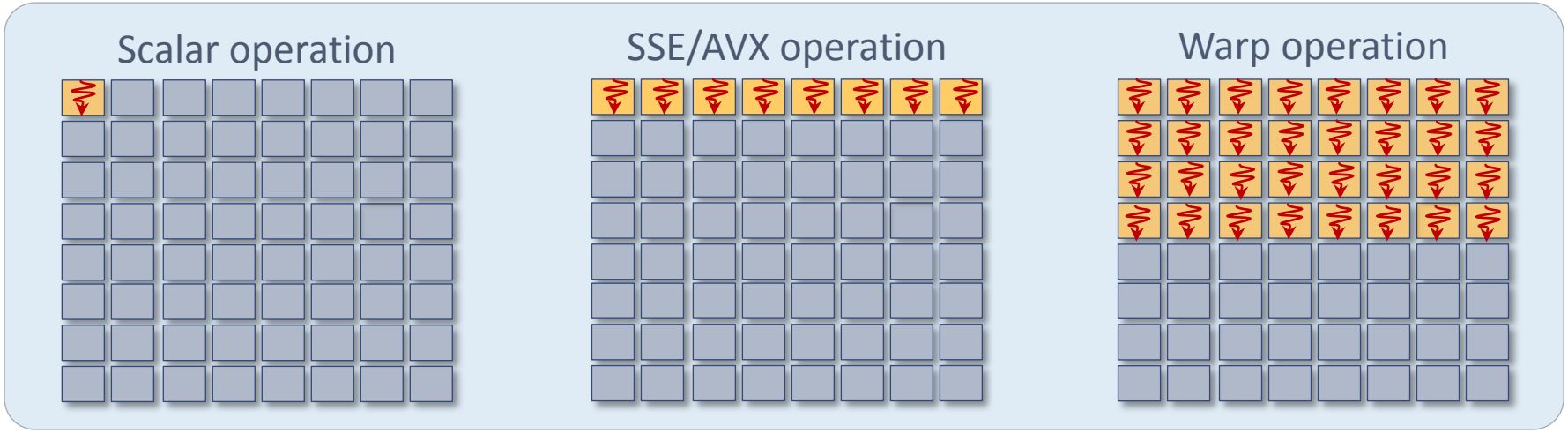
```
float computePi(int n_points) {
      dim3 grid = dim3(n_points, 1, 1);
      dim3 block = dim3(1, 1, 1);

      //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**

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;
        }
}
```

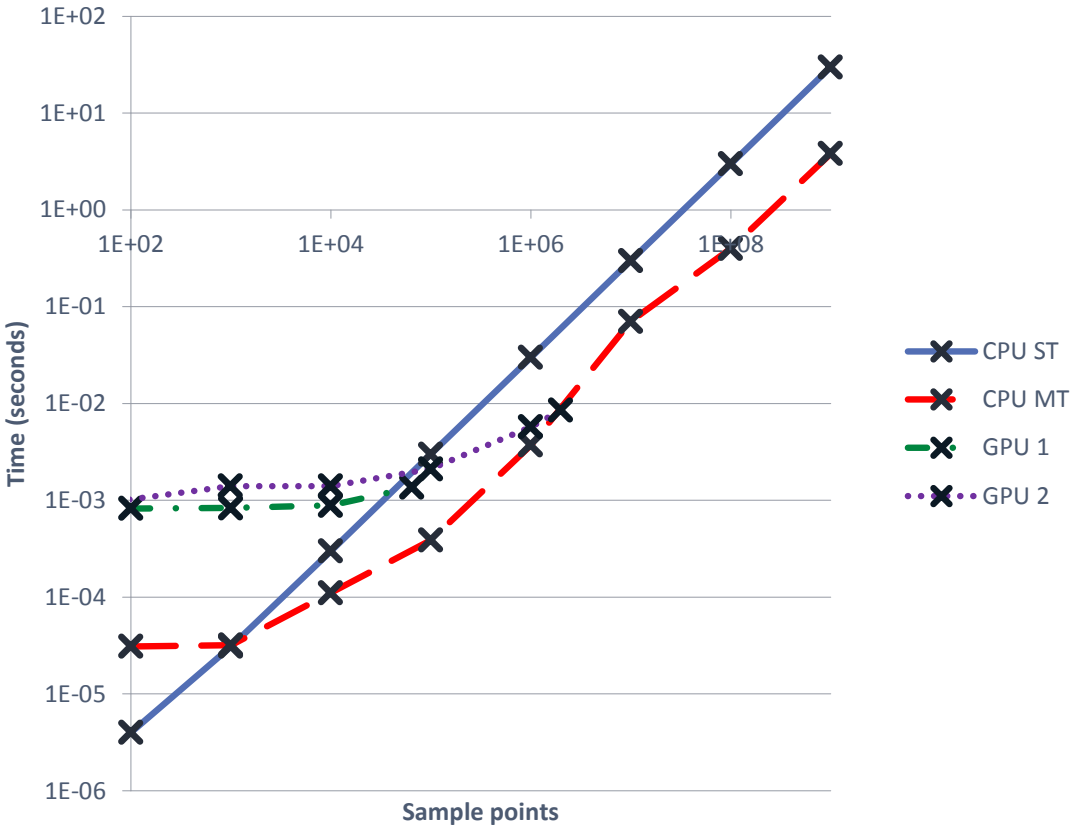
New indexing

```
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! <u>Optimal is 60 operations!</u>



Parallel GPU version 3 (CUDA) 1/3

```
__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
```

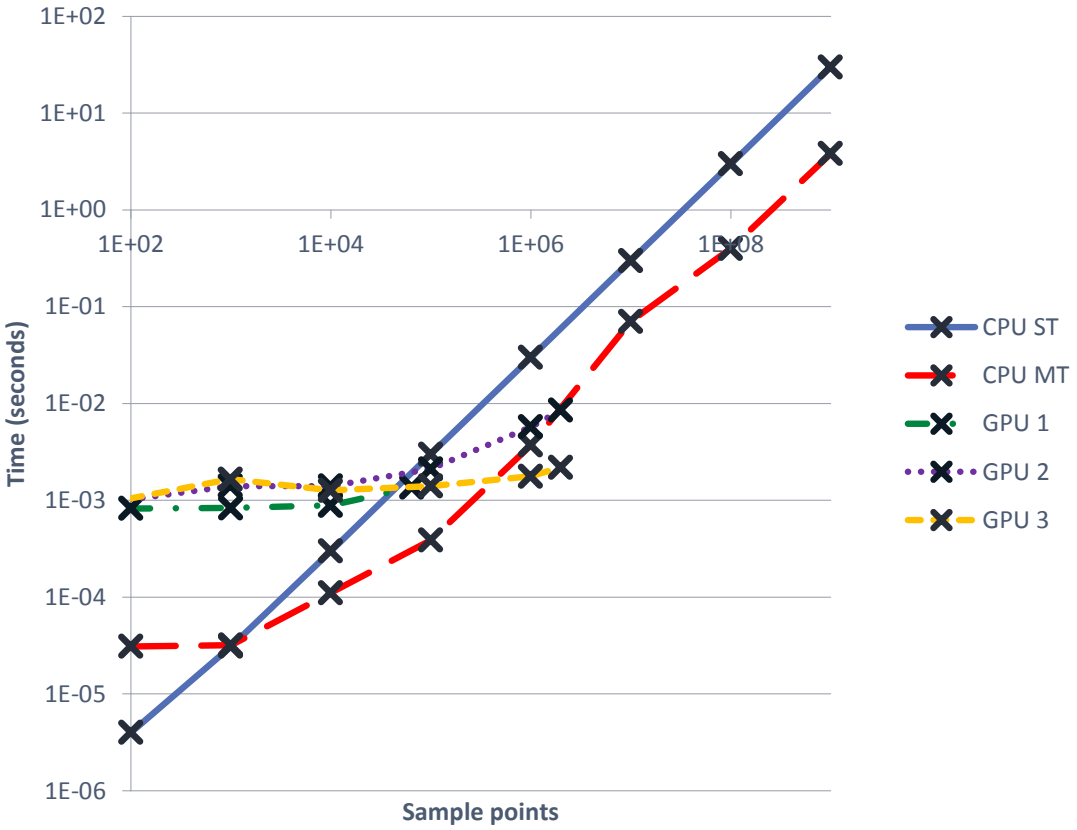
Shared memory: a kind of “programmable cache”
We have 32 threads: One entry per thread

Parallel GPU version 3 (CUDA) 2/3

```
… //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) 3/3

- 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

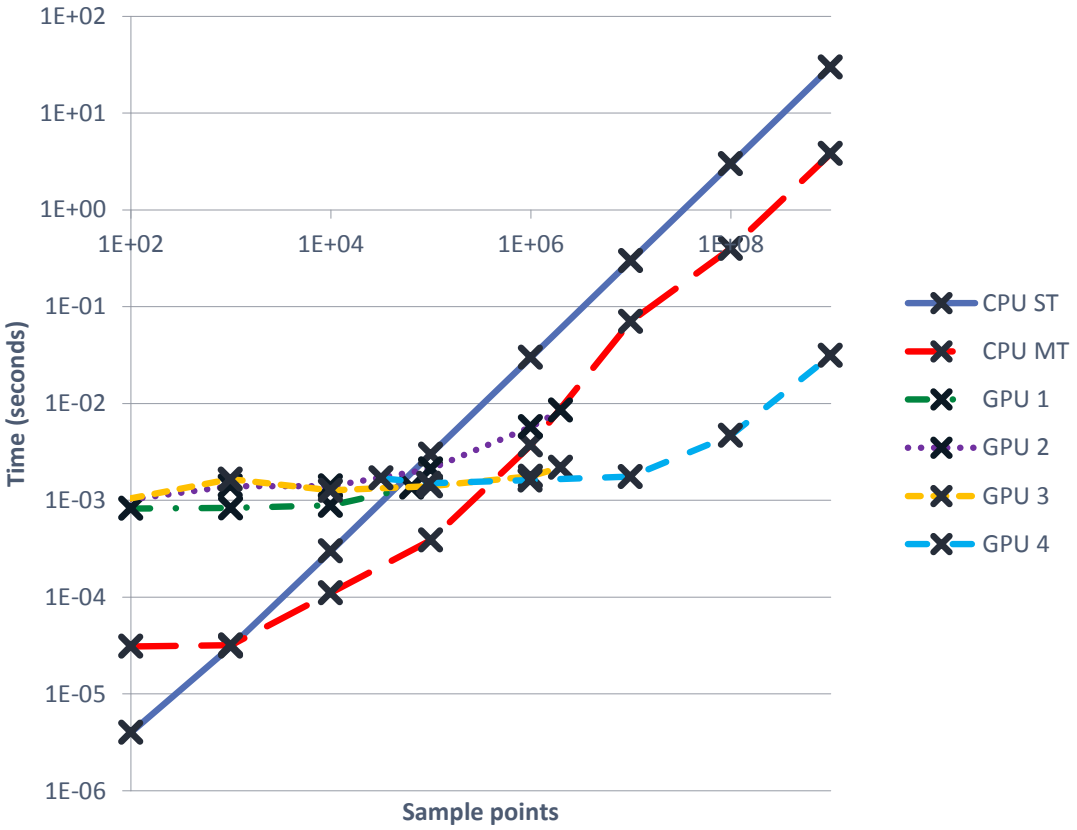
```
__global__ void computePiKernel4(unsigned int* output) {
        int n_inside = 0;

         //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).

Leveraging Domain Specific Languages

Slides based on "Simulators that write themselves",
Atgeirr Flø Rasmussen, Dune user group meeting, 2013.

Simulation is hard

- Writing a parallel simulator and running a simulation is notoriously difficult!
- Deep knowledge is required in multiple fields: Mathematics, Physics, Chemistry, Biology, Informatics, …
- Most simulator writing teams consist of **one** person: typically a single Ph.D. student.
- Most people are proficient in at most one and a half of the required levels.

Application
(Equations, Physics)

Numerics
(Discretization, Gridding)

Implementation
(C++, Parallelization)

Using domain specific languages

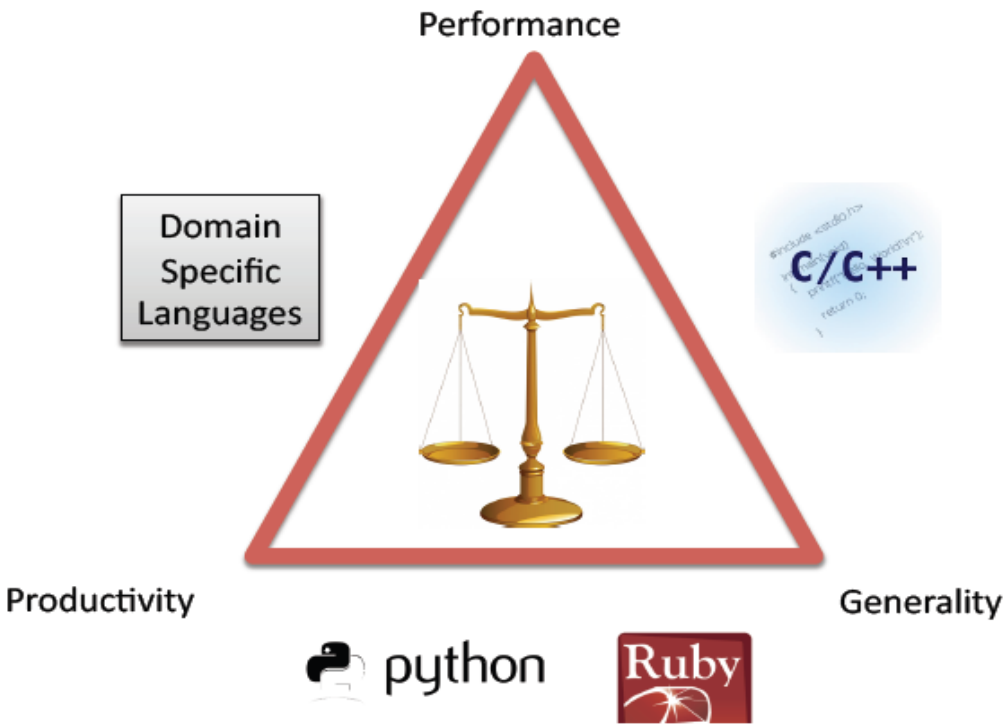


Figure from "Designing the Language Liszt" by DeVito, Joubert, Lemire, Hanrahan

Fool proof code using Equelle

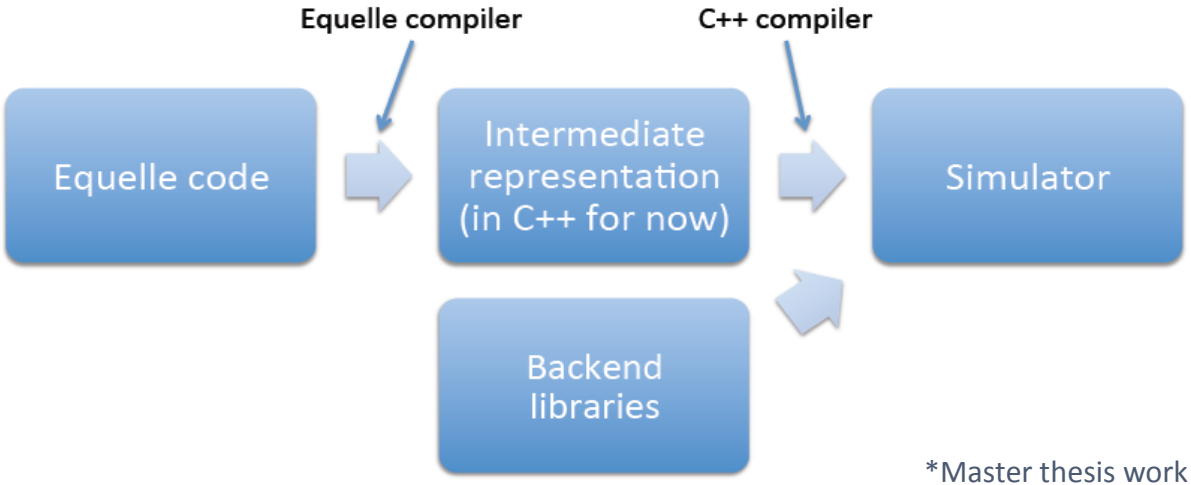
- SINTEF has designed Equelle, http://www.equelle.org/
  - Other similar languages: Liszt, Halide.

- Domain specific language
  - Will never support "everything"
  - Designed for safely writing finite-volume codes on (complex) grids
  - Currently takes Equelle programs as input, and generates C++ code

- Still early prototype

Benefits of Equelle

- Designed to prevent typical errors when writing simulators
  - All "off-by-one" and indexing errors: grid access is not handled explicitly.
  - Adding incompatible values: e.g., a face-value and a cell-value.
  - Easy to generate serial, OpenMP, CUDA*, or MPI codes.
  - Each project participant can work on the part he/she is most familiar!



*Master thesis work of Håvard Heitlo Holm, 2014

Current results

- Prototype up and running:
  Check it out yourself on http://equelle.org/ (open source!)

- The compiler found bugs in an existing simulator we had written.
  The code had been manually checked, and it required an effort to see that the compiler was right

- No optimization of abstract syntax tree performed yet:
  some performance loss is to be expected

- CUDA backend in progress for higher performance

Summary

Summary

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

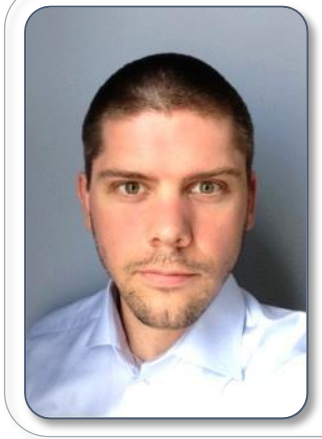
- Getting started coding in parallel has never been easier:
  - OpenMP is at your fingertips (C/C++/Fortran)
  - Nvidia CUDA tightly integrated into Visual Studio
  - Excellent profiling tools available with toolkit

- Domain specific languages can aid development
  - Can be expensive to design language first time
  - Easier to write and maintain code

Some references

- Code examples available online: http://github.com/babrodtk
- NVIDIA CUDA website: https://developer.nvidia.com/cuda-zone
- Equelle website: http://www.equelle.org/
- Brodtkorb, Hagen, Schulz and Hasle, **GPU Computing in Discrete Optimization Part I: Introduction to the GPU,** 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.

Thank you for your attention!



André R. Brodtkorb
Email:
Andre.Brodtkorb@sintef.no
Homepage:
http://babrodtk.at.ifi.uio.no/

SINTEF webpages:
http://www.sintef.no/math/