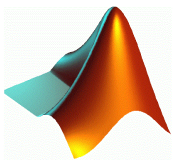
Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Introduction

Torben Larsen
Aalborg University



Course Content and Structure

- **What is Jacket? Why? How?**

"Jacket is a really wild horse to ride on ..."
- *Jacek Pierzchlewski, course assistant*

The objective of the course is to provide you with knowledge of GPU based computing. You will know about the strengths and weaknesses of the GPU computing paradigm. You will get hands on experience in programming single and multiple GPUs by use of MATLAB and Jacket. And you will learn to benchmark the code you develop.

- **Course content:**
  - **Parallel High Performance Computing – With Emphasis on Jacket Based GPU Computing**

- **Course overview:**
  - Day 1 – morning: lectures + simple hands-on examples.
  - Day 1 – afternoon: exercises based on the lectures.
  - Day 2 – morning: lectures + simple hands-on examples.
  - Day 2 – afternoon: exercises.
  - Day 3 – morning: lectures + simple hands-on examples.
  - Day 3 – afternoon: Introduction to "The Challenge".
  - Day 4 – 5: "The Challenge".
  - May 25: Deliver report on the challenge. A minimum 10 maximum 15 page report (10 pt. font size; Times New Roman font ; A4 page style; 20 mm margins on top-bottom-left-right). The report must describe: a) brief statement of problem including constraints; b) methodology; c) implementation considerations; d) benchmarking across the required hardware platforms; e) conclusion.

- **To pass:**
  - Participate in at least 3 out of 5 course days (unless otherwise agreed with Torben Larsen).
  - Deliver mandatory exercises including documentation to explain what has been done and why.
  - Final challenge. Must show you have understood the main concepts.

Course Content and Structure
Course Overview

- **Day 1 morning:**
  - Lecture 1: Introduction
  - Lecture 2: Jacket Introduction
- **afternoon:**
  - Exercises.

- **Day 2 morning:**
  - Lecture 3: Basic Programming
  - Lecture 4: Advanced Programming
- **afternoon:**
  - Exercises

- **Day 3 morning:**
  - Lecture 5: Programming Multiple GPUs
  - Lecture 6: Benchmarking
- **Day 3 afternoon:**
  - Exercises
  - Introduction to "**The Challenge**"

- **Day 4 morning:**
  - "**The Challenge**"
- **Day 4 afternoon and day 5:**
  - **"The Challenge"**

Course Overview

- **Lecture 01: Introduction**
  - Course Content and Structure
  - The World Is Going Parallel … In GPUs
  - GPU Platforms
  - Floating Point Representation
  - MATLAB Vectorization
  - Practical Information
  - Abbreviations

- **Lecture 02: Jacket Introduction**
  - Jacket Teaser
  - Jacket Framework
  - Jacket Fundamentals
  - Some Examples
  - Conclusions

- **Lecture 03: Basic Programming**
  - Programming Methodology
  - Maintaining CPU and GPU Code in 1 File
  - Efficient Array Indexing
  - Acquiring System Information
  - Benchmarking MATLAB and Jacket Functions
  - Jacket Code on Non-Jacket Enabled MATLAB Installations
  - Computational Threads
  - Handling Scalars in Jacket
  - Is Jacket Column or Row Major?

- **Lecture 04: Advanced Programming**
  - GPU Characteristics
  - Stop-Resume Paradigm
  - GFOR Loops
  - Handling Large Amounts of Data
  - Case Study: Jacobi and Hessian Computation

- **Lecture 05: Programming Multiple GPUs**
  - Problem and Motivation
  - SPMD: Single Program Multiple Data
  - Selecting Specific GPUs
  - Example: Simultaneous Multi-GPU GFLOPS Measurement
  - Case Study: Heterogeneous GPU Computations #1
  - Case Study: Hybrid CPU and GPU Computing
  - Case Study: Heterogeneous GPU Computations #2
  - Conclusions

- **Lecture 06: Benchmarking**
  - Overview
  - Core Principles
  - Floating Point
  - Memory Access
  - Functions
  - Disk Access
  - Toolboxes

1) **Course Content and Structure**
   - Objectives
   - Lectures
   - Exercises
2) **The World Is Going Parallel – In GPUs**
   - CPU and GPU Developments
   - Flynn's Taxonomy
   - Amdahls Law
   - Gustafson-Barsis' Law
   - Comparison of Amdahls and Guftafson-Barsis' Laws
   - Conclusions and Trends
3) **GPU Platforms**
4) **Floating Point Representation**
   - IEEE 754
   - Case Study: When $x^2+kx \neq x(x+k)$
5) **MATLAB Vectorization**
   - Introduction
   - BSXFUN Case Study: Euclidian Distances in Space
   - Case Study: Loops – Always to be Avoided?
6) **Practical Information**
   - Computer Platforms
   - Login
7) **Abbreviations**
8) **Further Reading**

The World Is Going Parallel

...

In GPUs

The World Is Going Parallel … In GPUs
CPU and GPU Developments

- **National Science Foundation, USA, April 2011:**

> The end of dramatic exponential growth in single-processor performance marks the end of the dominance of the single microprocessor in computing. The **era of sequential computing must give way to a new era in which parallelism is at the forefront**. Although important scientific and engineering challenges lie ahead, this is an opportune time for innovation in programming systems and computing architectures. We have already begun to see diversity in computer designs to optimize for such considerations as power and throughput. The **next generation of discoveries is likely to require advances at both the hardware and software levels of computing systems**.
>
> **There is no guarantee that we can make parallel computing as common and easy to use as yesterday's sequential single-processor computer systems**, but unless we aggressively pursue efforts suggested by the recommendations in this book, it will be "game over" for growth in computing performance. If parallel programming and related software efforts fail to become widespread, the development of exciting new applications that drive the computer industry will stall; if such innovation stalls, many other parts of the economy will follow suit.
>
> **- http://bit.ly/hYqH2H**

The World Is Going Parallel … In GPUs
CPU and GPU Developments

- **Raw performance can't anymore be met by just frequency scaling – we now see multi-core CPUs, which is a means to achieve high performance by parallelization. But software is generally immature to handle this new paradigm.**
- **Intel just released e.g. Xeon X5670 (Westmere) CPUs with 6 cores – approx. 150 GFLOPS single precision and 75 GFLOPS double precision with all cores in action.**
- **From top500.org Nov. 2010 – the worlds most powerful computers:**
  - **#1: Tianhe 1A: 186,368 cores**, CPU and GPU based. Placed in National Supercomputing Center, China. **Measured peak performance: 2566 TFLOPS** (**theoretical: 4701 TFLOPS**). Based on Intel X5670 (Westmere) 6 core and NVIDIA GPUs. Designed by NUDT. **Power: 4.0 MW**.
  - **#2: Jaguar: 224,162 cores**, CPU based. Placed at Oak Ridge National Laboratory, USA. **Measured peak performance: 1759 TFLOPS** (**theoretical: 2331 TFLOPS**). Based on AMD Opteron 6 core CPUs. Designed by Cray (XT5-HE). **Power: 7.0 MW**.
  - **#3: Nebulae: 120640 cores**, CPU (and GPU) based. Placed at National Supercomputing Centre in Shenzhen, China. **Measured peak performance: 1271 TFLOPS** (**theoretical: 2984 TFLOPS**). Based on Intel Xeon X56xx (Westmere) 6 core and NVIDIA C2050 GPUs. Designed by Dawning. **Power: 2.6 MW**.
- **The way to performance is cores – MANY cores …. Really MANY cores. … and POWER.**
- **Clear tendency towards using GPUs (Graphics Processing Units). GPUs are now more a high core (hundreds of cores) general computation unit. The change happened in 2010.**

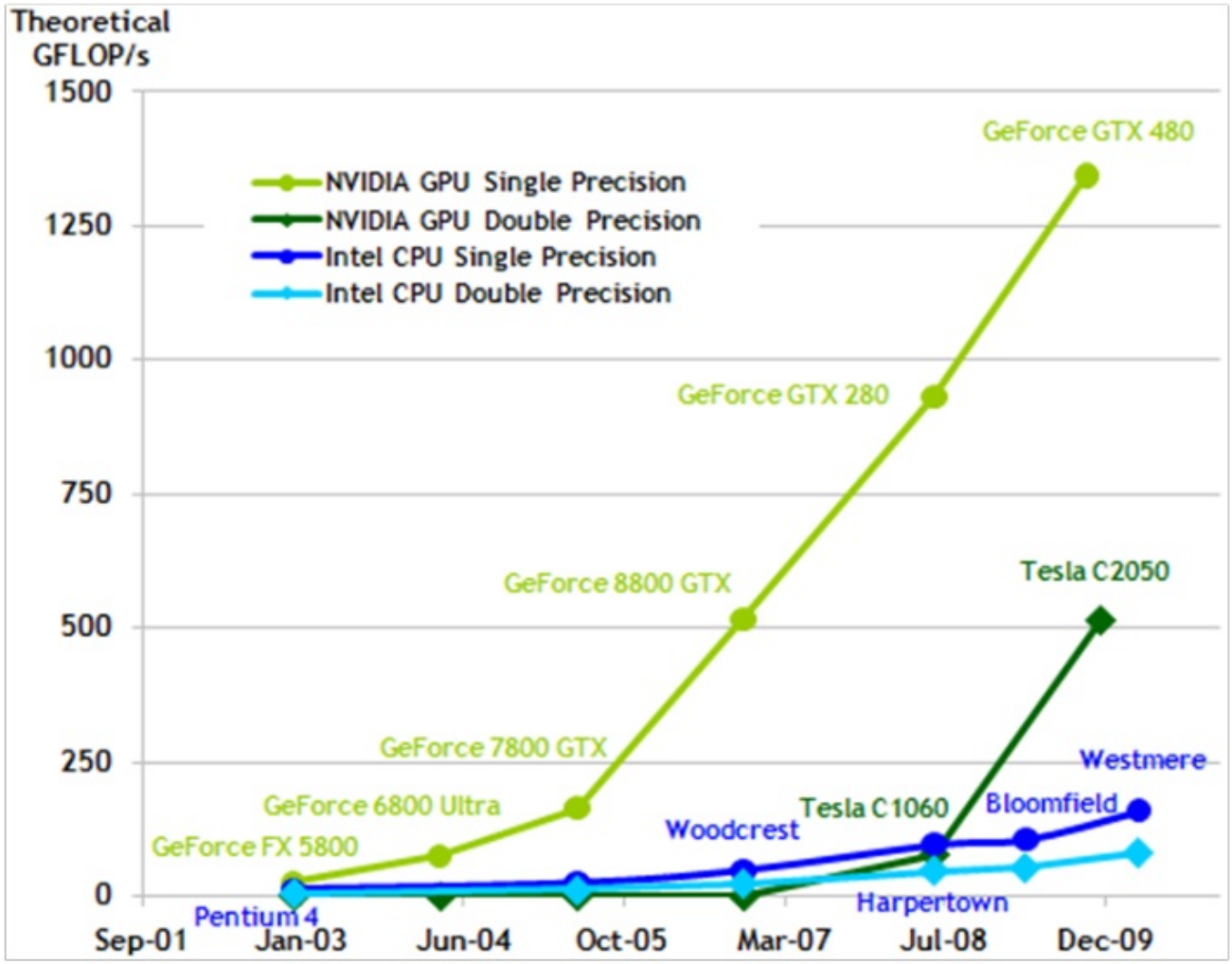
- Interesting observations:

Computer System	Efficiency Meas./theor.	Meas. Power eff. GFLOPS/W
Tianhe 1A	**54.6 %**	**0.64**
Jaguar	**75.5 %**	**0.25**
Nebulae	**42.6 %**	**0.49**

- Some indications:
  - **The CPU based system (Jaguar) has higher efficiency – it is easier to utilize the computational power of the CPUs than what we see for the GPU based systems (Tianhe and Nebulae).**
  - **The GPU based systems are way more power efficient than the CPU based solution.**

The World Is Going Parallel … In GPUs
CPU and GPU Developments

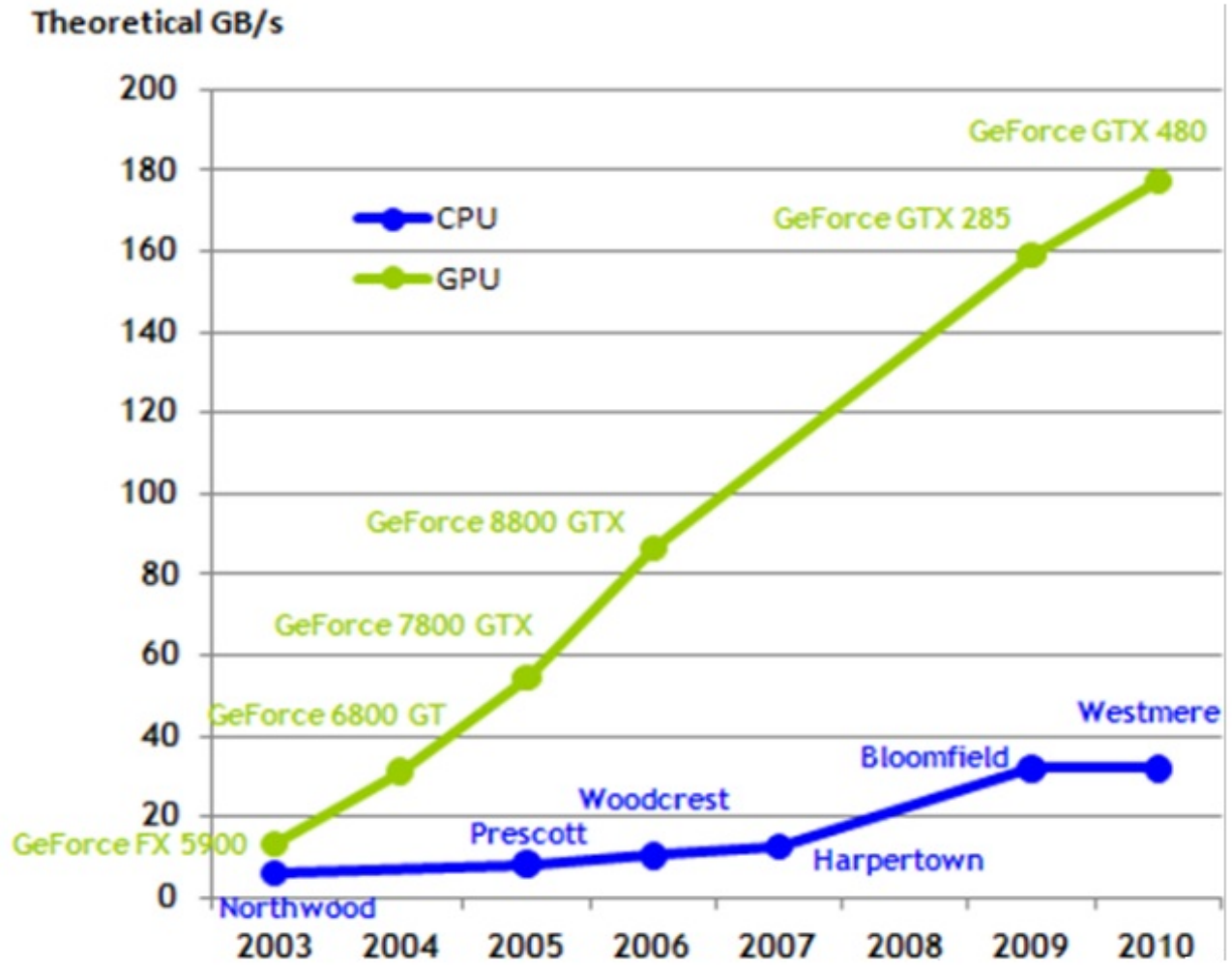
- **Raw floating point performance of CPUs and GPUs …**



From: "NVIDIA: NVIDIA CUDA C Programming Guide, 2010".

The World Is Going Parallel … In GPUs
CPU and GPU Developments

- **Raw memory bandwidth of CPUs and GPUs:**



From: "NVIDIA: NVIDIA CUDA C Programming Guide, 2010".

- **Estimation of peak floating point performance for recent Intel CPUs:**

$$\mathcal{P}_{\text{Intel}} = f \cdot N_{\text{cores}} \cdot k_{\text{opc}} \cdot \frac{128}{d} \quad [\text{Flops}]$$

where:

- f is the clock frequency, which can be sustained with the number of cores used.
- N_{cores} is the number of cores in the CPU.
- k_{opc} is the number of floating point operations the CPU can deliver (normally a multiply and add at in the same clock cycle).
- d is the data length (32 for single precision and 64 for double precision)

- **Examples for CPUs working with single precision data:**

$$\mathcal{P}_{\text{Core i7}-970,\text{sp}} = 3.20 \cdot 6 \cdot 2 \cdot \frac{128}{32} = 153.6 \text{ GFlops}$$

$$\mathcal{P}_{\text{Xeon X5570,sp}} = 2.93 \cdot 4 \cdot 2 \cdot \frac{128}{32} = 106.6 \text{ GFlops}$$

- **Note that only one core can run at the hyper frequency used by Intel – when all cores are in operation the max. sustainable frequency is less (it is the "standard" frequency given).**

- **Estimation of peak floating point performance for NVIDIA GPUs is far from trivial. The GPUs are designed differently – even the most recent Fermi architecture differs from type to type. And the GeForce architecture has some double precision units disabled – meaning Quadro/Tesla has a single/double unit ratio of ½ where it for GeForce is ¼.**
- A few examples (and don't extrapolate!!!):

$$\mathcal{P}_{\text{nvidia}} = f \cdot N_{\text{shp/sm}} \cdot N_{\text{sm}} \cdot k_{\text{opc}} \cdot k_{\text{sp,ratio}} \quad [\text{Flops}]$$

where:

- f is the clock frequency, which can be sustained with the number of cores used
- $N_{\text{shp/sm}}$ is the number of shading processors per streaming processor
- N_{sm} is the number of streaming processors
- k_{opc} is the number of floating point operations the CPU can deliver (normally a multiply and add at in the same clock cycle)
- $k_{\text{sp,ratio}}$ is the ratio of double to single precision units

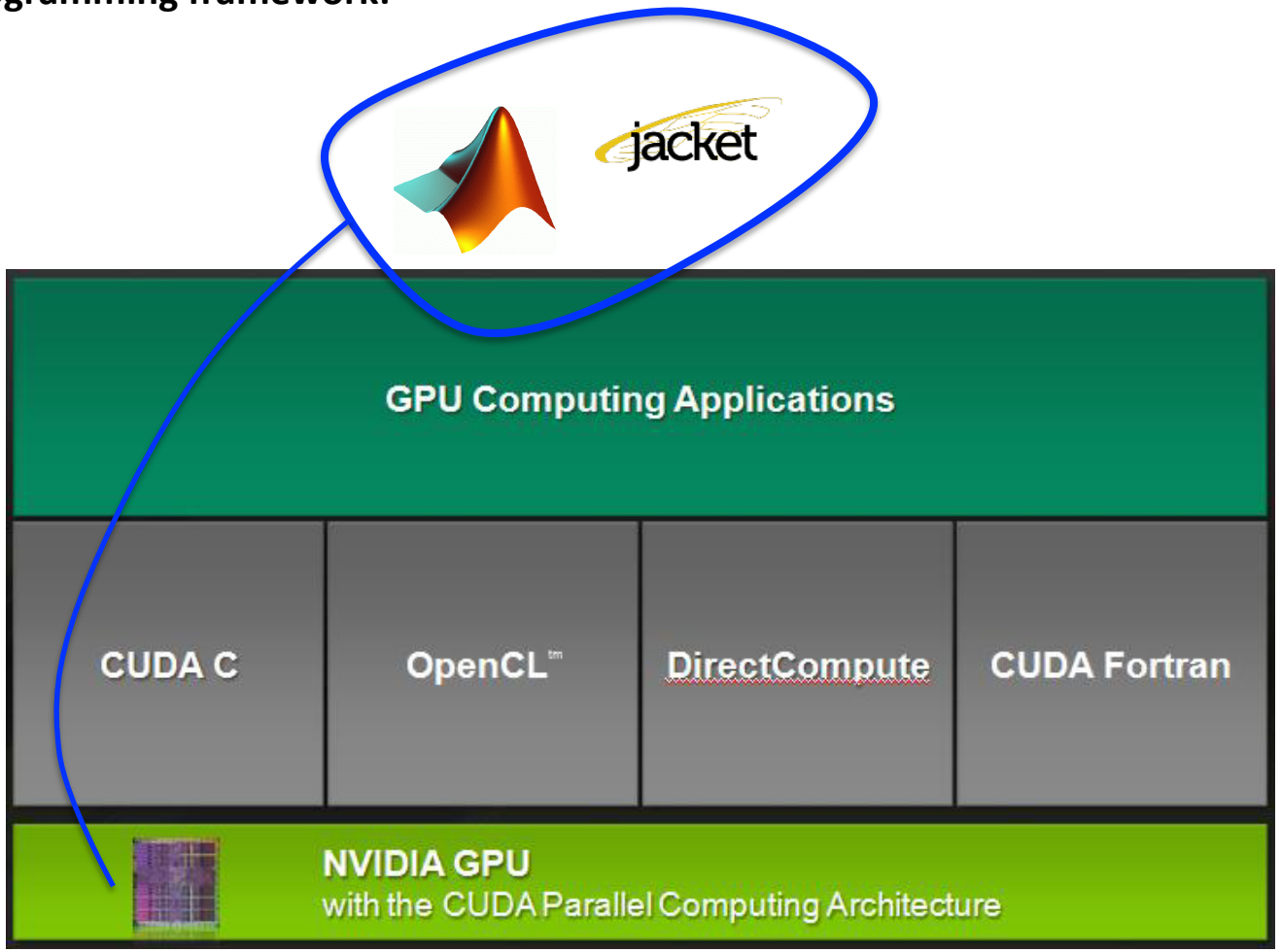
- **Examples for CPUs working with single precision data:**

$$\mathcal{P}_{4000,\text{sp}} = 0.95 \cdot 8 \cdot 32 \cdot 2 \cdot \frac{1}{1} = 486.4 \text{ GFlops}$$

$$\mathcal{P}_{\text{C2070,dp}} = 1.15 \cdot 14 \cdot 32 \cdot 2 \cdot \frac{1}{2} = 515.2 \text{ GFlops}$$

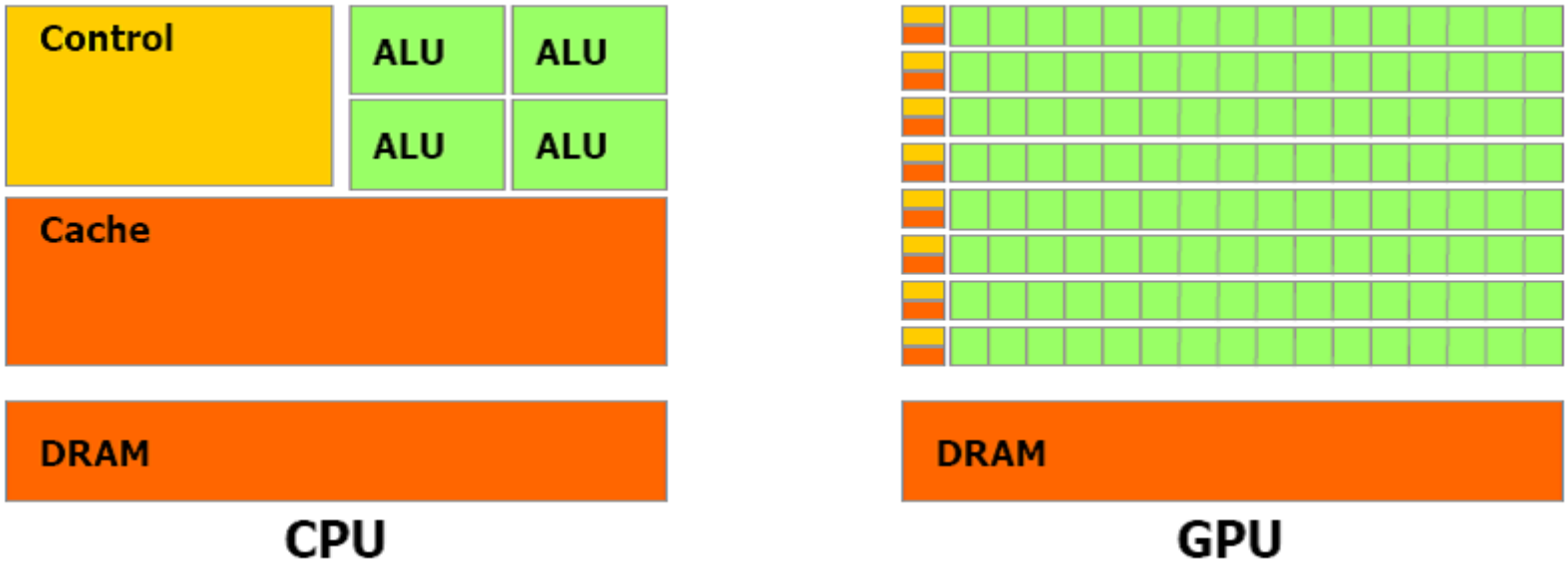
- **So where are GPUs used?**
  - Signal processing in general (LU decomposition, matrix operations etc.) [6-11]
  - Imaging [12-15]
  - Data base management [16]
  - N-body simulation [17]
  - Molecular dynamics [18]
  - …
- **History** …
  - The by far most developed tool is NVIDIAs CUDA tool released in 2006 [3].
  - Other low-level tools have appeared – like Brook from Stanford. A paradigm allowing parallel computing on a GPU, which is attached as a co-processor.
  - Jacket was released in early 2009 as a commercial product. Before that it was a freely available tool
  - In the autumn of 2010 MATLAB released support for GPUs with the PCT Toolbox – however, in a very limited version meaning that it is essentially useless for practical use. Very limited functional support and the paradigm does not allow efficient computing due to the lack of a lazy-like possibility

- **Jacket programming framework:**



From, NVIDIA: "NVIDIA CUDA C Programming Guide", 2010 [3]

- **The CPU is a computational unit characterized by a very complicated control logic and large cache. What is shown below is the typical CPU/GPU relations:**



- New GPUs such as the NVIDIA Fermi (Tesla C20xx / Quadro x000 / GeForce GTX4xx) have ECC (Error Correcting Code) and multiple cache levels while having hundreds of cores
- ECC is important in HPC as nobody wants computations to run for a long time just to see them fail short of the finish line
- The GPUs start to look more and more like CPUs except for the control part

- **CUDA is based on a heterogeneous computing scheme in which the CPU is acting as a host and the GPU(s) is(are) acting as device(s). The GPU is acting as a kind of co-processor**

- **GPUs are identified by a compute capability:**
  - 1.0-1.2: Single precision only GPUs – older types.
  - 1.3: Essentially the Tesla C1060 – a special computing GPU with strength in single precision. Delivers 78 GFLOPS double precision, which is not much any more. On the other hand it is fairly easy to reach 90+ % of theoretical performance. Lowest compute capability for double precision.
  - 2.0+: The Fermi generation, which was a huge step forward compared to the older technologies (GTX260 and the likes). Fermi architecture GPUs exist in GeForce (e.g. GTX580), Quadro (2000 and 4000) as well as in Tesla (C2050 and C2070).

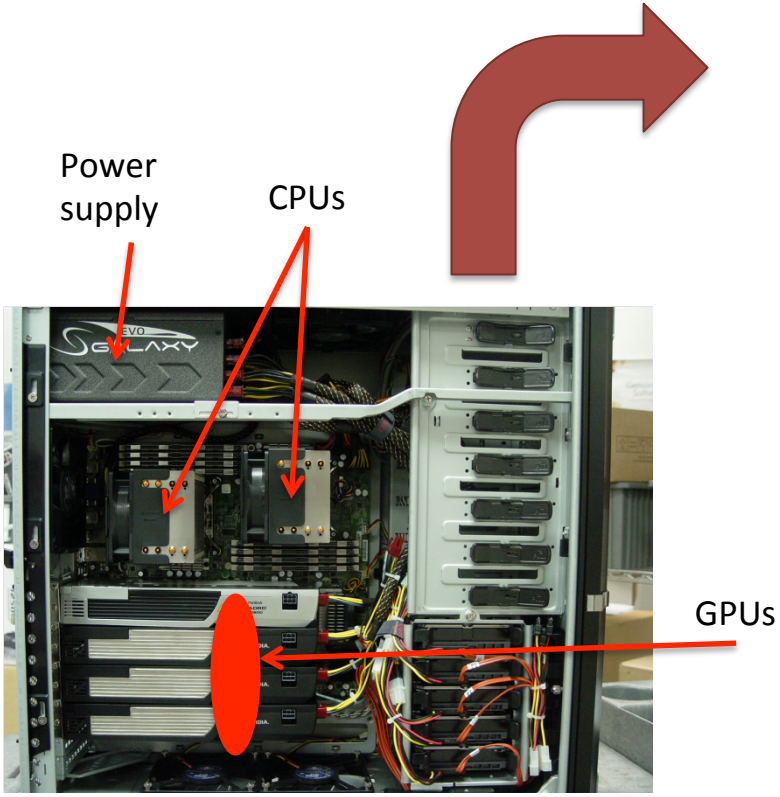
- **So GPUs hold lot of computational power … what's the downside?**
  - Development tools: OpenMP, CUDA, … are low level computational tools. They are difficult to use if you really want to squeeze out the last FLOP out of them
  - Drivers: It is extremely important to always use the latest drivers. Drivers can at present time destroy a computer – I had an unfortunately incidence with an NVIDIA driver which killed a Sony laptop during tough computations. Keep an open eye for driver info
  - Moving data; At the moment the PCIe 2.0 x16 bus is the fastest. And for high performance computing it is moving data, which is usually the most critical

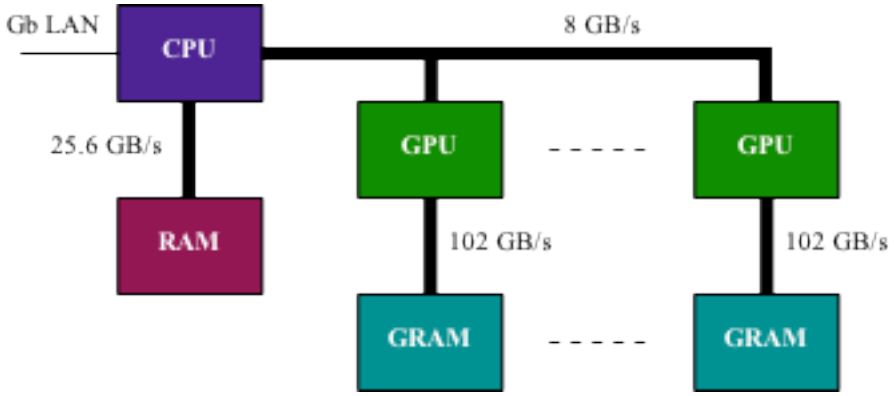
- **Moving data …?**

The World Is Going Parallel … In GPUs

CPU and GPU developments

- **A single (or dual) CPU configuration with multiple GPUs may look like the following from an architectural point of view:**



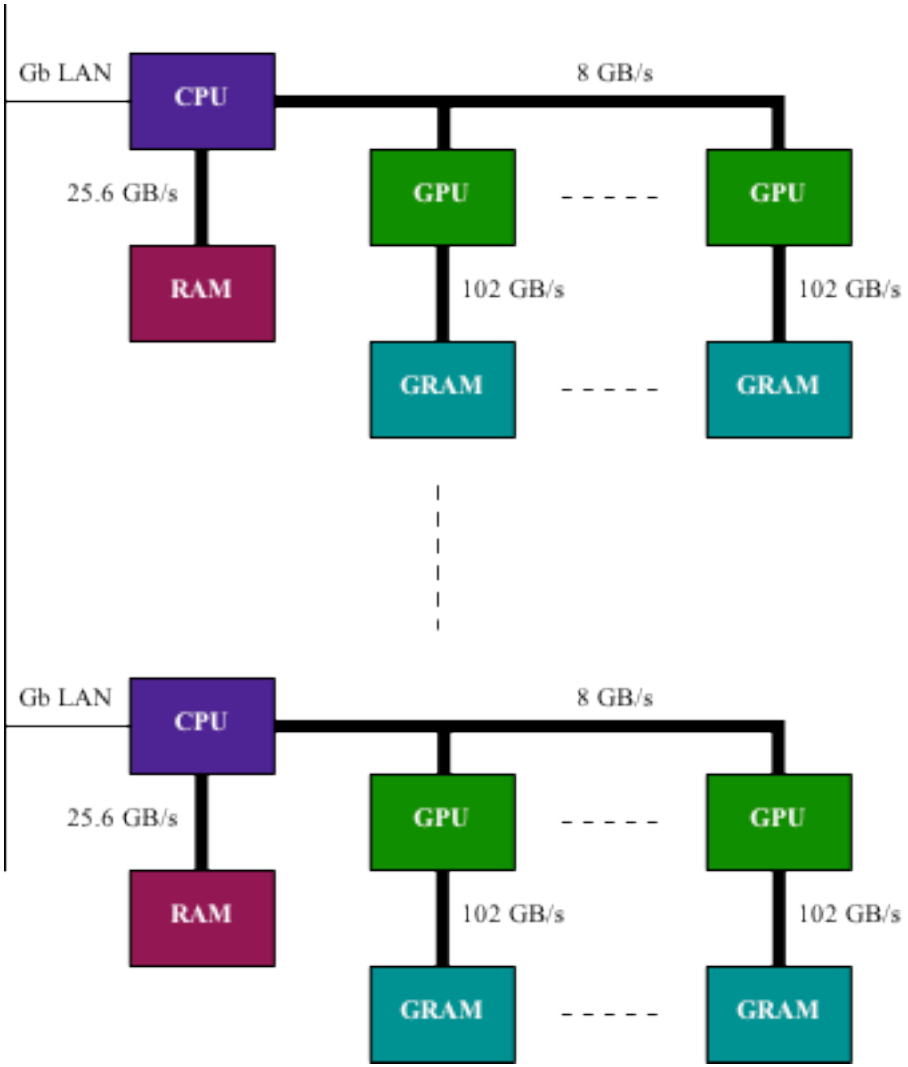


- **The PCI bus speed is way slower than CPU memory and GPU memory**
  - CPUs now use three channel memory.
  - GPUs approach 150 GB/s
  - And the PCI bus maxes out at 8 GB/s We usually measure around 0.5-1.8 GB/s for laptops and 2-3.5 GB/s for high performance workstations.
  - Faster host>device than device>host transfer

The World Is Going Parallel … In GPUs

CPU and GPU developments

- **In a cluster setup we may have the structure illustrated to the right.**



The World Is Going Parallel … In GPUs
CPU and GPU developments

- **We have a computational task which is well suited to parallel execution across multiple GPUs**
- **The challenge is that the GPU resources are heterogeneous – we focus on problems where the difference in execution time is the only issue we need to consider**
  (memory is sufficient no matter what GPU we use)

- **Overview of single precision theoretical and experimental floating point performance:**

CPU/GPU type	Performance			Power Efficiency	
	Theory [GFlops]	Measured [GFlops]	Efficiency [%]	Theory [GFlops/W]	Measured [GFlops/W]
Core i7–975	106.6	85.0	79.8	0.82	0.65
Core i7–970	153.6	122.5	79.8	1.18	0.94
Xeon X5570	93.8	88.3	94.1	0.99	0.93
Xeon X5670	140.6	00.0	00.0	1.48	00.0
GeForce GTX580	1572.9	849.2	54.0	6.45	3.48
Quadro FX3800	462.3	251.3	54.4	4.28	2.33
Quadro 2000	480.0	204.1	42.5	7.74	3.29
Quadro 4000	486.4	266.0	54.7	3.43	1.87
Tesla C1060	622.1	364.2	58.5	3.31	1.94
Tesla C2050 (ECC)	1030.4	549.8	53.4	4.33	2.31
Tesla C2050 (no ECC)	1030.4	552.0	53.6	4.33	2.32

Note that the efficiency is way higher for Intel's Nehalem like architecture than NVIDIAs GPU. For the Intel CPUs we are around 80% of theoretical performance. For the GPUs we are around 55%. However, since the fast GPUs are a factor 10 faster than the fastest CPUs we still have a significant advantage computationally wise for the GPUs.

- **Overview of double precision theoretical and experimental floating point performance**:

CPU/GPU type	Double Precision			Power Efficiency	
	Theory [GFlops]	Measured [GFlops]	Efficiency [%]	Theory [GFlops/W]	Measured [GFlops/W]
Core i7–975	53.3	48.7	91.4	0.41	0.37
Core i7–970	76.8	63.3	82.4	0.59	0.49
Xeon X5570	46.9	46.2	98.5	0.49	0.49
Xeon X5670	70.3	00.0	00.0	0.74	0.00
GeForce GTX580	393.2	193.0	49.1	1.61	0.79
Quadro FX3800	57.8	54.4	94.3	0.54	0.50
Quadro 2000	40.0	38.2	95.5	0.65	0.62
Quadro 4000	243.2	125.6	51.7	1.71	0.88
Tesla C1060	77.8	74.6	95.9	0.41	0.40
Tesla C2050 (ECC)	515.2	249.3	48.4	2.16	1.05
Tesla C2050 (no ECC)	515.2	260.9	50.7	2.16	1.10

Also for double precision the efficiency of the Nehalem like architecture is very high indeed – above 90% of theoretical performance is measured from within MATLAB. For the GPUs there is an interesting difference. The older architecture from the C1060 and FX3800 perform at 95% of theoretical performance where the Fermi architecture (used in GTX465, C2050, Quadro 2000 and 4000) is again at around 50%.

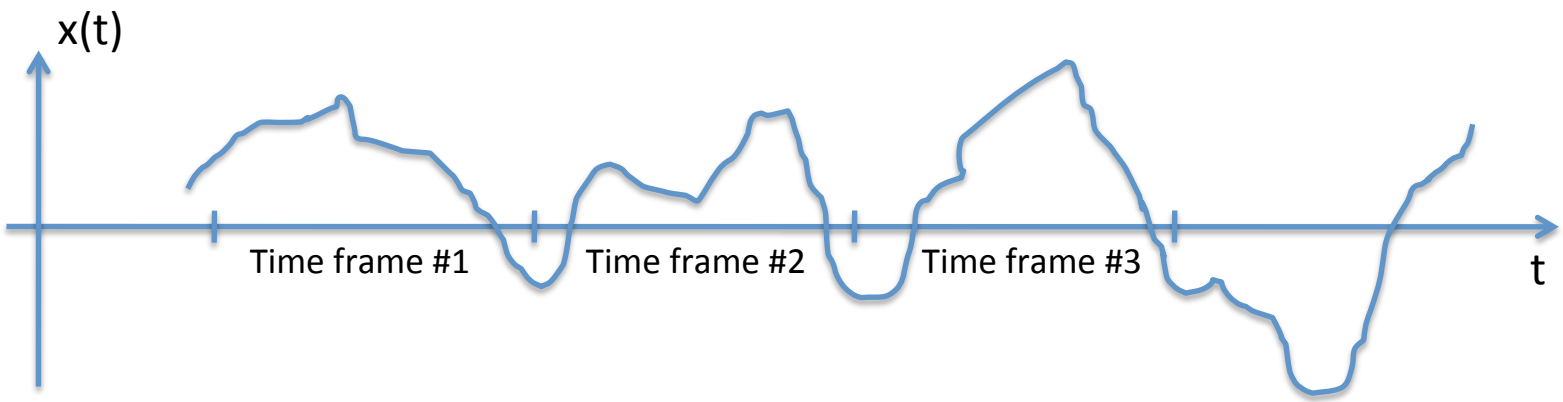
- **Computational philosophy …**

  - Why use tons of expensive engineering time to do parallel programming in low level OpenMP/CUDA/ … ?
  - Better to use slightly less efficient tools, which are much easier to use – and then take advantage of powerful libraries created by e.g. NVIDIA.
  - We want to use GPUs which are very efficient for some types of computations.

Typical computational problems in communication technology:

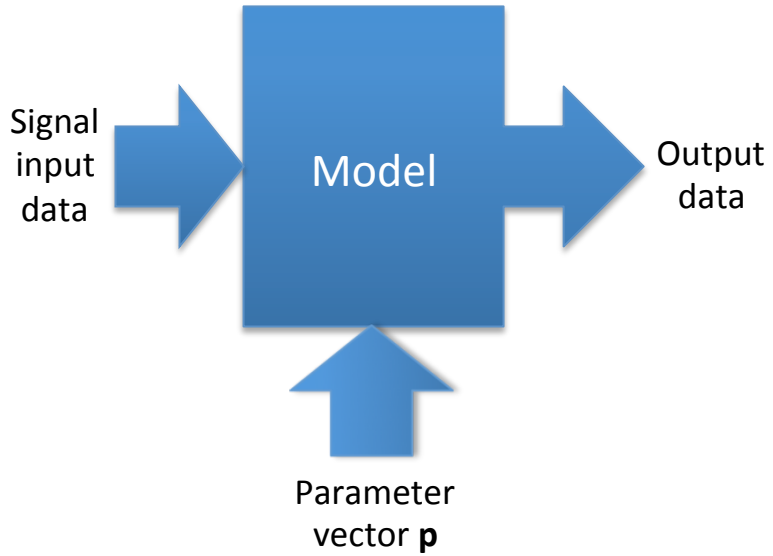
- Simulations of the same model with many different parameter sets.

- **Often analysis on very large time domain data sets may be divided into time frames:**
  - This is for example known when estimating EVM (Error Vector Magnitude) where at least 200 bursts (time frames) must be evaluated
  - It may be possible to speed up feedback based systems with dependence on earlier inputs – these can normally not be divided into time frames, but if a time overlap is used is may be possible



- **This is a an SPMD (Same Program Multiple Data) type of computation. The computation model is the same (or at least very close to the same) but the input data changes**
  - Very well suited to parallel computation
  - In particular for computation intensive tasks such that the sequential part plays a minor role
  - Even if GPU memory is insufficient for full computation it is easy to load more data

- **Simulation of the same model and input data but with different parameter vectors:**
  - Unavoidable situation in science and engineering
  - This is essentially a Monte Carlo type of simulations, which is generally extremely well suited to parallelization

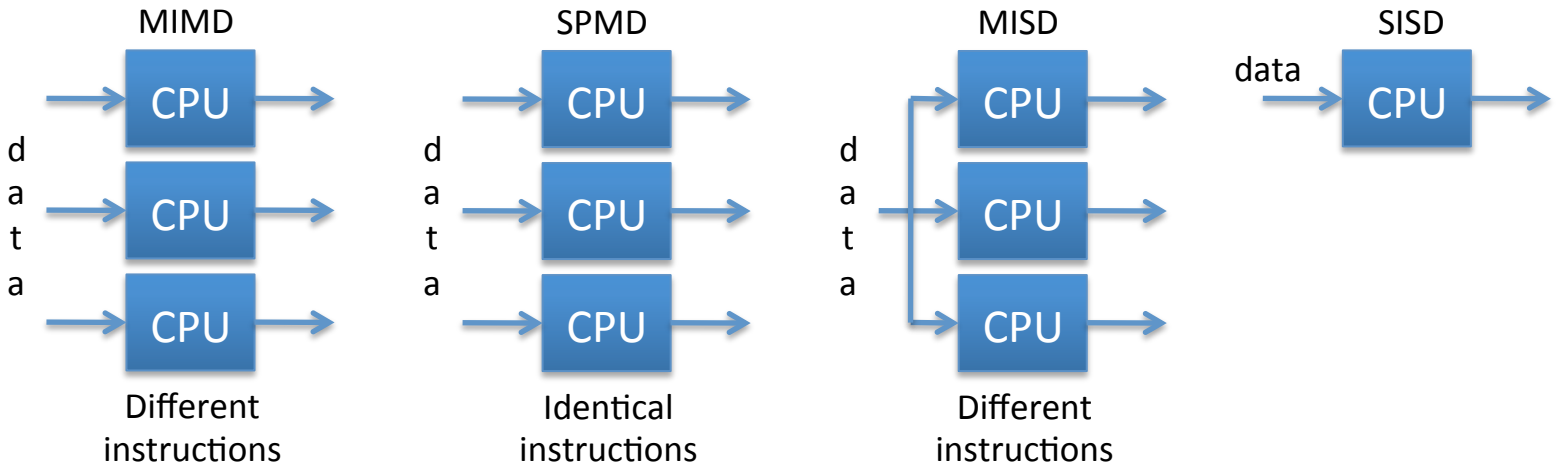


- **This is also an SPMD (Same Program Multiple Data) just in this case the input data is identical from run to run and the parameter vector is changed**
  - Extremely well suited to parallel execution as there is no sequential part in this case.

The World Is Going Parallel … In GPUs
CPU and GPU developments

- **Individual functions of huge data sets …**
  - FFTs / IFFTs.
  - Filtering.
  - SVD / QR-factorization / LU-factorization / …
  - …
- **This is much more uncertain but there may of course be situations where it makes sense.**

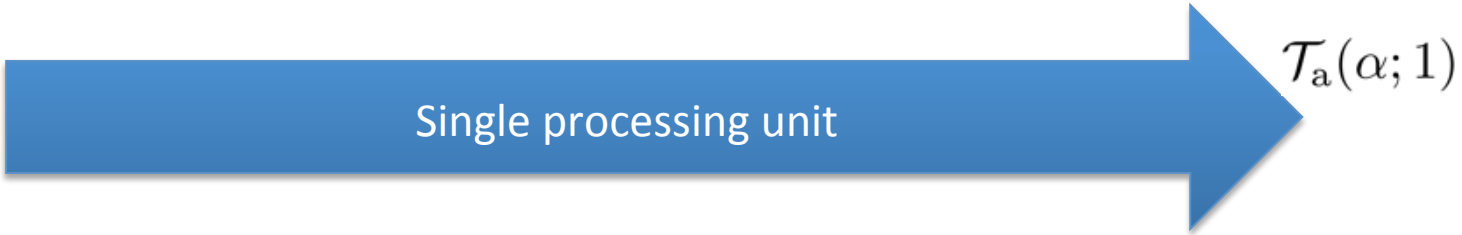
- **Different types of parallelization (Flynn's taxonomy):**
  - SISD (Single Instruction/Program, Single Data): A single processor is executing a single instruction stream on one set of data (residing in the same memory). This is the same as the von Neumann architecture.
  - SPMD (Single Program/Instruction, Multiple Data): Multiple processing elements (CPUs/GPUs) execute the same instruction stream on multiple data sets (may reside in different memories).
  - MISD (Multiple Instruction, Single Data): Multiple processing elements execute possibly different operations on the same data set.
  - MIMD (Multiple Instruction, Multiple Data): Multiple processing elements may execute possibly different operations on multiple data sets.



- In a sense SPMD is a special case of MIMD so the key parallelization techniques are **MIMD** and **MISD**.

Amdahl's Law

- **Amdahl's law says how much we can gain by going from sequential execution to parallel**
  - Say the time needed to perform a given computational task on one processing unit is



- **The problem has a certain part which must be executed sequentially and the rest in parallel:**

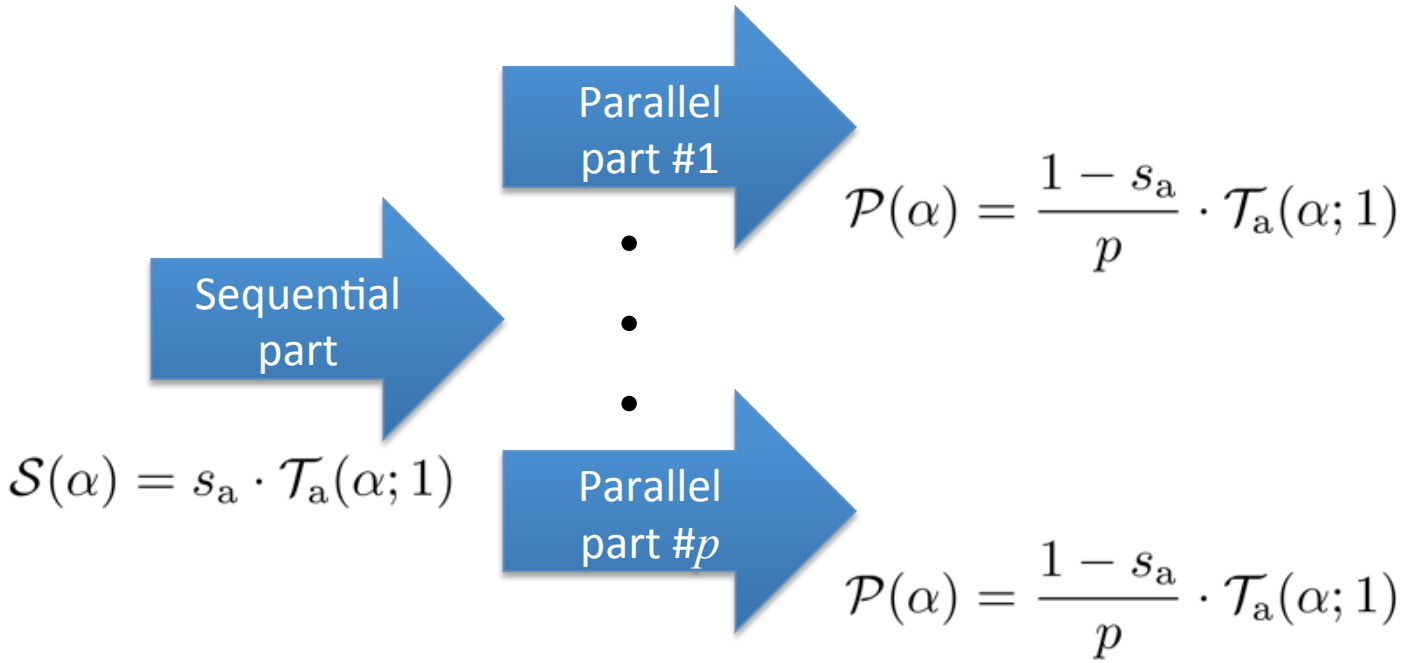
$$\mathcal{P}(\alpha) = \frac{1 - s_{\mathrm{a}}}{p} \cdot \mathcal{T}_{\mathrm{a}}(\alpha;1)$$

$$\mathcal{S}(\alpha) = s_{\mathrm{a}} \cdot \mathcal{T}_{\mathrm{a}}(\alpha;1)$$

$$\mathcal{P}(\alpha) = \frac{1 - s_{\mathrm{a}}}{p} \cdot \mathcal{T}_{\mathrm{a}}(\alpha;1)$$

Amdahl's Law

- **The problem has a certain part which must be executed sequentially and the rest in parallel:**



- **The total execution time for p processing units is then:**

$$\mathcal{T}_a(\alpha; p) = s_a \cdot \mathcal{T}_a(\alpha; 1) + \frac{1 - s_a}{p} \cdot \mathcal{T}_a(\alpha; 1)$$

- **We assume that the problem size is unchanged by parallelization – this is not always the case.**

- **The speed-up factor of using multiple versus only one computational unit is then:**

$$S_{\mathrm{a}}(s_{\mathrm{a}};p) = \frac{\mathcal{T}_{\mathrm{a}}(\alpha;1)}{\mathcal{T}_{\mathrm{a}}(\alpha;p)} = \frac{\mathcal{T}_{\mathrm{a}}(\alpha;1)}{s_{\mathrm{a}} \cdot \mathcal{T}_{\mathrm{a}}(\alpha;1) + \frac{1-s_{\mathrm{a}}}{p} \cdot \mathcal{T}_{\mathrm{a}}(\alpha;1)}$$

- **Rearranging a bit leads to:**

$$S_{\mathrm{a}}(s_{\mathrm{a}};p) = \frac{p}{s_{\mathrm{a}} \cdot (p-1) + 1}$$

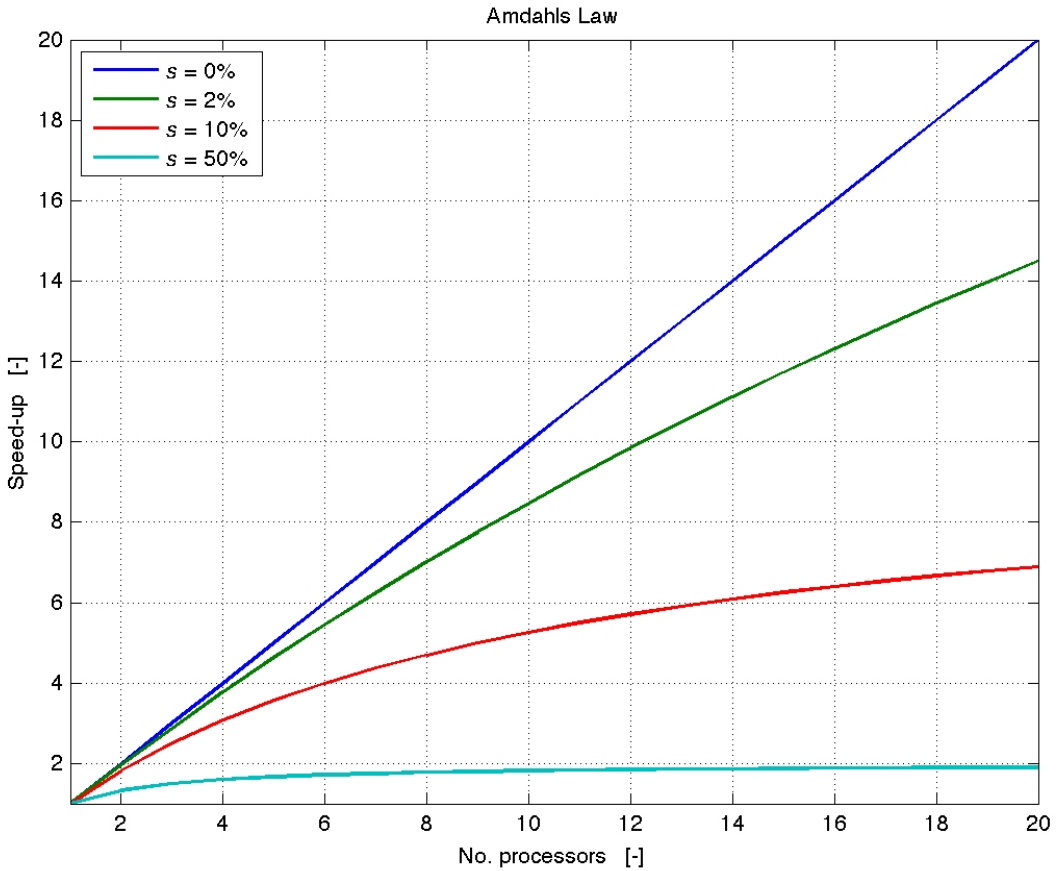
- All this assumes perfect scaling and utilization of the processing units. **An important number to find is the speed-up, which can be achieved by infinitely many processing units:**

$$\lim_{p \to \infty} S_{\mathrm{a}}(s_{\mathrm{a}};p) = \frac{1}{s_{\mathrm{a}}}$$

- **This extremely important result says that the speed-up by using many computational units can never be higher than one divided by the sequential part – or in other words; the sequential part determines how well parallel execution works**

Amdahl's Law

- **Behavior of the speed-up for various combinations of percentage sequential computation and number of processing units:**

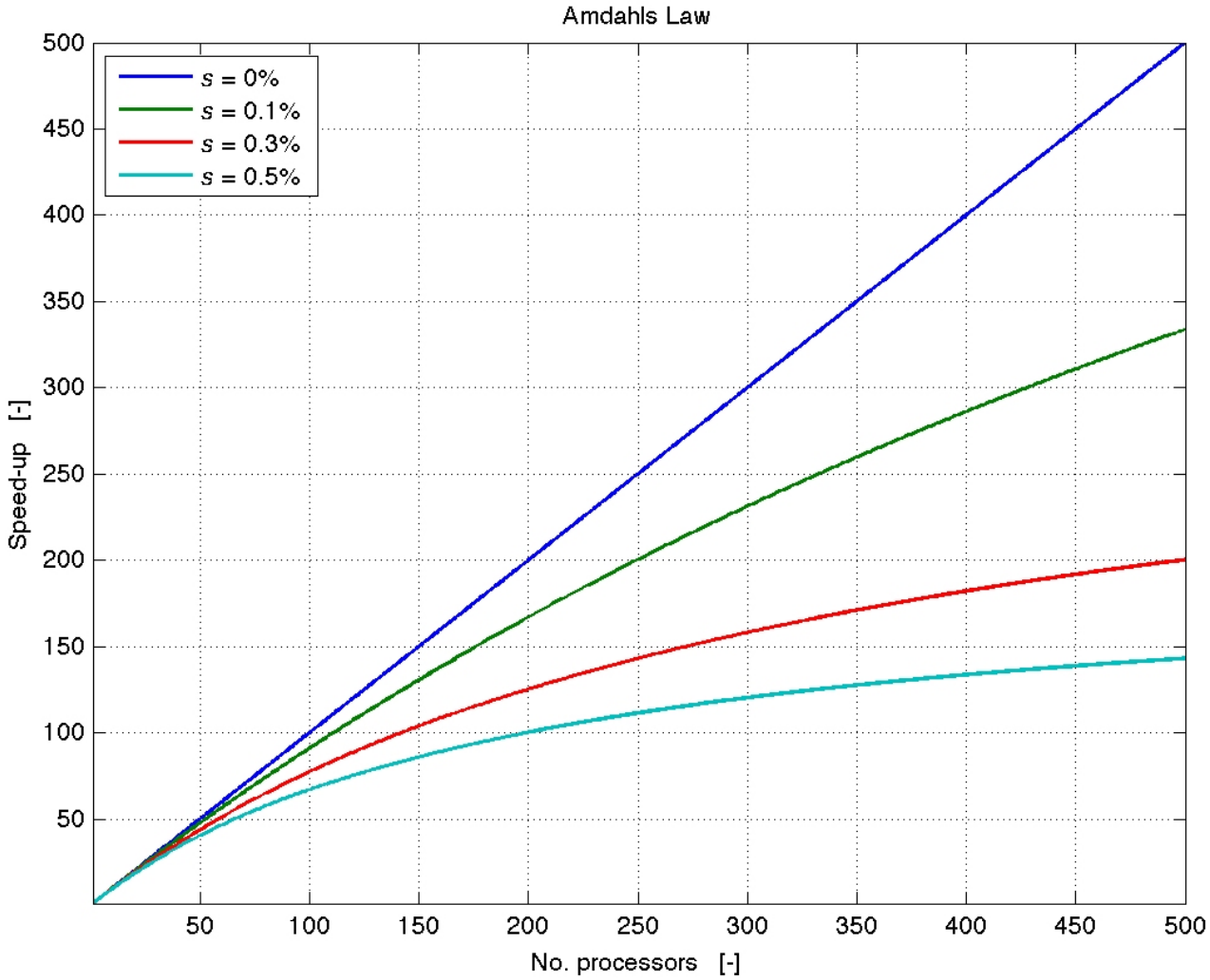


For *s*=2% the speed-up converges at 50 meaning that there is a good speed-up for the number of processors shown in the figure.

For *s*=10% the speed-up converges at 10. The speed-up by going from 10 to 20 processors increases from approximately 5 to 7.

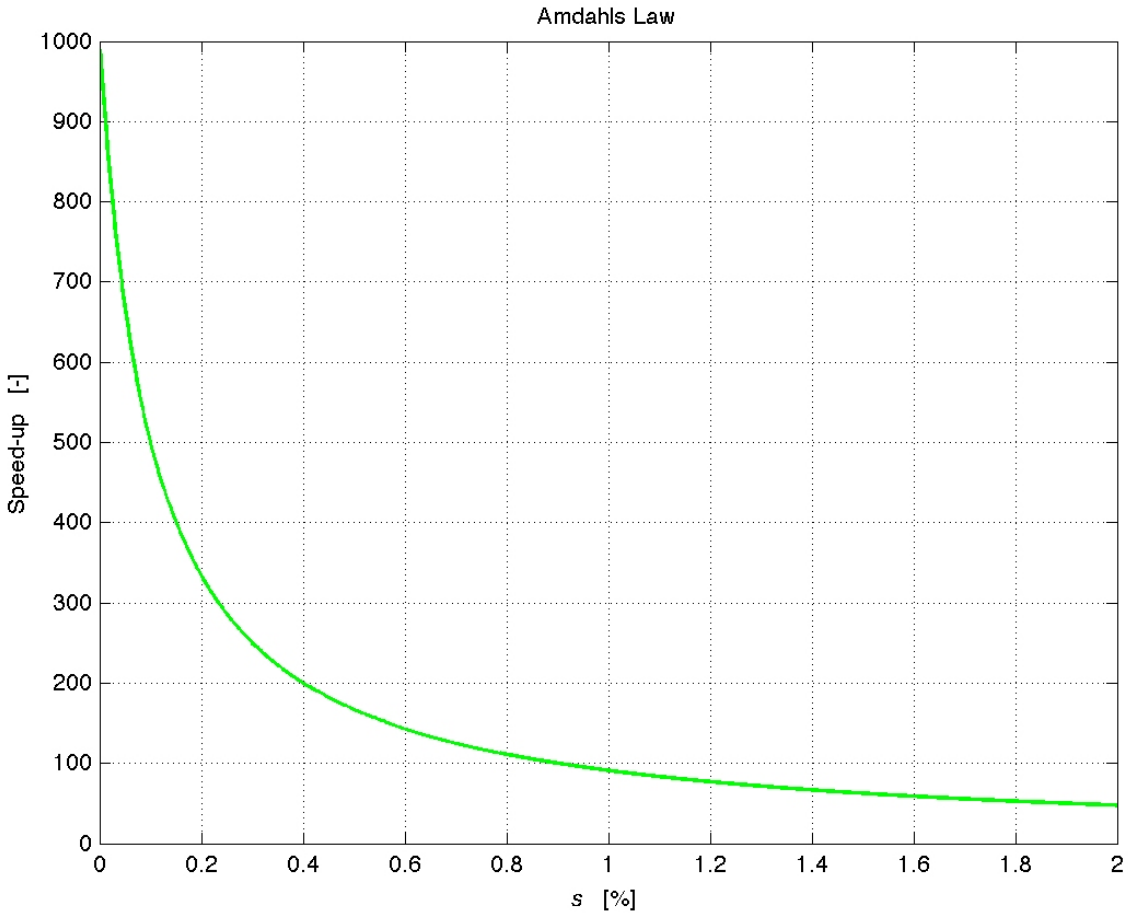
For s=50% the speed-up converges to

Amdahl's Law


Amdahls Law
s = 0%
s = 0.1%
s = 0.3%
s = 0.5%
Speed-up [-]
No. processors [-]
50
100
150
200
250
300
350
400
450
500

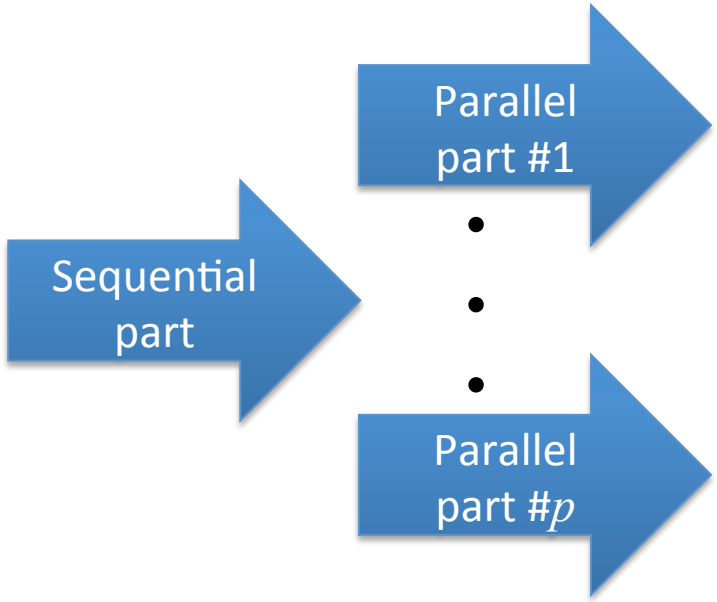
Amdahl's Law

- Speedup versus serial part of the code:



The speedup is very sensitive to s changes when s is below 1%. The figure uses p=1000 processing units.

- **So if we have a parallel computer with p computing units we have the situation:**



- **The execution time is then:**

$$\text{Parallel computer: } \mathcal{S}(\alpha) + \mathcal{P}(\alpha) = \mathcal{T}_{\text{gb}}(\alpha; p)$$

Sequential part — **Parallel part** — **Problem size** — **# computing units available**

The World Is Going Parallel … In GPUs
Gustafson-Barsis' Law

- **With a purely sequential computer we have the situation:**

- **The execution time is then:**

$$\text{Sequential computer: } \mathcal{S}(\alpha) + p \cdot \mathcal{P}(\alpha) = \mathcal{T}_{\text{gb}}(\alpha; 1)$$

Sequential part — **Parallel part** — **Problem size** — **# computing units available**

- **The speed-up we achieve by having p computing units is then:**

$$S_{\mathrm{gb}}(p) = \frac{\mathcal{T}_{\mathrm{gb}}(\alpha;1)}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)} = \frac{\mathcal{T}_{\mathrm{gb}}(\alpha;p) + (p-1)\cdot\mathcal{P}(\alpha)}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)}$$

- **Rewriting a bit leads to:**

$$\begin{aligned} S_{\mathrm{gb}}(p) &= \frac{\mathcal{T}_{\mathrm{gb}}(\alpha;p) + (p-1)\cdot\{\mathcal{T}_{\mathrm{gb}}(\alpha;p) - \mathcal{S}(\alpha)\}}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)} \\ &= \frac{p\cdot\mathcal{T}_{\mathrm{gb}}(\alpha;p) + (1-p)\cdot\mathcal{S}(\alpha)}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)} \\ &= p + (1-p)\cdot\frac{\mathcal{S}(\alpha)}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)} \end{aligned}$$

- **The latter fraction is the relative sequential to total time for p units parallel processing:**

$$S(s_{\mathrm{gb}};p) = p + (1-p)\cdot s_{\mathrm{gb}}, \qquad s_{\mathrm{gb}} = \frac{\mathcal{S}(\alpha)}{\mathcal{T}_{\mathrm{gb}}(\alpha;p)}$$

Gustafson-Barsis' Law

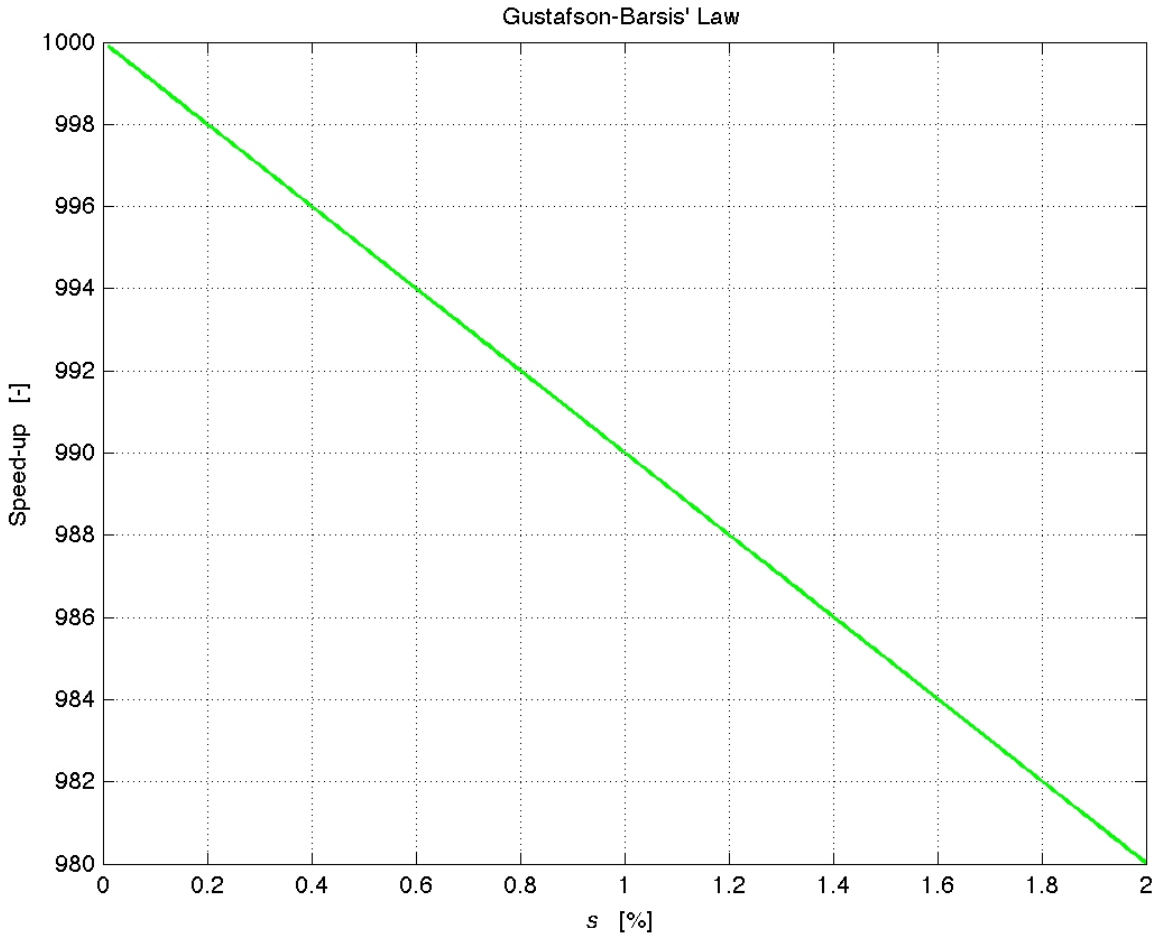
- **As the number of processing units p increases we see:**

$$\lim_{p \to \infty} S_{\mathrm{gb}}(s_{\mathrm{gb}}; p) = p \qquad \text{for} \qquad 0 \leq s_{\mathrm{gb}} < 1$$

- **Unlike for Amdahls law we here have a linear dependence of the speed-up versus s, and the speed-up converges to the number of processing units.**

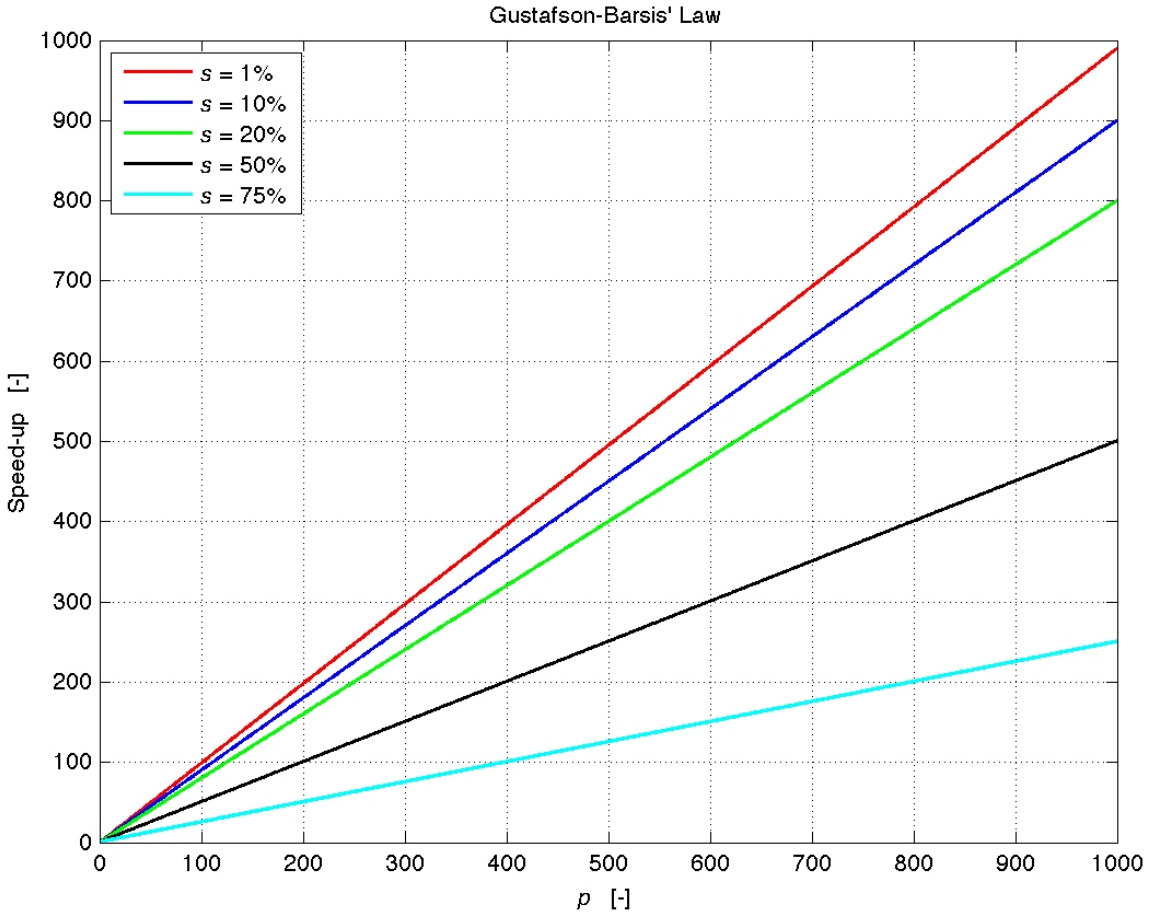
Gustafson-Barsis' Law

- **Speed-up versus ratio of sequential part for 1000 processing units:**



Gustafson-Barsis' Law

- **Speed-up versus number of processors for different sequential ratios:**



Comparison of Amdahls and Gustafson-Barsis' Laws

- So it may seem as if we get two different results depending on how we observe the problem … we don't though.
  - But … observe that for identical problem sizes α the sequential percentages are not the same – i.e. s_{a} and s_{gb} are different.
  - So for a specific case we will get the same speed-up with the two methods.
  - The general observations says something different though (at least apparently).
- **Amdahls Law basically says:**
  - Focus: solving a problem of specific size.
  - If we have a problem of a certain size α and a certain part of this must be run sequentially then the speed-up can never be higher than $1/s_{\mathrm{a}}$ - in the practical world many problems are so that s_{a} goes down if the problem size increases. Thereby, Amdahls law may be overly pessimistic when drawing general conclusions.
- **Gustafson-Barsis' Law says:**
  - Focus: solve as large a problem as at all possible.
  - For large problems it always pays off to use parallelism (if possible at all) – even if a sequential part has taken some time. As long as the problem is large enough the speed-up can be p with p being the number of processing units.
- Comments:
  - Be careful not to put too much in what is happening in the limit ($p \rightarrow \infty$). It is the specific cases we should be most aware of. Obviously we gain most when the problem scales in size – but see what reality is and consider that.

Comparison of Amdahls and Gustafson-Barsis' Laws

- So the question is how the forced sequential part of the problem increase with problem size:

- **Amdahl:**
  - Basically has as starting point that the sequential part scales linearly with problem size.

- **Gustafson-Barsis' Law:**
  - Basically has as starting point that the sequential part scales slowly with problem size.

- Solution:
  - Be aware of both laws and see what your problem is like.
  - How do your problem scale with size.
  - Do you want to increase the problem as much as possible or is your problem in nature limited in size.
  - Compute the possible speed-up you may have for different problem sizes.
  - Consider how s_{a} and s_{gb} varies with problem size.

Comparison of Amdahls and Gustafson-Barsis' Laws

- Let's take a small example. Suppose we have a situation where:
  - We have 10 processing units.
  - The sequential part takes 10 seconds.
  - The parallel part is such that one parallel unit needs 250 seconds to compute the that part alone.

- Amdahls law:
  - The total sequential execution time is $T = 260$ seconds for one processing unit alone.
  - This means that $s_a = 10/260 = 3.8462\ \%$.
  - By parallel execution it takes a) $sT = 10/260*260 = 10$ seconds sequential execution time, and b) $(1\text{-}s)T/p = 25$ seconds. This is a total of 35 seconds.
  - The speed-up according to the derived equation is 7.4286.

- Gustafson-Barsis' law:
  - The parallel part takes $T/p = 250/10 = 25$ seconds. Thereby, $s_{gb} = 10/(25+10) = 28.5714\ \%$.
  - By use of the derived equation, the speed-up can be computed to 7.4286.

- Comments:
  - The speed-up is of course the same for the two cases but the relative times for the sequential part is different depending on what approach we use.

The World Is Going Parallel … In GPUs
Conclusions and Trends

- So to sum up …
  - Parallelization is the way forward for HPC (High Performance Computing) – while using reasonable power.
  - Problems must be suited for this type of paradigm.
  - GPGPUs (General Purpose Graphics Processing Units) outperform even modern CPUs by an order of magnitude in raw floating point performance.

- Characteristics particular for GPGPUs:
  - Many cores (typically 16-500).
  - Excellent single precision floating point performance – and double precision is following in modern processors.
  - GPGPU memory typically around 128 MB – 6 GB. True computationally intended GPGPUs have 3-6 GB of memory.
  - PCI bus connected with max. 8 GB/s transfer rates (in real life typically 2-3 GB/s) – this is a bottleneck and an area, which needs improvement.
  - CUDA / OpenMP / … require significant programming skills. It takes many, many months to really master it.

- The main challenge: The programming environment!

The World Is Going Parallel … In GPUs
Conclusions and Trends

- Hardware:
  - Power efficiency; even our small cluster consumes around 15kW when running full speed. Add cooling.
  - Many cores; Power consumption scales with frequency and memory access can't keep up with the computational core.
  - Currently we can have as many computations as we like. Memory access is the main issue. Often difficult to keep the computational engine busy.
  - Haque and Pande from Stanford reported in 2010 the following: soft (random) memory errors do happen in consumer grade GPUs in particular. Take care of data integrity.
- Software:
  - Maintenance of large software packages is extremely expensive; and porting it to new hardware architectures even more so.
  - Clear trends to aim for high abstraction level programming; MATLAB, Matematica etc. where multi-core possibilities are emerging. Easy access to parallel computing tools is seen as the major obstacle to overcome.
  - Clear trend just to add more computational nodes to combat the loss, which high abstraction level programming may cost.

GPU Platforms

GPU Platforms

- When considering Jacket platforms there are a bunch of things to consider
  - **Laptops:** These are usually used for development of code and rarely to do any production runs. Take a pretty fast CPU/GPU combination with plenty of memory. Double precision is practically impossible here – unless the laptop is super heavy.
  - **Desktop systems:** These are generally computers which can handle one or max 2 GPUs. Focus is on low noise and low thermal footprint.
  - **Deskside systems:** Here we move to powerful computers with typically 2-4 GPUs, which are usually able to handle double precision computations.
  - **Clusters:** The best of the best with typically a login node and 3-100 compute nodes – each with up to 3-4 GPUs. Often a fast InfiniBand is connecting all nodes to allow fast memory access across nodes.
- General remarks:
  - Use Linux whenever possible – it is more stable and reliable than Microsoft Windows (also Windows 7) and Mac OSX can only run on Mac computers, which are too expensive for this application area.
  - The amount of CPU memory should be at least 4 GB per CPU plus the accumulated amount of GPU memory. So a two CPU system (e.g. Colfax CXT5000) with 4 Tesla C1060 would give at least 2 x 4 GB + 4 x 4 GB = 24 GB of CPU memory. If many users then more memory should be available.
  - In a cluster configuration, choose as much CPU memory as money and hardware allows – for example 192 GB memory. Here there are bound to be many users and jobs, and it will handle large problems – the more memory the better.
  - A 4 GPU system with 2 CPUs likely consumes at least 1 kW – enough to increase the room temperature quite a bit and it sounds like a 747 at take-off. Better place this in a room for the purpose.

GPU Platforms

- What GPU? The best money allows but there are some considerations
  - Single/double precision? The NVIDIA GeForce series is cheap and it is fast in single precision but relatively slow in double precision and it has limited memory. Also it has not been designed for 24-7 operation so probably not the choice for a GPU cluster running day and night – see [5] on soft errors.
  - Check the internet for benchmarks of various GPUs – one example could be:

    http://wiki.accelereyes.com/wiki/index.php/Torben's_Corner

- Balance between CPU and GPU
  - There should be a reasonable balance between CPU and GPU. It doesn't make sense to have the latest Tesla C2070 and a cheap Core 2 Duo. This CPU may have trouble feeding the GPU with data. But the other way around is equally bad.

GPU Platforms

- Avoid transferring data between CPU and GPU memory as much as possible
  - this goes via the PCI bus which is operating at perhaps 2-5% of the speed of the GPU memory access. If possible do more computations on the GPU – this often is much more efficient.
- Use single precision whenever possible – there is a huge speed improvement to get here
  - For Intel CPUs the speed-up is a factor 2 and it is at least the same for NVIDIA GPUs.
  - Always handle matrix inversion, high order filters etc. in double precision – otherwise expect accuracy to suffer.
- Vectorize code.
  - Often it is possible to rewrite loops to MATLAB vectorized code. This also helps Jacket.
  - Check BSXFUN – a grossly overlooked function in MATLAB, which has potential for huge speed improvements. Only downside is that it is heavy on memory.

GPU Platforms

- State of the art GPUs:

Tesla C2070:

- Highest double precision computational power available (~0.5 TFlops).
- 6 GB memory.
- ECC memory (including registers and cache).

GeForce GTX580:

- Highest single precision computational power available (~1.5 TFlops).
- 1.5-3 GB memory.

Quadro 4000:

- Quite decent performance in both single and double precision.
- Relatively low power.

GPU Platforms

- Some hardware platforms:

Colfax CXT8000:

- Dual Intel Xeon CPU possibility.
- Up to 144 GB memory.
- Up to 8 x NVIDIA Tesla C2070.
- Up to 3 (2 min) 1200W PSUs.
- Up to 2 internal 2.5” hard disks.

Performance:

- 8 x C2070 >> 4 TFlops double precision.
- 8 x GTX580 >> 13 TFlops single precision.

- … but expect to loose two PCI slots: 1 for hardware RAID, and 1 for InfiniBand (unless it is a stand alone unit).

Floating Point Representation

- **Numbers are written as:**

$$x_b = x_a + \epsilon, \qquad a, b \in \mathcal{Z} \wedge a < b$$

$$x_{decimal} = (-1)^s \cdot c \cdot b^q \qquad s, c, q \in \mathcal{Z} \qquad (1)$$

$$x_{binary} = (-1)^s \cdot c \cdot b^q \qquad s, c, q \in \mathcal{Z} \qquad (2)$$

- Variables:
  - b must be an integer belonging to the set $b \in \{2,10\}$.
  - s must be an integer belonging to the set $s \in \{0,1\}$. s=0 for positive numbers, and s=1 for negative numbers.
  - c must be an integer belonging to the set $c \in \{0,1,\ldots,b^p\text{-}1\} = \{0,1,\ldots,c_{\max}\}$.
  - q must be an integer complying with the inequality $1\text{-}e_{\max} \leq q\text{+}p\text{-}1 \leq e_{\max}$. This translates to a constraint on q as belonging to the set $q \in \{2\text{-}e_{\max}\text{-}p,\ 3\text{-}e_{\max}\text{-}p\ ,\ldots,1\text{+}e_{\max}\text{-}p\}$.

- IEEE 754 floating point representation:

IEEE type	Base b	p [bits]	e_{max} [—]
Decimal-64	10	16	+384
Decimal-128	10	34	+6144
Binary-16	2	$10+1$	+15
Binary-32	2	$23+1$	+127
Binary-64	2	$52+1$	+1023
Binary-128	2	$112+1$	+16383

- Let's assume we have a functional input-output relation for some sensor as:

$$\begin{aligned} y &= \sum_{\ell=0}^{L} a_\ell \cdot x^\ell && (1) \\ &= a_0 + a_1 \cdot x + \cdots + a_{L-1} \cdot x^{L-1} + a_L \cdot x^L && (2) \\ &= (\cdots((x + a_L) \cdot x + a_{L-1}) \cdot x + \cdots) \cdot x + a_0 && (3) \end{aligned}$$

- where:

$$\text{sgn}\{a_\ell\} = \begin{cases} +1 & \text{for } \ell \text{ even} \\ -1 & \text{else} \end{cases} \qquad (1)$$

- Numerically there is huge difference between y in (2) and y in (3):

$$\text{Suppose } |a_\ell| = 9.99 \text{ for } \ell \in \{1, 2, \ldots, L\}, \text{ and that } x = 10, \text{ and } L = 5$$

- Using Binary-16 means a max. of 6.55E4, and with x=10 leads to y from (2) as:

$$\begin{array}{lllll}
\text{Step 1:} & a_0 + a_1 \cdot x & = & 9.99 - 9.99 \cdot 10 & \rightarrow \quad x_1 = -8.99 \cdot 10^1 \\
\text{Step 2:} & x_1 + a_2 \cdot x^2 & = & x_1 + 9.99 \cdot 10^2 & \rightarrow \quad x_2 = +9.09 \cdot 10^2 \\
\text{Step 3:} & x_2 + a_3 \cdot x^3 & = & x_2 - 9.99 \cdot 10^3 & \rightarrow \quad x_3 = -9.08 \cdot 10^3 \\
\text{Step 4:} & x_3 + a_4 \cdot x^4 & = & x_3 + 6.55 \cdot 10^4 & \rightarrow \quad x_4 = +5.64 \cdot 10^4 \\
\text{Step 5:} & x_4 + a_5 \cdot x^5 & = & x_4 - 6.55 \cdot 10^4 & \rightarrow \quad y = -9.08 \cdot 10^3
\end{array}$$

- Using Binary-16 means a max. of 6.55E4, and with x=10 leads to y from (3) as:

$$\begin{array}{lllll}
\text{Step 1:} & x + a_5 & = & 10 - 9.99 & \rightarrow \quad x_5 = +1.00 \cdot 10^{-2} \\
\text{Step 2:} & x_5 \cdot x + a_4 & = & x_5 \cdot 10 + 9.99 & \rightarrow \quad x_4 = +1.01 \cdot 10^1 \\
\text{Step 3:} & x_4 \cdot x + a_3 & = & x_4 \cdot 10 - 9.99 & \rightarrow \quad x_3 = +9.09 \cdot 10^1 \\
\text{Step 4:} & x_3 \cdot x + a_2 & = & x_3 \cdot 10 + 9.99 & \rightarrow \quad x_2 = +9.19 \cdot 10^2 \\
\text{Step 5:} & x_2 \cdot x + a_1 & = & x_2 \cdot 10 - 9.99 & \rightarrow \quad y = +9.18 \cdot 10^3
\end{array}$$

- Different IEEE floating point representations:

IEEE type	Coeff. $c_{max}+1$	Exp. q	Largest pos./neg.	Resolution (smallest non-neg.)	# sign. dec. digits
Decimal-64	10^{16}	$-398 \rightarrow 369$	$\pm 9.99 \cdot 10^{384}$	$1 \cdot 10^{-398}$	16
Decimal-128	10^{34}	$-6176 \rightarrow 6111$	$\pm 9.99 \cdot 10^{6144}$	$1 \cdot 10^{-6176}$	64
Binary-16†	2^{11}	$-24 \rightarrow 5$	$\pm 6.55 \cdot 10^{4}$	$5.96 \cdot 10^{-8}$	3.3
Binary-32	2^{24}	$-149 \rightarrow 104$	$\pm 3.40 \cdot 10^{38}$	$1.40 \cdot 10^{-45}$	7.2
Binary-64	2^{53}	$-1074 \rightarrow 971$	$\pm 1.80 \cdot 10^{308}$	$4.94 \cdot 10^{-324}$	16.0

MATLAB
Vectorization

- So what is vectorization ...?
- Well vectorization is the opposite of scalar operations on data. An example of performing a computation as scalar is to do for example:

```
a = randn(N,1);  S=0;
for n=1:N
  S = S + a(n)
end
```

$$S = \sum_{n=1}^{N} a_n$$

```
a = randn(N,1);
S = sum(a(:));
```

```
a =randn(N,1);
b = randn(N,1);
S = 0;
for n=1:N
  S = S + a(n)*b(n)
end
```

$$S = \sum_{n=1}^{N} a_n \, b_n$$

```
a = randn(N,1);
b = randn(N,1);
S = sum(a.*b);
```

- Let's consider the problem where we have two matrices representing N points in space:

$$\mathbf{a} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ a_{2,1} & \cdots & a_{2,N} \\ a_{3,1} & \cdots & a_{3,N} \end{bmatrix} = [\mathbf{a}_1, \ldots, \mathbf{a}_N]^{\mathrm{T}} \quad \in \quad \mathcal{R}^{3 \times N}$$

$$\mathbf{b} = \begin{bmatrix} b_{1,1} & \cdots & b_{1,N} \\ b_{2,1} & \cdots & b_{2,N} \\ b_{3,1} & \cdots & b_{3,N} \end{bmatrix} = [\mathbf{b}_1, \ldots, \mathbf{b}_N]^{\mathrm{T}} \quad \in \quad \mathcal{R}^{3 \times N}$$

- We want to determine the Euclidian distance between two arbitrary points in space represented by:

$$\mathbf{E} = \begin{bmatrix} e_{1,1} & \cdots & e_{1,N} \\ \vdots & & \vdots \\ e_{N,1} & \cdots & e_{N,N} \end{bmatrix} \quad \in \quad \mathcal{R}_{0+}^{N \times N}$$

$$e_{k,\ell} = \sqrt{|a_{1,k} - b_{1,\ell}|^2 + |a_{2,k} - b_{2,\ell}|^2 + |a_{3,k} - b_{3,\ell}|^2}$$

- A first MATLAB implementation of this might look like:

```
%% VERSION 1

% Number of data points
N = 3000;

% Generate data and pre-allocate result matrix
a = randn(3,N);
b = randn(3,N);
E1 = zeros(N,N);

% Loop based solution
ts = tic;
for k=1:N
  for ell=1:N
    E1(k,ell) = norm(a(:,k) - b(:,ell));
  end
end
t1 = toc(ts)
```

On a MacBook Air 2011 (Core 2 Duo 2.13 GHz & 4 GB memory) this takes 21.8 seconds.

Norm is not supported by MATLABs JIT (Just-In-Time compilation) and the implementation is therefore quite slow.

- In the second version let's expand norm by arithmetic, which is supported by MATLABs JIT:

```
%% VERSION 2

% Number of data points
N = 3000;

% Generate data and pre-allocate result matrix
a = randn(3,N);
b = randn(3,N);
E2 = zeros(N,N);

% Loop based solution
ts = tic;
for k=1:N
  for ell=1:N
    E2(k,ell) = sqrt( (a(1,k)-b(1,ell)).^2 ...
                + (a(2,k)-b(2,ell)).^2 ...
                + (a(3,k)-b(3,ell)).^2 );
  end
end
t = toc(ts)
```

On a MacBook Air 2011 (Core 2 Duo 2.13 GHz & 4 GB memory) this takes 0.978 seconds. This is a speed-up compared to version 1 of 22.3 times.

The reason for the much faster execution time of version 2 compared to version 1 is that **NORM** used in version 1 is not supported by the MATLAB JIT (Just-In-Time) compiler – and the arithmetic variant in version 2 is.

- In version 3 we will try to vectorize the code – i.e. remove the two nested loops:

```
%% VERSION 3

% Number of data points
N = 3000;

% Generate data and pre-allocate result matrix
a = randn(3,N);
b = randn(3,N);

% Loop based solution
ts = tic;
E1t = bsxfun(@minus, a(1,:)', b(1,:));
E2t = bsxfun(@minus, a(2,:)', b(2,:));
E3t = bsxfun(@minus, a(3,:)', b(3,:));
E3 = sqrt( E1t.^2 + E2t.^2 + E3t.^2 );
t = toc(ts)
```

On a MacBook Air 2011 (Core 2 Duo 2.13 GHz & 4 GB memory) this takes 0.325 seconds. This is a speed-up compared to version 1 of 61 times, and it is 3 times faster than version 2.

Here we use **BSXFUN** to generate three matrices each containing the differences of all points. The MATLAB function **BSXFUN** is a clever one as it automatically expands incompatible size arrays.

A Jacket version based on the exact same methodology is close to 8 times faster when doing all tests on a dual Intel Xeon X5570 with 48 GB memory and an NVIDIA Tesla C2070. The MacBook Air has insufficient memory to do the test.

- Let's look at one of the **BSXFUN**'s:

```
E1t = bsxfun(@minus, a(1,:)', b(1,:));
```

- BSXFUN is so clever that it automatically expands the rows/columns of the input arrays such that the minus operation can be used. The result of the BSXFUN is:

$$\mathbf{E1t} = \begin{bmatrix} a_{1,1} & a_{1,1} & \cdots & a_{1,1} \\ a_{1,2} & a_{1,2} & \cdots & a_{1,2} \\ \vdots & \vdots & & \vdots \\ a_{1,N} & a_{1,N} & \cdots & a_{1,N} \end{bmatrix} - \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,N} \\ b_{1,1} & b_{1,2} & \cdots & b_{1,N} \\ \vdots & \vdots & & \vdots \\ b_{1,1} & b_{1,2} & \cdots & b_{1,N} \end{bmatrix}$$

- This means that a specific element of the **E1t** matrix is given by:

$$\mathbf{E1t}(k,\ell) = \mathbf{a}(1,k) - \mathbf{b}(1,\ell)$$

- BSXFUN can often lead to very respectable speed-up's but there is one main concern, which is also obvious from the information above. **It uses lots of memory because it expands the input arrays**.

- Considering further what BSXFUN is actually doing in the command:

```
E1t = bsxfun(@minus, a(1,:)', b(1,:));
```

- The key issue is that BSXFUN allows normally incompatible arrays to be put together via normal arithmetic operations (plus, minus etc.) by expanding the arrays.
- To use a smaller vector size, say we have:

```
F = randn(2,1);     % Column vector
G = (1:2);          % Row vector
```

- MATLAB is doing the following:

```
>> bsxfun(@plus, F, G)

ans =

    0.6786    1.6786
   -1.1046   -0.1046
```

```
>> F_ = repmat(F, 1, 2);
>> G_ = repmat(G, 2, 1);
>> F_ + G_

ans =

    0.6786    1.6786
   -1.1046   -0.1046
```

The two methods provides the same result but BSXFUN is faster and can also handle much more complicated cases than the one shown here.

- We will in the following look at an estimation used in an optimization problem. Our input is two vectors **r** (reference) and **t** (test) and we need to compute:

$$x_3 = \frac{-2}{N\left(N-1\right)} \sum_{n=1}^{N-1} \sum_{\Delta_n=1}^{N-n} \frac{1}{\Delta_n} \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(n+\Delta_n)}{t(n+\Delta_n)} \right| \right\}$$

- And the input data is organized as:

$$\mathbf{r} = \begin{bmatrix} r(1) \\ \vdots \\ r(N) \end{bmatrix} \in \mathcal{C}^{N \times 1} \qquad \mathbf{t} = \begin{bmatrix} t(1) \\ \vdots \\ t(N) \end{bmatrix} \in \mathcal{C}^{N \times 1}$$

- To understand better what is going on let's define a couple of intermediate variables:

$$\kappa = \frac{-2}{N\left(N-1\right)} \qquad f(n,\Delta_n) = \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(n+\Delta_n)}{t(n+\Delta_n)} \right| \right\}$$

$$\Rightarrow \quad x_3 = \kappa \sum_{n=1}^{N-1} \sum_{\Delta_n=1}^{N-n} \frac{1}{\Delta_n} f(n,\Delta_n)$$

- If we now write out the sums we get:

$$\begin{aligned}\frac{x_3}{\kappa} = & \frac{1}{1}f(1,1) + \cdots + \frac{1}{N-2}f(1,N-2) + \frac{1}{N-1}f(1,N-1) \\ & + \frac{1}{1}f(2,1) + \cdots + \frac{1}{N-2}f(2,N-2) \\ & \vdots \\ & + \frac{1}{1}f(N-2,1) + \frac{1}{2}f(N-2,2) \\ & + \frac{1}{1}f(N-1,1)\end{aligned}$$

- Let's look closer at the f function:

$$f(n, \Delta_n) = \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(n+\Delta_n)}{t(n+\Delta_n)} \right| \right\}$$

- We at least need the following function values:

$$\begin{bmatrix} f(1,1) & f(1,2) & \cdots & f(1,N-2) & f(1,N-1) \\ f(2,1) & f(2,2) & \cdots & f(2,N-2) & \\ \vdots & & & & \\ f(N-2,1) & f(N-2,2) & & & \\ f(N-1,1) & & & & \end{bmatrix}$$

- Let's form a vector:

$$\mathbf{w} = [w(1), w(2), \ldots, w(N)]^{\mathrm{T}} = \left[\left| \frac{t(1)}{r(1)} \right|, \left| \frac{t(2)}{r(2)} \right|, \ldots, \left| \frac{t(N)}{r(N)} \right| \right]^{\mathrm{T}}$$

- Thereby:

$$f(n,\Delta_n) = \ln\left\{\left|\frac{t(n)}{r(n)}\right|\left|\frac{r(n+\Delta_n)}{t(n+\Delta_n)}\right|\right\}$$
$$= \ln\left\{\frac{w(n)}{w(n+\Delta_n)}\right\}$$

- where:

$$n = 1,\ldots,N-1 \qquad n+\Delta_n = n+1,\ldots,N$$

- Let's turn to an implementation of the estimation.

- Version 1 is a direct implementation of the double sum:

$$x_3 = \frac{-2}{N(N-1)} \sum_{n=1}^{N-1} \sum_{\Delta_n=1}^{N-n} \frac{1}{\Delta_n} \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(n+\Delta_n)}{t(n+\Delta_n)} \right| \right\}$$

```
function [ x3 ] = x3est1( r, t )
  %x3est1 Estimation of x3 using method 1 with
  %       full data set

  N = length(r);
  x30 = zeros(N*(N-1)/2,1);
  idx = 0;

  for n=1:N-1
    for Dn=1:N-n
      idx = idx + 1;
      x30(idx) = log(abs(t(n)/r(n)) ...
                 *abs(r(n+Dn)/t(n+Dn)))/Dn;
    end
  end
  x3 = -2/(N*(N-1))*sum(x30(:));
end
```

Inefficient implementation using a double sum. Using an Intel Core 2 Duo 2.13 GHz with 4 GB memory requires 1.8 s for N=5000.

- Version 2 where the inner sum over Δ_n has been vectorized:

$$x_3 = \frac{-2}{N(N-1)} \sum_{n=1}^{N-1} \left\{ \frac{1}{1} \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(n+1)}{t(n+1)} \right| \right\} + \cdots + \frac{1}{N-n} \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \left| \frac{r(N)}{t(N)} \right| \right\} \right\}$$

```
function [ x3 ] = x3est2( r, t )
  %x3est2 Estimation of x3 using method 2 with
  %       full data set

  % Vector length
  N = length(r);

  % Compute estimate of x3
  x30 = zeros(N-1,1);
  for n=1:N-1
    x30(n) = sum(log(abs(t(n)/r(n)) ...
             *abs(r(n+1:end)./t(n+1:end))))./(1:N-n)');
  end
  x3 = -2/((N-1)*N)*sum(x30(:));
end
```

Quite memory and time efficient implementation using a double sum. Using an Intel Core 2 Duo 2.13 GHz with 4 GB memory requires 1.5 s for N=5000. Not a big difference from the 1.8 s in case of a double loop.

Important to use “end” and not “N” here. It is faster to use end in colon expressions.

- As for upper limits never use N but use end. This is quite a bit faster:

$$x_3 = \kappa \sum_{n=1}^{N-1} \sum_{\Delta_n=1}^{N-n} \frac{1}{\Delta_n} \left\{ \ln \left\{ \left| \frac{t(n)}{r(n)} \right| \right\} + \ln \left\{ \left| \frac{r(n+\Delta_n)}{t(n+\Delta_n)} \right| \right\} \right\}$$

$$= \kappa \sum_{n=1}^{N-1} \sum_{\Delta_n=1}^{N-n} \frac{1}{\Delta_n} \left[\ell(n+\Delta_n) - \ell(n) \right]$$

- where:

$$\ell(n) = \ln \left\{ \left| \frac{r(n)}{t(n)} \right| \right\} \qquad \kappa = \frac{-2}{N\,(N-1)}$$

- This can in principle be based on $(N\text{-}1)\times(N\text{-}1)$ matrices. This is, however, quite memory hungry. The outer loop over n can then be used and the inner loop handled by vector operations.
- It is an advantage to pre-calculate some of the variables - for example Δ_n.

Case Study: Loops – Always to be Avoided?

- The implementation in version 3 according to the principle just derived is:

```
function [ x3 ] = x3est3( r, t )
  %x3est3 Estimation of x3 using method 3 with
  %       full data set

  % Vector length
  N = length(r);

  % Compute estimate of x3
  x30 = zeros(N-1,1);
  logd   = log(abs(r./t));
  F = 1./(1:N-1)';
  for n=1:N-1
    x30(n) = sum((logd(n+1:end)-logd(n)).*F(1:N-n));
  end
  x3 = -2/((N-1)*N)*sum(x30(:));
end
```

Quite memory and time efficient implementation using a single loop. Using an Intel Core 2 Duo 2.13 GHz with 4 GB memory requires 0.33 s for N=5000. Note that variables are computed outside the loop as much as possible.

Case Study: Loops – Always to be Avoided?

- Version 4 completely without loops – but with high memory usage:

```
function [ x3 ] = x3est4( r, t )
  %x3est4 Estimation of x3 using method 4 with
  %       full data set

  % Vector length
  N = length(r);

  % Form matrix of Dn values - only upper
  % diagonal used later on
  Dn = toeplitz(1:-1:-(N-3),1:N-1) ...
       + diag(ones(N-2,1),-1);

  % Compute D and estimate x3
  D = log(abs(t(1:N-1)./r(1:N-1)) ...
      * transpose(abs(r(2:N)./t(2:N)))) …
      ./ Dn .* triu(ones(N-1,N-1));
  x3 = -2/(N*(N-1)) * sum(D(:));
end
```

The principle applied here is to completely avoid loops. To facilitate this the second sum is expanded to N as the upper limit – end then multiply the terms not needed by zero (via an upper triangular matrix – TRIU). This obviously costs more computations – but as computations (multiplications and additions) comes almost for free it is often worth the trouble.

The down-side of avoiding the loops here is that our matrices become very large – size $(N\text{-}1)\times(N\text{-}1)$ which for N≈1E6 is asking for trouble. Using an Intel Core 2 Duo 2.13 GHz with 4 GB memory requires 1.5 s. This is substantially slower than version 3 based on a single loop with pre-computed variables.

Case Study: Loops – Always to be Avoided?

- And finally version 5 avoiding the SUM call inside the loop:

```
function [ x3 ] = x3est5( r, t )
  %x3est5 Estimation of x3 using method 5
  %       with full data set

  % Vector length
  N = length(r);

  % Compute estimate of x3
  x30 = zeros(N-1,1);
  logd   = log(abs(r./t));
  F = 1./(1:N-1);
  for n=1:N-1
    x30(n) =  F(1:N-n) * (logd(n+1:end)-logd(n));
  end
  x3 = -2/((N-1)*N)*sum(x30(:));
end
```

This implementation is similar to version 3 except the sum inside the loop is changed with a vector-vector product.

From a parallelization point of view it is important to observe that the loop can be computed in any order – i.e. n=100 can be computed before n=1. This is a huge advantage if we later want to take advantage of parallel execution.

- Overview of timing for the 4 methods (in seconds):

	N = 1000	N = 3000	N = 5000
1: Double sum	0.073	0.649	1.796
2: Single sum, no reuse	0.079	0.570	1.506
3: Single sum, reuse	0.029	0.142	0.330
4: No loops	0.129	1.232	5.882
5: Sum via vector-vector multiplication	0.014	0.077	0.190

- To learn:
  - Not always an advantage to avoid loops – it depends entirely on how it is done.
  - Do as many computations outside loops as possible – make pre-computations.
  - Use “end” when making colon indexing (more on this later).

Practical Information

Colfax GPU HPC Cluster

- **ged0.lab.es.aau.dk:**
  - **Colfax CXT3000N.**
  - CPU: Amd AM3 Phenom II QC Model 965; 3.4 GHz, 140 Watt, 8 MB cache.
  - Memory: 4 x 4 DDR2 RAM.
  - Disks:
    - 1 x Western Digital RE3 WD5002ABYS 500 GB; 7200 RPM, 16 MB cache, SATA 3.0 Gb/s.
    - 2 x Western Digital RE3 WD1002FBYS 1000 GB; 7200 RPM, 32 MB cache, SATA 3.0 Gb/s.
  - GPUs: 4 x NVIDIA Tesla C1060.
  - Operating system: CentOS Linux 5.4.
  - Software: MATLAB R2010b; Jacket 1.7.
- **ged1.lab.es.aau.dk:**
  - **Colfax CXT5000.**
  - CPU: 2 x Intel Xeon X5560.
  - Memory: 8 x 4 DDR3 RAM; 1333 MHz.
  - Disks:
    - 5 x Seagate Cheetah 15K.6 450 GB; 15,000 RPM SAS .
  - GPUs: 1 x NVIDIA Quadro FX5800 & 3 x NVIDIA Tesla C1060 (FX5800 ~ C1060 but is needed to drive a display).
  - Operating system: CentOS Linux 5.4.
  - Software: MATLAB R2010b; Jacket 1.7.

scout supercomputer for "signal processing and computing"
Colfax GPU HPC

- **Login-in node:**
  - Colfax CXT4000
  - 2 x Intel Xeon X5670 CPUs
  - 192 GB memory
  - 2 x 300 GB system disks
  - DDR-4X InfiniBand
  - 3 x NVIDIA Tesla C2070 GPUs
- **Administration node:**
  - Intel SR2612SR
  - 2 x Intel Xeon X5570
  - 72 GB memory
  - 2 x 300 GB system disks
  - 12 x 1 TB disks
  - DDR-4X InfiniBand
- **2 x Storage nodes**
  - LSI SAS 6 GB/s
  - 24 TB disks
- **5 x Compute Nodes:**
  - Colfax CXT4000
  - 2 x Intel Xeon X5570 CPUs
  - 48 GB memory
  - 300 GB system disks
  - DDR-4X InfiniBand
  - 3 x NVIDIA Tesla C2070 GPUs
- **5 x Compute Nodes:**
  - Colfax CXT4000
  - 2 x Intel Xeon X5570 CPUs
  - 48 GB memory
  - 300 GB system disks
  - DDR-4X InfiniBand
  - 3 x NVIDIA GeForce GTX580 GPUs
- **Backbone:**
  - InfiniBand DDR-4X (20 Gb/s)
  - Ethernet (1 Gb/s)

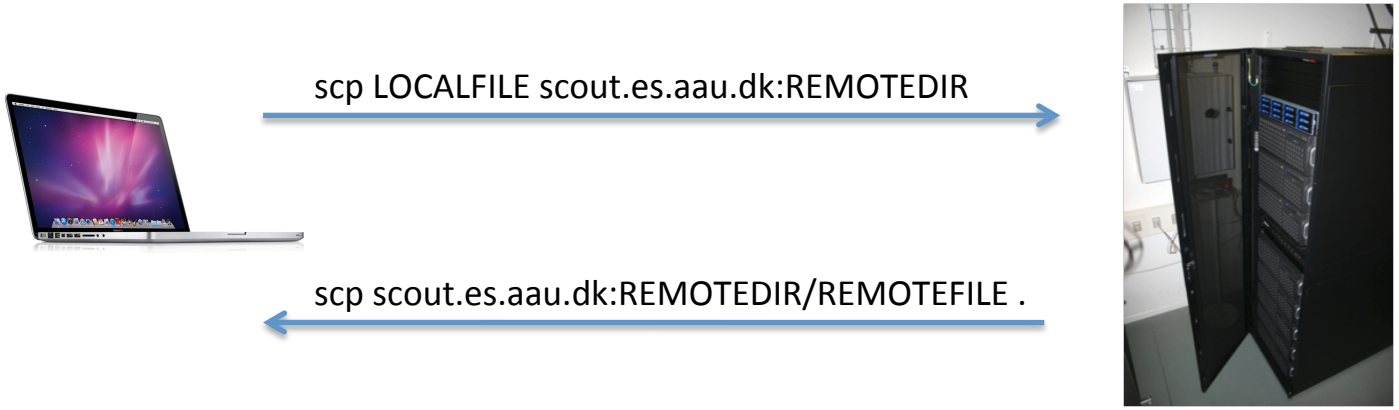
- Performance:
  - # CPU cores: 100
  - CPU Single Precision: 2.4 TFLOPS
  - CPU Double Precision: 1.2 TFLOPS
  - CPU memory: 744 GB
  - # GPU cores: 15744
  - GPU Single Precision: 41.6 TFLOPS
  - GPU Double Precision: 15.2 TFLOPS
  - GPU memory: 153 GB
  - Storage: 60 TB
  - Peak power consumption: ~15 kW
  - Main purpose: research in signal processing and computing; radio frequency modeling and simulation; audio and video processing and computing.
  - Login: scout.es.aau.dk

One of the two rack cabinets forming the scout supercomputer.

- Login via ssh (omit "-X" to avoid the network demanding windowing of X):
  - ssh –X [user]@ged0.lab.es.aau.dk
  - ssh –X [user]@ged1.lab.es.aau.dk
  - ssh –X [user]@scout.es.aau.dk
- Login from Mac OSX and Linux should be no problem at all. On Windows computers you may need to install for example "Putty" to be able to use ssh.

- **Note: We don't guarantee ANY kind of backup of any of your files on these computers. You have the responsibility to keep copies of important files.**

- **Copying files:**



Abbreviations

- Abbreviations:
  - Cache: Memory close to the CPU (usually on the CPU die) – often comes in several levels where level 1 is a small amount of very fast memory, level 3 is usually much larger (MB) but somewhat slower to access and level 2 is something between level 1 and 3.
  - CUDA: Compute
  - ALU: Arithmetic Logical Unit
  - MATLAB: MATrix LABoratory
  - GPU: Graphics Processing Unit
  - GPGPU: General Purpose Graphics Processing Unit
  - HPC: High Performance Computing (used to describe problems which either take too long time to compute on large desktop computers or problems which simply can't fit on such computers)
  - IB: InfiniBand (high speed / low latency bus to connect computers in a cluster – most often to share memory data)
  - LAN: Local Area Network
  - FLOPS: FLOating Point operations per Second (usually measured by matrix multiplication / LINPACK or something else which is arithmetically dense)

Further Reading

1) AccelerEyes: Torben's Corner. http://wiki.accelereyes.com/wiki/index.php/Torben's_Corner.
2) Torben Larsen, Gallagher Pryor, and James Malcolm: "Jacket: GPU Powered MATLAB Acceleration", In "GPU Computing Gems", Morgan-Kauffman, July 2011.
3) NVIDIA: NVIDIA CUDA C Programming Guide, 2010.
4) Jason Sanders and Edward Kandrot: "CUDA By Example – An Introduction to General-Purpose GPU Programming", Addison-Wesley, (Boston, MA, USA), 2011.
5) I.M. Haque and V.S. Pande: "Hard Data on Soft Errors: A Large-scale Assessment of Real-World Error Rates in GPGPUs". ACM International conference on Cluster, Cloud and Grid Computing, 2010.
6) Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha: "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware".SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, 2005, pp. 3-14.
7) Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder: "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, pp. 917-924, San Diego, California, USA. Published by ACM, New York, NY, USA.
8) Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan: "Brook for GPUs: stream computing on graphics hardware". SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 777-786, 2004, Los Angeles, California, USA. Published by ACM, New York, NY, USA.
9) Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron: "A performance study of general-purpose applications on graphics processors using CUDA''. {\it Journal of Parallel and Distributed Computing}, vol. 68, no. 10, pp. 1370--1380, 2008. DOI: 10.1016/j.jpdc. 2008.05.014.

Further Reading

10) Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba: “Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment”. Lecture Notes in Computer Science: High Performance Computing for Computational Science - VECPAR 2006, pp. 305-318. Springer Berlin/ Heidelberg, vol. 4395/2007.

11) K. Fatahalian, J. Sugerman, and P. Hanrahan: “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication”. HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Grenoble, France, pp. 133-137, 2004. Published by ACM in New York, NY, USA.

12) Timothy D. R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon: “Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores”. Proceedings of the 22nd annual International Conference on Supercomputing (ICS’08), June 7-12, 2008, Island of Kos Aegean Sea, Greece.

13) James Fung: “Computer Vision on the GPU”. Chapter 40 (pp. 649--666) in: “GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation” edited by Matt Pharr. Addison-Wesley and NVIDIA, 2005. ISBN: 0-321-33559-7.

14) S.S. Stone, J.P. Haldar, S.C. Tsao, W.-M. W. Hwu, B.P. Sutton, and Z.-P. Liang: “Accelerating advanced MRI reconstructions on GPUs”. Journal of Parallel and Distributed Computing, vol. 68, no. 10, pp. 1307-1318, 2008.

15) Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, and John Spiletic: “Benchmarking and implementation of probability-based simulations on programmable graphics cards”. Computers & Graphics, vol. 29, no. 1, pp. 71-80, 2005.

Further Reading

16) Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha: “GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management”. SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 325-336, 2006, ISBN: 1-59593-434-0, Chicago, IL, USA. Published by ACM in New York, NY, USA.

17) Robert G. Belleman, Jeroen Bédorf, and Simon F. Portegies Zwart: “High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA”. New Astronomy, vol. 13, no. 2, pp. 103-112. 2008. ISSN: 1384-1076.

18) Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig: “Molecular dynamics simulations on commodity GPUs with CUDA”. HiPC'07: Proceedings of the 14th international conference on High performance computing, pp. 185-196, 2007, ISBN: 3-540-77219-7, 978-3-540-77219-4, Goa, India. Published by Springer-Verlag in Berlin/Heidelberg, Germany.