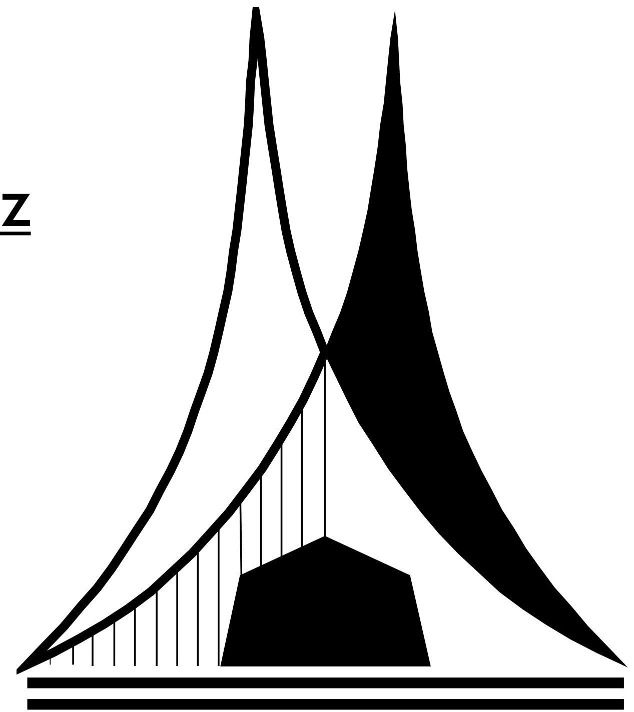
PANM 16
Programy a algoritmy numerické matematiky 16
3. - 8. června 2012
Dolní Maxov

Massive parallel implementation of ODE solvers

Cyril Fischer
fischerc@itam.cas.cz



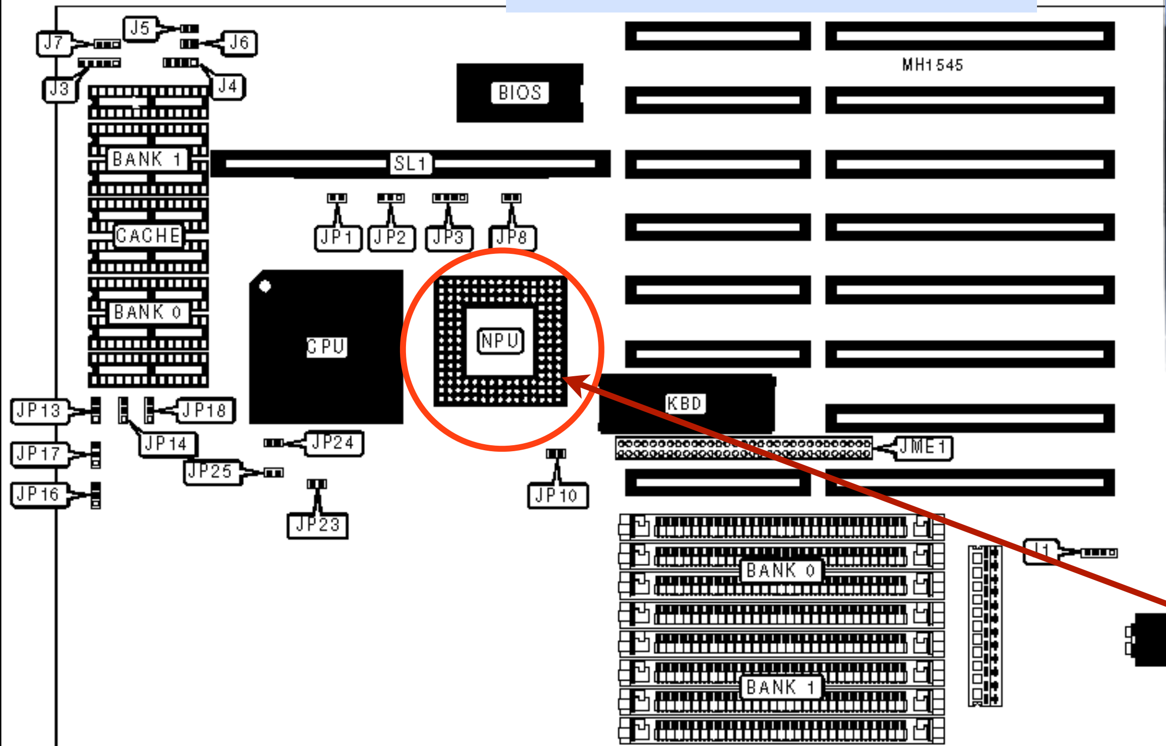
Institute of Theoretical and Applied Mechanics AS CR, v.v.i.
Prosecká 76, 190 00 Prague 9, CZ

WEITEK 4167 (1989)

weitek 4167 @ 25MHz	
Linpack single	3,8 mflops
Linpack double	2,4 mflops
intel 486 @ 33MHz	
Linpack	0,66 mflops

Weitek 4167 is a high-performance Floating Point Unit for Intel 80486 family of microprocessors.
- **memory-mapped interface**
- **asynchronous execution**
- **fast in single, slower in double precision**

486 motherboard



486UV AT (BUFFALO PRODUCTS, INC., 1990)

Processor	80486SX/80487SX/80486DX
Processor Speed	20/25/33/50(internal)/50MHz
Chip Set	UMC
Max. Onboard DRAM	**32MB**
Cache	**64/128/256KB**
BIOS	**AMI/Phoenix**
NPU Options	**Weitek 4167**

OUTLINE + Introduction

- Theory & Howto
- GPU architecture
- Example 1 - blind usage
- Example 2 - intensive usage
- Quadrature
- Motivation
- ODE - Parallelization
- ODE - Parametric studies
- ODE - Examples
- Conclusions

NVIDIA Tesla M2050

1.03 Tflops (single precision)

TOP 10 Systems - 11/2011

1. K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect
2. NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050
3. Cray XT5-HE Opteron 6-core 2.6 GHz
4. Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU
5. HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows
6. Cray XE6, Opteron 6136 8C 2.40GHz, Custom

TOP 500 #2: Tianhe-1A
14336 X5670(6 core Xeon) CPUs
7,168 Nvidia Tesla M2050 GPUs.
186368 cores, 229376GB mem
2566 Tflops, to build $88 million

TOP 500 #3: Cray XT5-HE:
±40000 Opteron (6 core) 2.6 GHz
224162 cores, 1759Tflops

Graphics Card Processors (GPU)

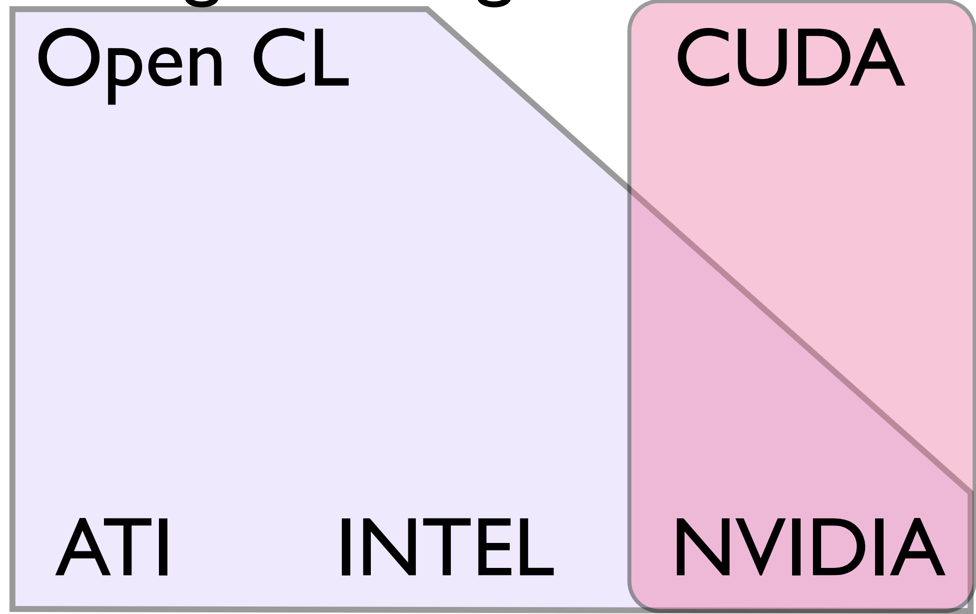
massive parallel processors
equipped with fast memory
several dozens to several hundreds computational cores per GPU
dedicated GPUs for “number crunching” (no video output)

Works on most modern GPUs

Programming standards:

Open
Computing
Language

open industry standard

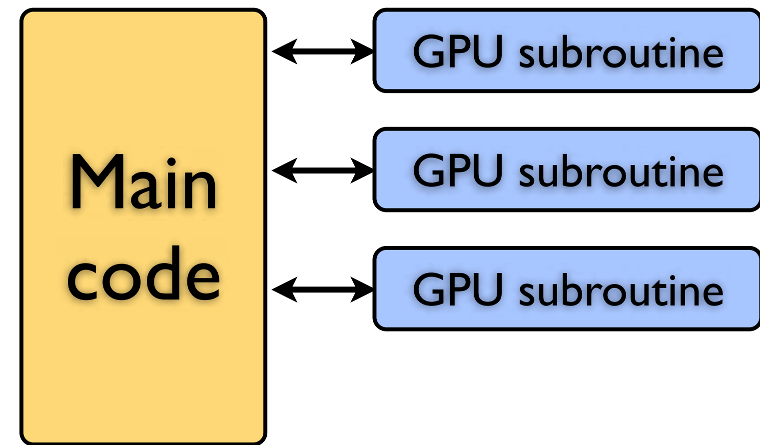


Compute
Unified
Device
Architecture

proprietary, freely available

How to use?

GPU is not COMPUTER in COMPUTER
GPU performs specific tasks at higher speed than CPU
GPU act as co-processor.



Typical usage:

- **blind usage - via specialised library (CUBLAS, CUFFT,...)**
  Most of the peculiarities are hidden in the interface to the library. Limited additional knowledge is necessary.
  E.g. basic usage in Mathematica, Matlab etc.

- **extensive usage - Monte Carlo methods, data processing etc.**
  Relatively simple algorithm has to be applied in the same way for wide variety of input data.
  Unmodified algorithm can be compiled for GPU and run in parallel on all available cores.
  Limitations: limited size of the code, limited memory/registers, no I/O operations, no 3rd party libraries, ...

- **intensive usage - sophisticated algorithms**
  specially tailored kernels, block matrix operations, sorting algorithms, image processing, data filtering, numerical integration, finite elements, ...

Hardware overview (older cards)

GeForce 200 Series Hardware

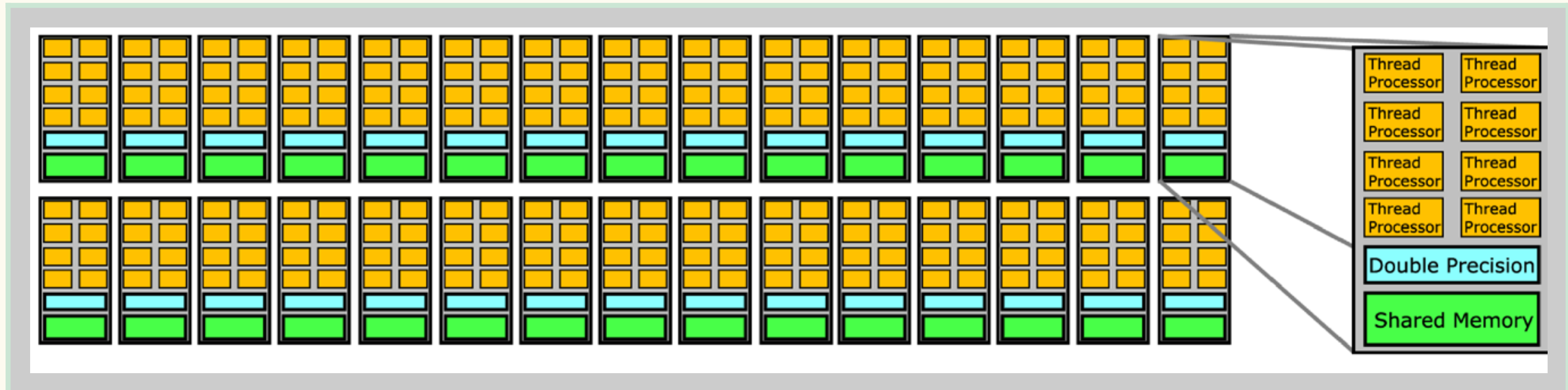
This hardware has 240 thread processors that can execute kernel threads.

These processors are grouped into 30 multiprocessors.

Each multiprocessor contains 8 single-precision thread processors, one double precision unit, and shared memory.

Each multiprocessor performs in Single Instruction Multiple Thread (SIMT).

Each thread in SIMT runs its own kernel separately from the others. They have their own instruction address and register state.



For devices of compute capability 1.x, a multiprocessor consists of:

8 CUDA cores for integer and single-precision floating-point arithmetic operations,

1 double-precision floating-point unit for double-precision floating-point arithmetic operations,

2 special function units for single-precision floating-point transcendental functions

(these units can also handle single-precision floating-point multiplications),

1 warp scheduler.

Hardware overview (newer cards)

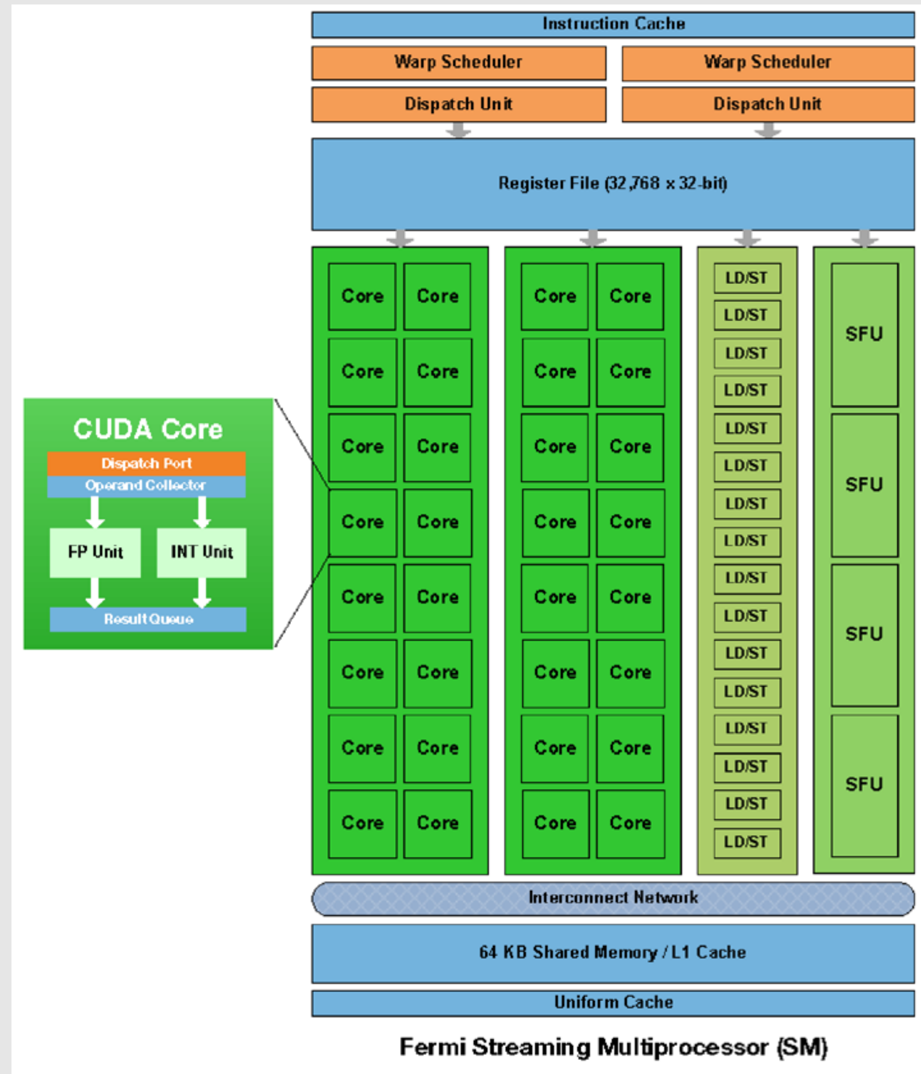
FERMI: up to 16 Multiprocessors, 32 cores each.

Each CUDA processor (core) has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).

Each Multiprocessor has four Special Function Units (SFUs), that execute transcendental instructions such as sin, cosine, reciprocal, and square root.

Each SFU executes one instruction per thread, per clock; a warp executes over eight clocks.

The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied



Fermi Streaming Multiprocessor (SM)

What do we pay?

Hardware

Graphics Card Product	cores	Memory	Clock	flops (single)	Price
TESLA C2070	448	6.0GB ECC, 384-bit, 144GB/s	1.15Ghz	1030 Gflops	3 400 €
Quadro 6000 (Fermi)	448	6.0GB ECC, 384-bit, 144GB/s	0.95GHz	1030 Gflops	3 200 €
Quadro 5000 (Fermi)	352	2.5GB ECC, 320-bit, 120GB/s			1 800 €
Quadro 4000 (Fermi)	256	2.0GB, 256-bit, 89.6GB/s	0.95GHz	486 Gflops	800 €
Quadro 2000 (Fermi)	192	1.0GB, 128-bit, 41.6GB/s			500 €
GeForce GT 430	96	1.0GB, 128-bit, 25.6GB/s	1.4GHz		70 €

Software

C/C++/FTN etc	NVIDIA CUDA development toolkit	free
Mathematica(tm)	CUDALink	included
MATLAB(tm)	Parallel computation toolbox	$$$

Skills

Parallel programming techniques, shared/distributed memory, etc.
C/C++ or other languages

Example I - blind usage

Power method for maximal eigenvalue of a matrix.

```
#include "mkl/mkl_cblas.h"
 float* h_A,* h_B, * h_C;
```

Initialisation

$$A = \begin{pmatrix} 1^2 & 0 & \cdots & \\ 0 & 2^2 & 0 & \cdots \\ & \vdots & \ddots & \vdots \\ & & 0 & n^2 \end{pmatrix}$$

```
h_A = (float*)malloc(n2 * sizeof(h_A[0]));
h_B = (float*)malloc(N * sizeof(h_B[0]));
h_C = (float*)malloc(N * sizeof(h_C[0]));

for (i = 0; i < n2; i++) h_A[i] = 0.0f;
for (i = 0; i < N; i++){ h_B[i] = 1.0f;h_C[i] = 1.0f;
                         h_A[i*N+i]=(i+1)*(i+1); }
```

prepare data on CPU

Allocate GPU memory, transfer data

for $n = 1, \ldots$ do

$$\mathbf{w} = \mathbf{A}\mathbf{v}_n$$

$$\lambda_{n+1} = \sqrt{\mathbf{w}^T \mathbf{w}}$$

$$\mathbf{v}_{n+1} = \frac{\mathbf{w}}{\lambda_{n+1}}$$

```
beta=0.0f; lambda=1.0f;
   for(i=0;i<100;i++){
        alpha=(1.0f)/lambda;
        cblas_sgemv(CblasColMajor,CblasNoTrans,  N, N,
                              alpha, h_A, N, h_B, 1, beta, h_C, 1);
        lambda=cblas_snrm2 (N , h_C, 1);
        tmp=h_B;h_B=h_C;h_C=tmp;
   }
```

Execute algorithm

Collect data

Correct result is

$$\lambda_{\max} = n^2$$

```
...
//Free memory
}
```

Example I - blind usage

CUDA
CUBLAS Library

Power method for
maximal eigenvalue of a matrix.

$$A= \begin{pmatrix} 1^2 & 0 & \cdots & \\ 0 & 2^2 & 0 & \cdots \\ & \vdots & \ddots & \vdots \\ & & 0 & n^2 \end{pmatrix}$$

for $n = 1, \ldots$ do

$$\begin{aligned} \mathbf{w} &= \mathbf{A}\mathbf{v}_n \\ \lambda_{n+1} &= \sqrt{\mathbf{w}^T \mathbf{w}} \\ \mathbf{v}_{n+1} &= \frac{\mathbf{w}}{\lambda_{n+1}} \end{aligned}$$

Correct result is

$$\lambda_{\max} = n^2$$

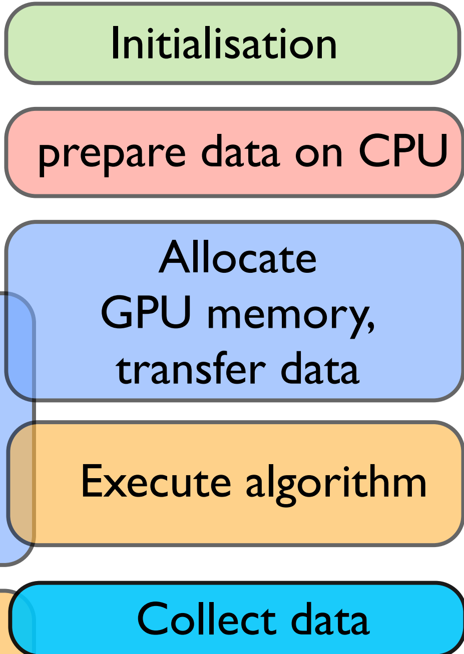
```
#include "cublas.h"

float *h_A,*h_B,*h_C;   // vectors in main memory
float *d_A,*d_B,*d_C;   // vectors in GPU memory
status = cublasInit();
```

```
h_A = (float*)malloc(n2 * sizeof(h_A[0]));
h_B = (float*)malloc(N * sizeof(h_B[0]));
h_C = (float*)malloc(N * sizeof(h_C[0]));

for (i = 0; i < n2; i++) h_A[i] = 0.0f;
for (i = 0; i < N; i++) h_B[i] = h_C[i] = 1.0f;
for (i = 0; i < N; i++) h_A[i*N+i]=(i+1)*(i+1);
```

```
status = cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
status = cublasAlloc(N, sizeof(d_B[0]), (void**)&d_B);
status = cublasAlloc(N, sizeof(d_C[0]), (void**)&d_C);

status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
status = cublasSetVector(N, sizeof(h_B[0]), h_B, 1, d_B, 1);
status = cublasSetVector(N, sizeof(h_C[0]), h_C, 1, d_C, 1);
```

```
beta=0.0;lambda=1.0;
for(i=0;i<100;i++){
   alpha=(1.0f)/lambda;
   cublasSgemv('n',  N, N, alpha, d_A, N, d_B, 1, beta, d_C, 1);
   lambda=cublasSnrm2 (N , d_C, 1);
  tmp=d_B;d_B=d_C;d_C=tmp;
}
```

```
status = cublasGetVector(N, sizeof(h_C[0]), d_C, 1, h_C, 1);
...
//free memory
```



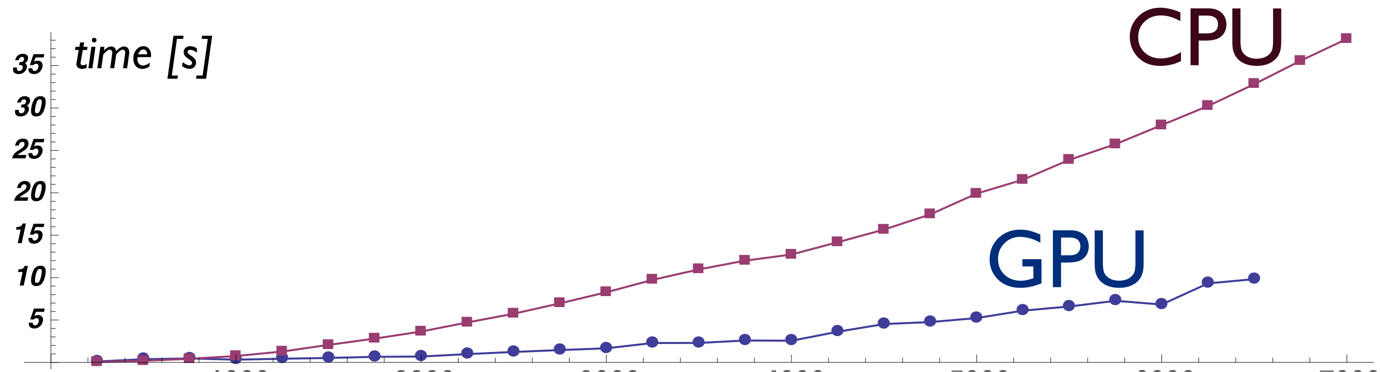
Example 1 - Timing

# of iterations: 400, single precision

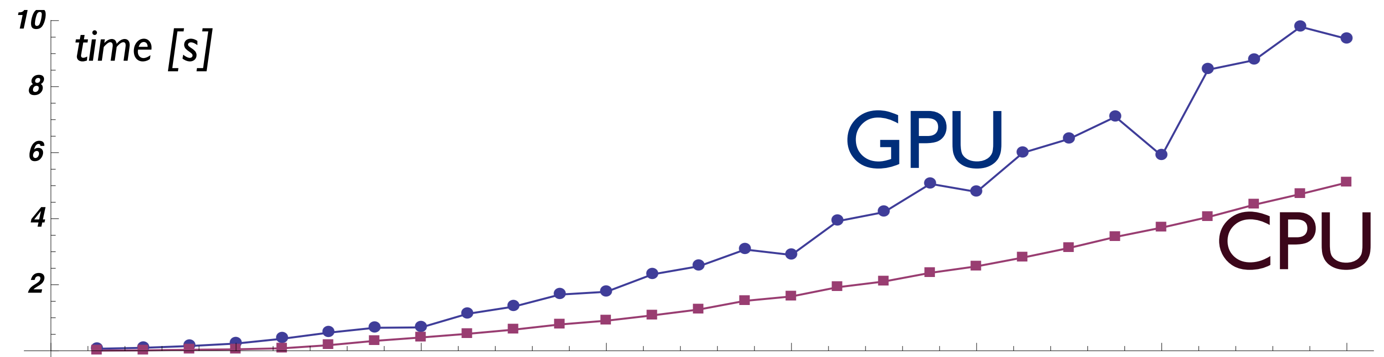
CPU BLAS implementation: intel's MKL

GPU BLAS implementation: nvidia's cudaBLAS

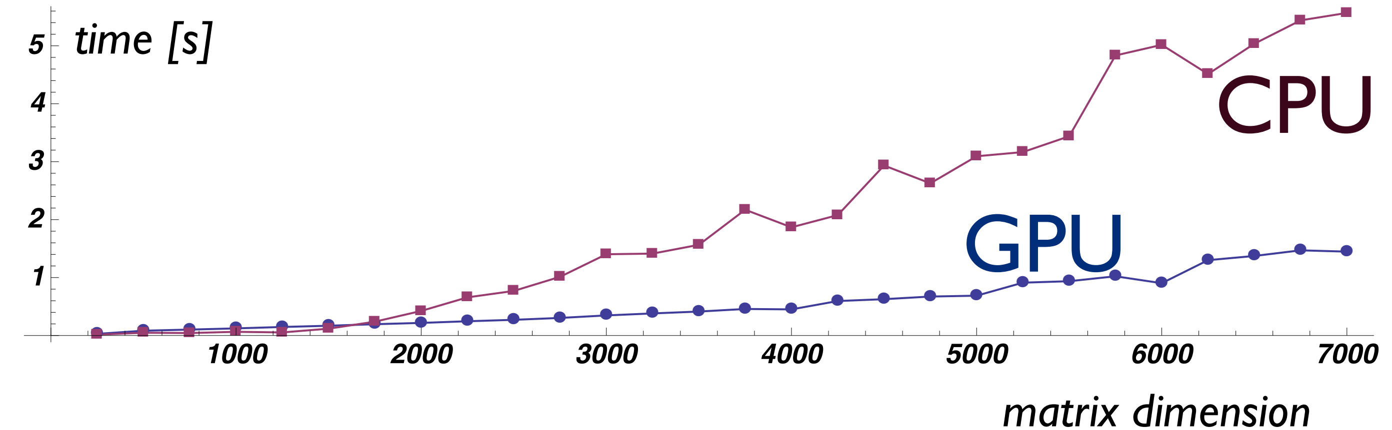
notebook:
2 core
Core2, 2.13GHz
(1600 €)
Nvidia 320M



desktop:
8 threads
i7 950, 3.07GHz
(1400 €)
Nvidia GT430
(~ 70 €)



server:
16 threads
2x Xeon 5560, 2.8GHz
(4000 €)
Nvidia Quadro 4000
(800 €)



Example 2 - intensive usage - Laplace equation

Laplace's equation takes the form of:

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

A finite difference equation can be constructed by superimposing a regular grid, with equal spacing the in x and y direction, over the region of interest. Laplace's equation can then be approximated at each grid point. The resulting equations are solved by iteration, with the previous approximation being the input to the next iteration.

$$\Phi(x+h,y) = \Phi(x,y) + h\frac{\partial \Phi}{\partial x} + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial x^2} + \frac{h^3}{6}\frac{\partial^3 \Phi}{\partial x^3} + O(h^4)$$

+

$$\Phi(x-h,y) = \Phi(x,y) - h\frac{\partial \Phi}{\partial x} + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial x^2} - \frac{h^3}{6}\frac{\partial^3 \Phi}{\partial x^3} + O(h^4)$$

=

$$\Phi(x+h,y) + \Phi(x-h,y) = 2\Phi(x,y) + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial x^2} + O(h^4)$$

$$\Phi(x,y+h) = \Phi(x,y) + h\frac{\partial \Phi}{\partial y} + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial y^2} + \frac{h^3}{6}\frac{\partial^3 \Phi}{\partial y^3} + O(h^4)$$

+

$$\Phi(x,y-h) = \Phi(x,y) - h\frac{\partial \Phi}{\partial y} + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial y^2} - \frac{h^3}{6}\frac{\partial^3 \Phi}{\partial y^3} + O(h^4)$$

=

$$\Phi(x,y+h) + \Phi(x,y-h) = 2\Phi(x,y) + \frac{h^2}{2}\frac{\partial^2 \Phi}{\partial y^2} + O(h^4)$$

+

Four point numerical scheme:

$$\Phi(x,y) = \frac{1}{4}\left(\Phi(x+h,y) + \Phi(x-h,y) + \Phi(x,y-h) + \Phi(x,y-h)\right) + O(h^4)$$

Numerical scheme in C:

```
for(int i=1;i<N-1;i++)
for(int j=1;j<N-1;j++)
B[i,j] = 0.25*( A[i+1,j] + A[i-1,j] + A[i,j+1] + A[i,j-1] );
```

Example 2 - intensive usage - Laplace equation

Kernel source

```
__global__ void Laplace_d(float *A, float *B, int N)
 {
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    int j = blockIdx.y * blockDim.y + threadIdx.y ;

    if(i>0 && i<N-1 && j>0 && j<N-1)
         B[i*N + j] = 0.25*(  A[i*N+j+1] + A[i*N+j-1] + A[(i+1)*N + j] + A[(i-1)*N + j] ;
   };
```

```
#define A(r,s) Am[(r)+(s)*N]
#define B(r,s) Bm[(r)+(s)*N]
```

Laplace kernel

```
void Laplace_h(double *Am, double *Bm, long N, long M)
{
#pragma omp parallel for
  for(int i=1;i<N-1;i++)
   for(int j=1;j<M-1;j++)
       B(i,j) = 0.25*(  A(i+1,j) + A(i-1,j) + A(i,j+1) + A(i,j-1) );
}
```

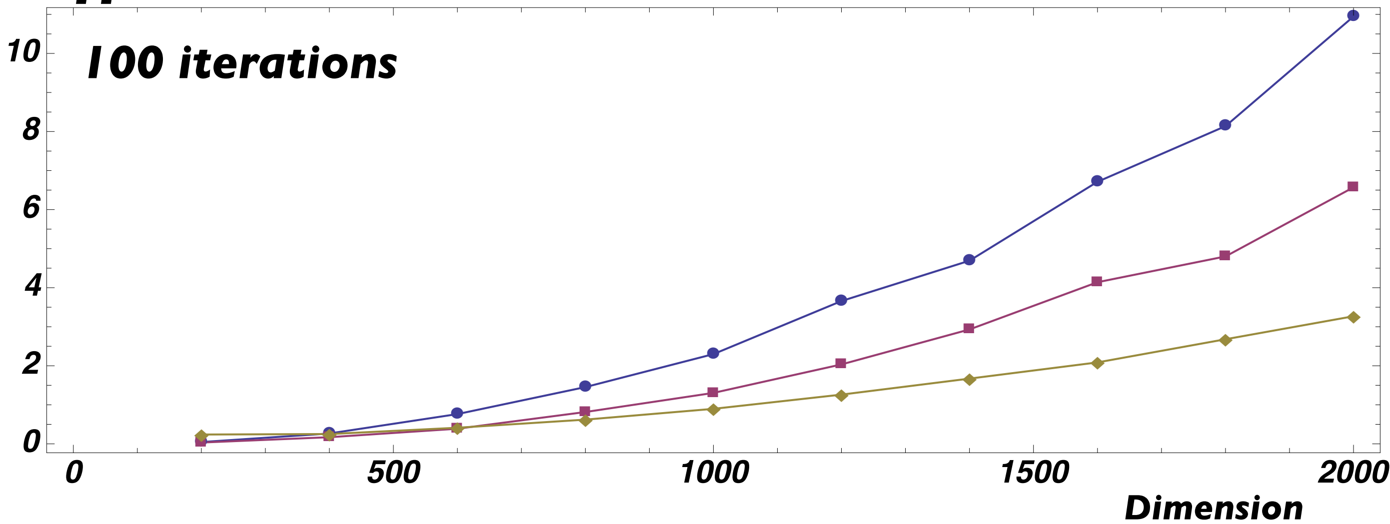
Directive for OpenMP (parallel run on CPU)

```
 ......

 for(int i=0; i<iter; i+=2){
    Laplace_h(A,B,dimensions[0],dimensions[1]);
    Laplace_h(B,A,dimensions[0],dimensions[1]);
  }

...
```

Example 2 - Timing

Intel Core2, 2x2.13GHz
Nvidia 320M

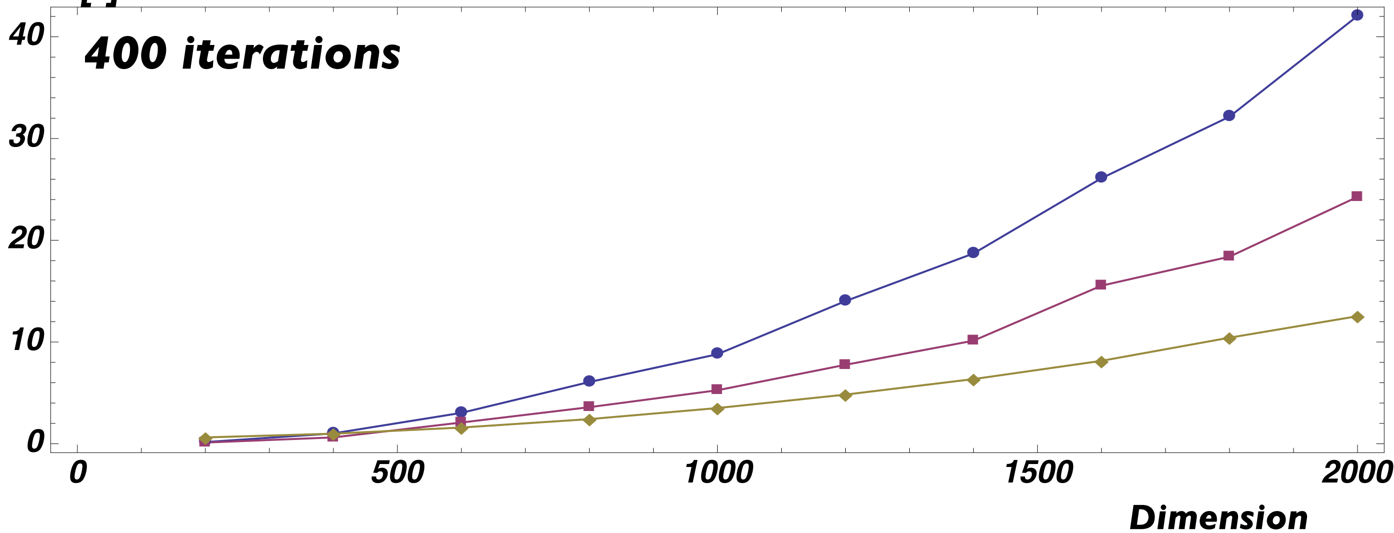
```
CUDAInformation[]
   "Clock Rate" -> 950000
   "Compute Capabilities" -> 1.2
   "Multiprocessor Count" -> 6
   "Core Count" -> 48
   "Total Memory" -> 265027584
```



scalar (1CPU)

parallel (2CPU)

CUDA kernel



scalar (1CPU)

parallel (2CPU)

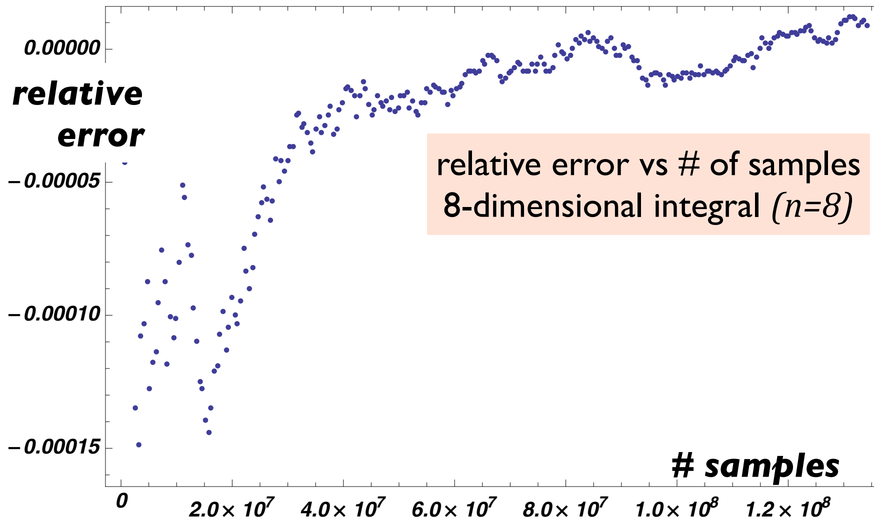
CUDA kernel

Quadrature

typically excellent for parallelization

Monte Carlo methods:
– powerful GPU random number generators are available
– cheap evaluation

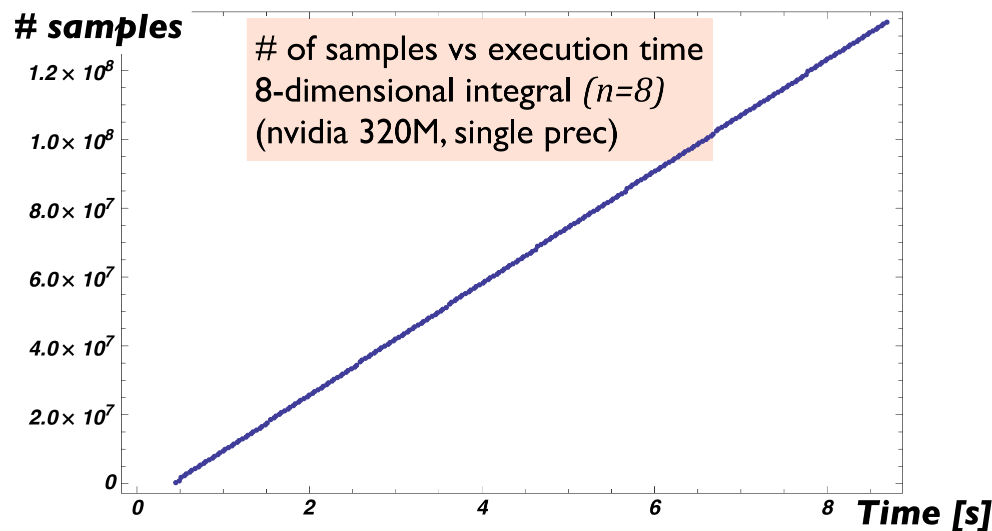
quadrature formulas:
– evaluate values in nodes
– evaluate error estimate for individual intervals
– (locally) refine the mesh
OR
– choose steplength *h=machine epsilon* (in single precision)

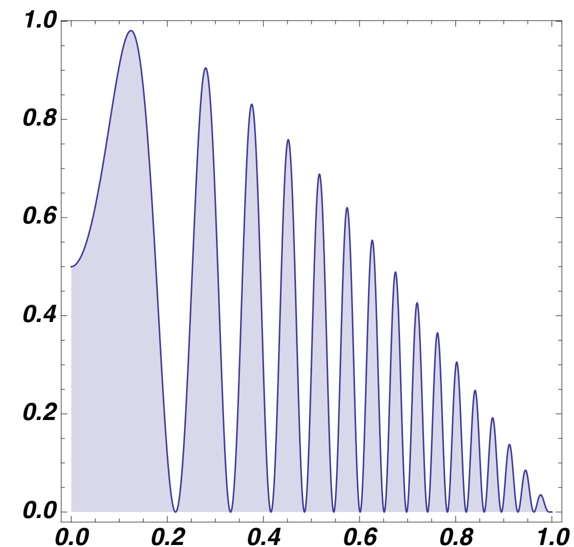


Example computation times for $h = 2^{-24} = 5.96 \times 10^{-8} = \varepsilon$

n=1: **0.2s** on notebook nvidia 320M, **2^{24} = 1.68×10^7** steps
n=2: **62s** on server nvid. Quadro 4000, **2^{48} =2.81×10^{14}** steps

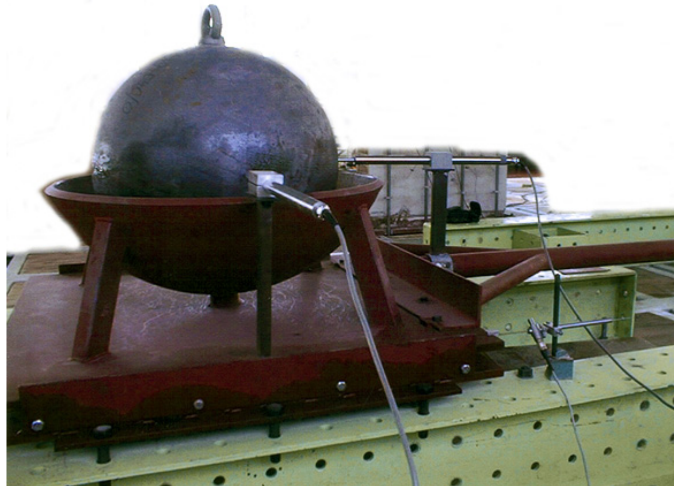
$$\int_0^1 \sum_{i=1}^{n} \frac{1}{2} \left(\cos\left(\frac{\pi x_i}{2}\right) + \sin\left(100 x_i^2\right) \cos\left(\frac{\pi x_i}{2}\right) \right) \mathrm{d}\mathbf{x}$$





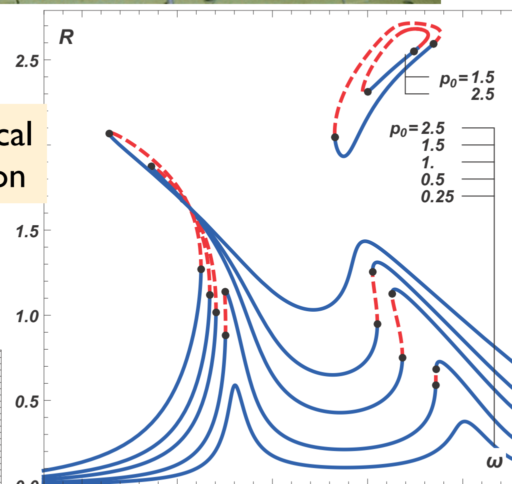
ODE - Motivation

Resonance behaviour of tuned mass dampers



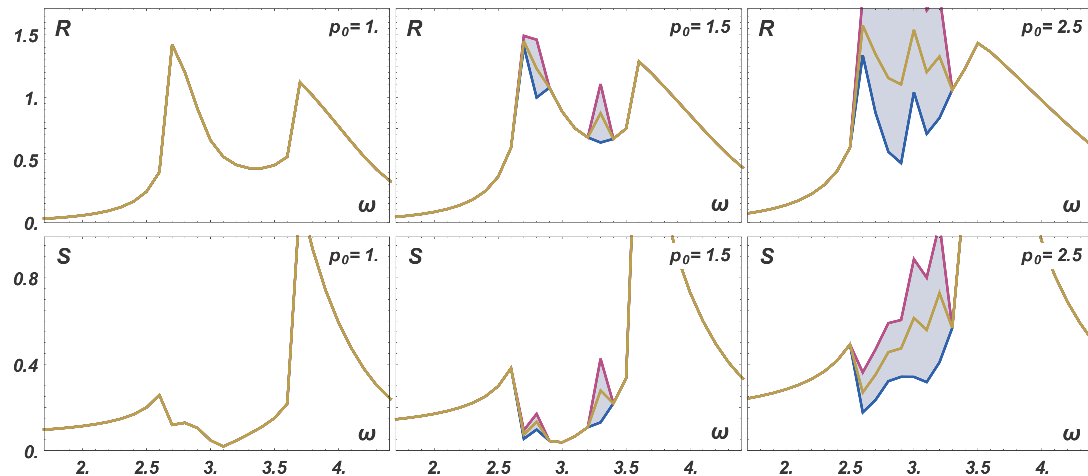
"Resonance curve" : dependence of the maximal output amplitude on input frequency

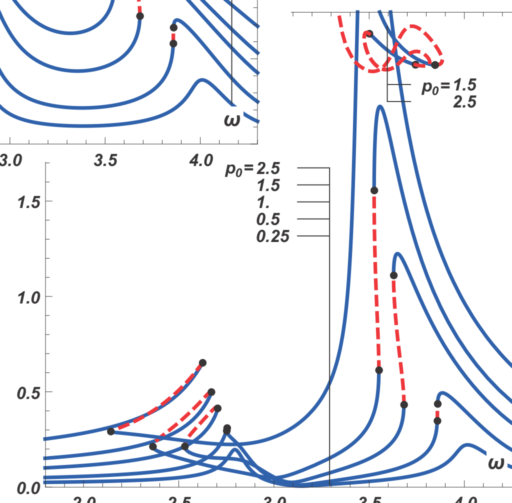
analytical solution



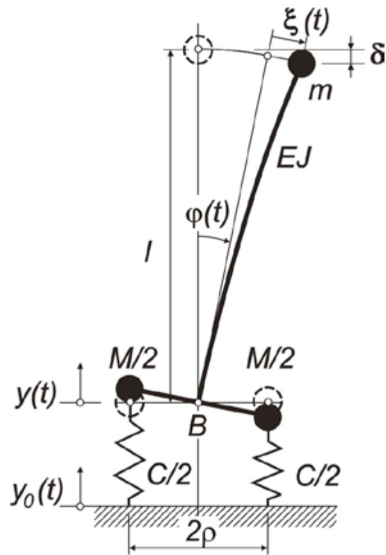
Frequency - amplitude plots:

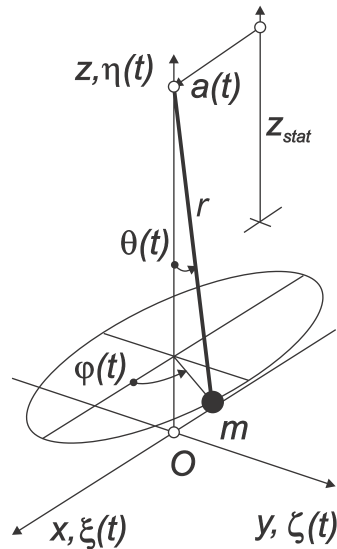
numerical solution

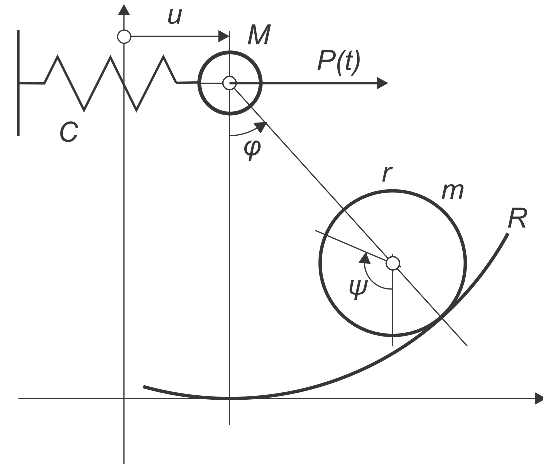




example structures:







example equation:

$$
\begin{aligned}
\ddot{\xi} + \frac{1}{2r^2}\xi\frac{\mathrm{d}^2}{\mathrm{d}t^2}(\xi^2+\zeta^2) + 2\beta_\xi\dot{\xi} + \omega_0^2\left(\xi + \frac{1}{2r^2}\xi(\xi^2+\zeta^2)\right) &= \omega^2\sin(\omega t) \\
\ddot{\zeta} + \frac{1}{2r^2}\zeta\frac{\mathrm{d}^2}{\mathrm{d}t^2}(\xi^2+\zeta^2) + 2\beta_\zeta\dot{\zeta} + \omega_0^2\left(\zeta + \frac{1}{2r^2}\zeta(\xi^2+\zeta^2)\right) &= 0
\end{aligned}
$$

ODE - Libraries

•CULSODA

– Livermore LSODA ODE solver for CUDA
– http://code.google.com/p/culsoda/

•ODEINT_v2

– header only C++ library
– various methods
– CUDA interface via abstract approach & THURST interface

•CUDA-sim

– Biology motivated equations
– Python-based, general purpose
– Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael P. H. Stumpf and Chris Barnes,
GPU accelerated biochemical network simulation. *Bioinformatics* Vol 27 (6) pp. 874-876
(http://bioinformatics.oxfordjournals.org/content/27/6/874.full)
– http://cuda-sim.sourceforge.net/

From ODEINT documentation

Important
The full power of CUDA is only available for really large systems where the number of coupled ordinary differential equations is of order *N=10^6* or larger. For smaller systems the CPU is usually much faster.

ODE - Parallelization, classical approach

A simple modification in the existing fourth order Runge-Kutta method which makes it amenable to parallelization on p processors.
C.P. Katti, D.K. Srivastava, On a parallel mesh-chopping algorithm for a class of initial value problems using fourth-order explicit Runge-Kutta method, *Applied Mathematics and Computation* 143 (2003) 565–570.

A parallel mesh chopping algorithm for a class of two point boundary value problems.
C.P. Katti, S. Goel, A parallel mesh chopping algorithm for a class of two-point boundary value problems, *Comput. Math. Appl.* 35 (1998) 121–128.

A theoretical framework for parallelization of Runge-Kutta methods:
A. Iserles, S.P. Norsett, On the theory of parallel Runge-Kutta methods, *IMA J. Numer. Anal.* 10 (1990) 463–488.

Family of explicit two step, two stage Runge-Kutta methods in which the two right hand side evaluations can be computed in parallel.
P.J. van der Houwen, B.P. Sommeijer, P.A. van Mourik, *Note on Explicit Parallel Multistep Runge-Kutta Methods,* Centrum voor Wiskunde en Informatica, Report NM-R8814, 1988.

A set of parallel block predictor corrector methods:
P.J. van der Houwen, Nguyen huu Cong, *Parallel Block Predictor-Corrector Methods of Runge-Kutta Type,* Centrum voor Wiskunde en Informatica Report NM-R9200, 1992.

Khalaf and Hutchinson have developed parallel algorithms for the initial value problem (1.1) by the use of (i) parallel shooting techniques, (ii) broadening of the computation front (BCF) techniques and (iii) block implicit methods to speed up the computation by using MIMD computing systems:
D. Hutchinson, B.M.S. Khalaf, Parallel algorithms for solving initial value problems: front broadening and embedded parallelism, *Parallel Comput.* 17 (1991) 957–968.
D. Hutchinson, B.M.S. Khalaf, Parallel algorithms for solving initial value problems: parallel shooting, *Parallel Comput.* 18 (1992) 661–673.

ODE - Parallelization, CUDA difficulties

– expensive CPU-GPU communication
– slow global / fast local memory
– thread divergence (no IFs, if possible)
– limited inter-thread communication

ODE - Parallelization, CUDA challenges

– adaptive step length - governed by maximum error estimate from all threads, i.e. global step for the whole ODE system

ODE - Parallelization, CUDA opportunities

– numerical approximation of Jacobian
– evolutionary PDE

ODE - Parametric study

Parametric study involves a huge number of simple runs.
Stiff system needs an implicit method.
Thus, for simplicity's sake, we use the implicit backward Euler method:

to avoid thread divergence
- fixed number of Newton steps
- explicit formula for linear system solution

Let's run the ode solver for harmonic rhs $u_0 \sin(\omega t)$ for N_ω values of $\omega \in (\Omega_{min}, \Omega_{max})$

```
...
omegas_h= (Real_t*)malloc(Nomega*sizeof(Real_t));
cudaMalloc((void**)&om_dev, Nomega*sizeof(Real_t));
results_h = (Real_t*)malloc(Nomega*sizeof(Real_t));
```

allocate memory

```
for (i=0;i<Nomega;i++) omegas_h[i]=MinOmega+i*(MaxOmega-MinOmega)/(Nomega-1);
cudaMemcpy(om_dev, omegas_h, Nomega*sizeof(Real_t), cudaMemcpyHostToDevice);
```

copy to GPU

```
int numThreadsPerBlock=128;
int numBlocks = (Nomega+numThreadsPerBlock-1) / numThreadsPerBlock;
```

specify size

```
EulCU<<<numBlocks, numThreadsPerBlock>>>(om_dev,Nomega,T);
```

execute program on GPU

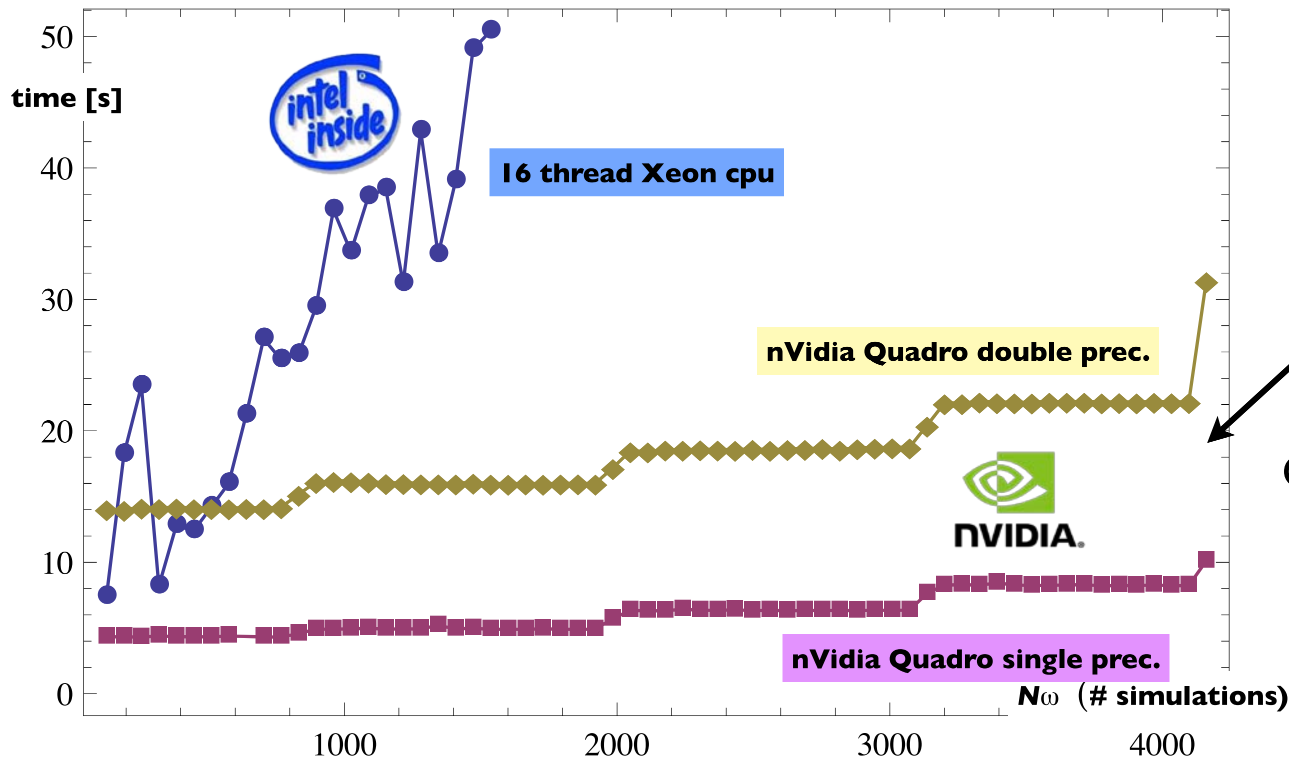
```
cudaMemcpy(results_h, om_dev, Nomega*sizeof(Real_t), cudaMemcpyDeviceToHost);
```

retrieve data from GPU

```
for (i=0;i<Nomega; i++) printf("%4d %f: %f \n",i,omegas_h[i],results_h[i]);

cudaFree(om_dev);
free(omegas_h);
free(results_h);
....
```

free memory

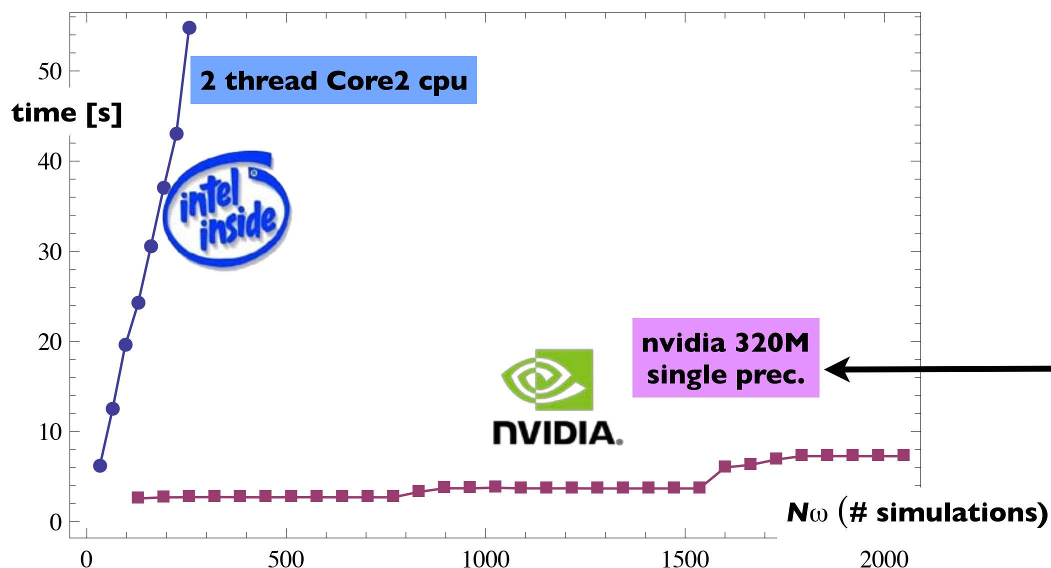
ODE - Parametric study - timing



nvidia Quadro 4000

```
Clock Rate → 950000 MHz
Compute Capabilities → 2
Multiprocesor Count → 8
Core Count → 256
Total Memory → 2G
```

ODE solution for $t \in (0, 100)$

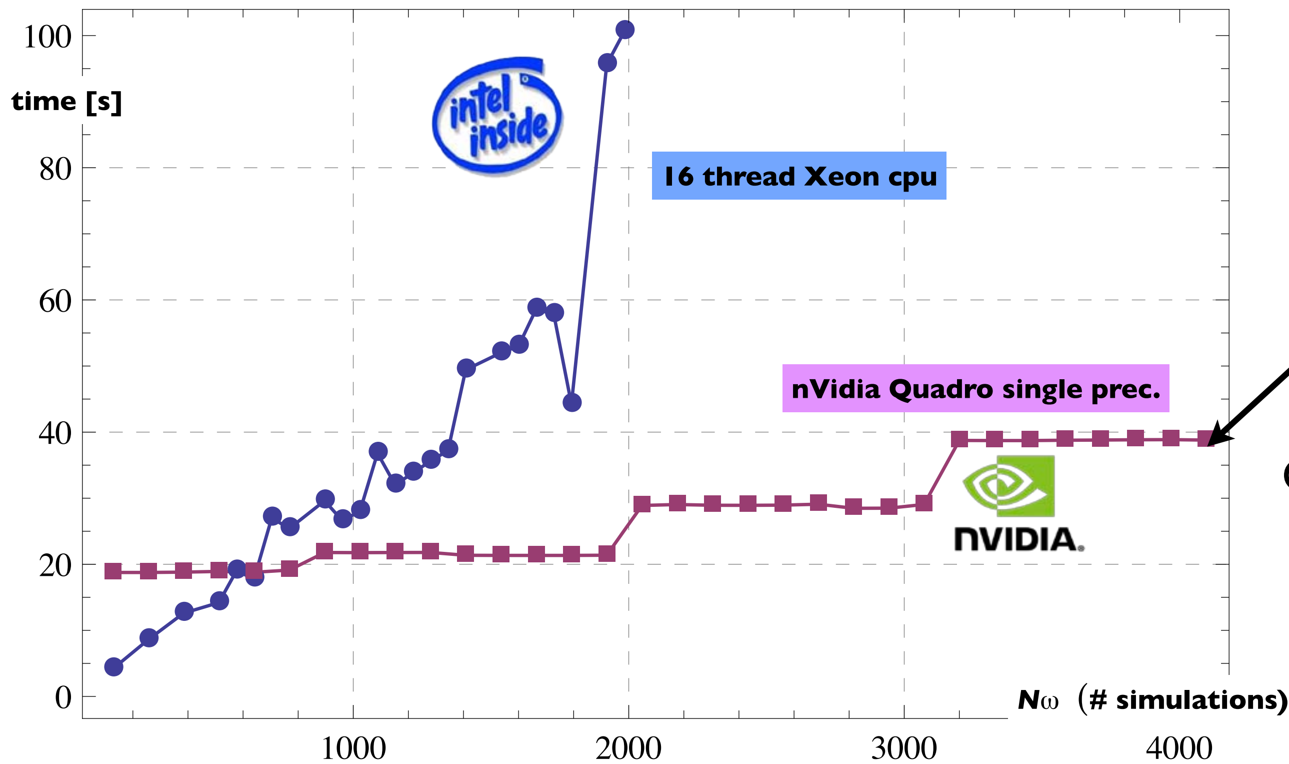


ODE solution for $t \in (0, 40)$

nvidia 320M

```
Clock Rate → 950000 MHz
Compute Capabilities → 1.2
Multiprocessor Count → 6
Core Count → 48
Total Memory → 256M
```

ODE - Parametric study - timing, continued



nvidia Quadro 4000

```
Clock Rate → 950000 MHz
Compute Capabilities → 2
Multiprocesor Count → 8
Core Count → 256
Total Memory → 2G
```

ODE solution for $t \in (0, 500)$

Conclusions

- It is always difficult to reasonably balance the time, which is necessary for certain task, between long computation and tedious programming.
- The great progress of the computer technology (driven mainly by the entertainment industry) enables the engineering community to solve fairly complex problems.
- Current GPUs are cheap devices with power of supercomputer. Using several GPUs one can build a teraflops supercomputer.
- This demands the engineers to adopt higher level of knowledge of programming techniques.
- High level programming packages like Matlab and Mathematica make this easier.
- NVIDIA CUDA programming toolkit offers API (Application programming interface), which hides the architectonical peculiarities of individual (NVIDIA) GPUs.
- OpenCL standard tries to cover wide variety of processor/GPU computation, not limited only to one vendor.
- Parallel numerical algorithms have been studied for several decades, many robust algorithms are available.
- It appears, that current trend in the computer technology is to bind CPU and GPU closely together to remove bottleneck in GPU/CPU communication (Intel “Sandy Bridge”, AMD “Fusion”). This trend offers new possibilities for high performance parallel computing.