

# Scientific computing on Graphics Processing Units

Basic principles and some experiments

Ph. Caussignac

May 2011

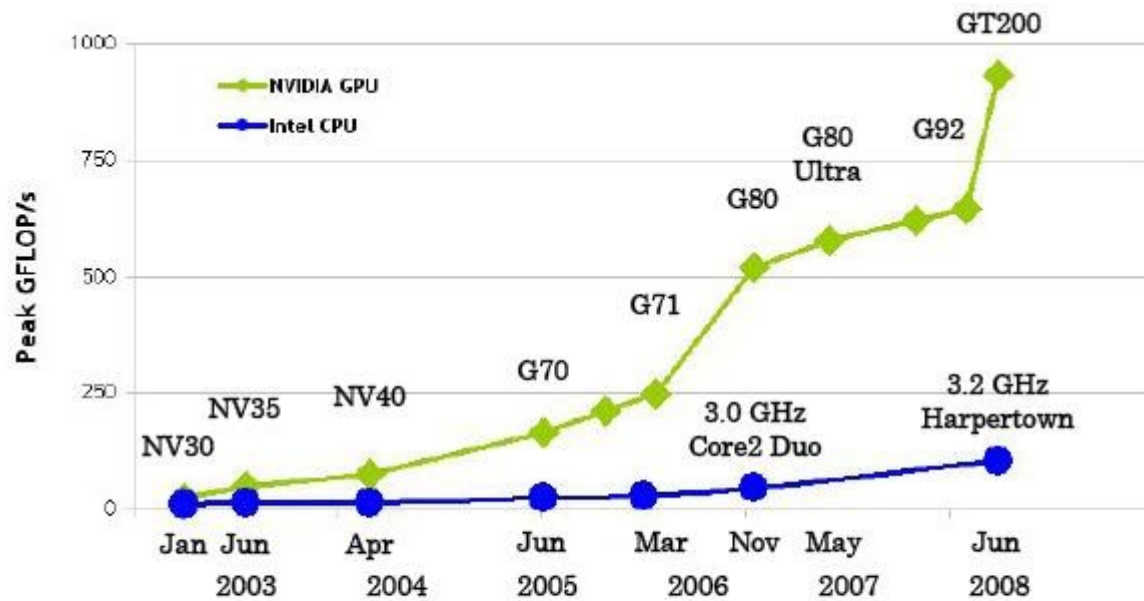
NB: Some figures are borrowed to the CUDA User's Guide

# CPU <-> GPU



Figure 1-2. The GPU Devotes More Transistors to Data Processing

# Performances



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

# Problems

Data transfer CPU  $\leftrightarrow$  GPU slow

User-friendly programming interface

Mathematical functions

Definition of arrays (integer, real)

Simple  $\leftrightarrow$  double precision

Standards?

# Linear Algebra Softwares

## ACML-GPU

GPU version of the AMD Core Math Library

Only sgemm, dgemm

Automatic balancing CPU  $\leftrightarrow$  GPU

Requires ATI cards (especially Radeon)

## NVIDIA Cublas, Cula

Requires Nvidia Cuda (Compute Unified Device Architecture) cards

Cublas: Blas functions

Cula: free version: only single precision

MAGMA Matrix Algebra on GPU and Multicore Architectures

Based on Cuda

Magmablas, Libmagma

# Nvidia Cuda

Used for coding scientific computing applications on GPU

No source

Two interfaces:

1. C for Cuda

Extension of C through the compiler nvcc

Initialize GPU, processor grid, allocate arrays, data transfer

2. Cuda driver API

Low level functions in C for programming GPU

# OpenCL

C++ API for programming GPU (NVIDIA, ATI cards)  
Originally developed by Apple

Can be used with a recent version of gcc

Allows easy coding of Multicore-CPU+GPU applications

Also heterogeneous computing:

One host connected to compute units (devices)

Same code can be compiled for different devices

Cuda to OpenCL translator (Swan)

# Free Nvidia Cuda Software

Nvidia driver

Cuda Toolkit

Utilities library

Cudablas

Cudafft, Cula (single precision)

nvcc compiler (C, C++: based on gnu)

Documentation (good)

Development Toolkit (SDK)

Application examples

Allows to develop new applications (makefiles)

# Basic principles

GPU initializing: 2D or 3D grid of shared memory processors blocks

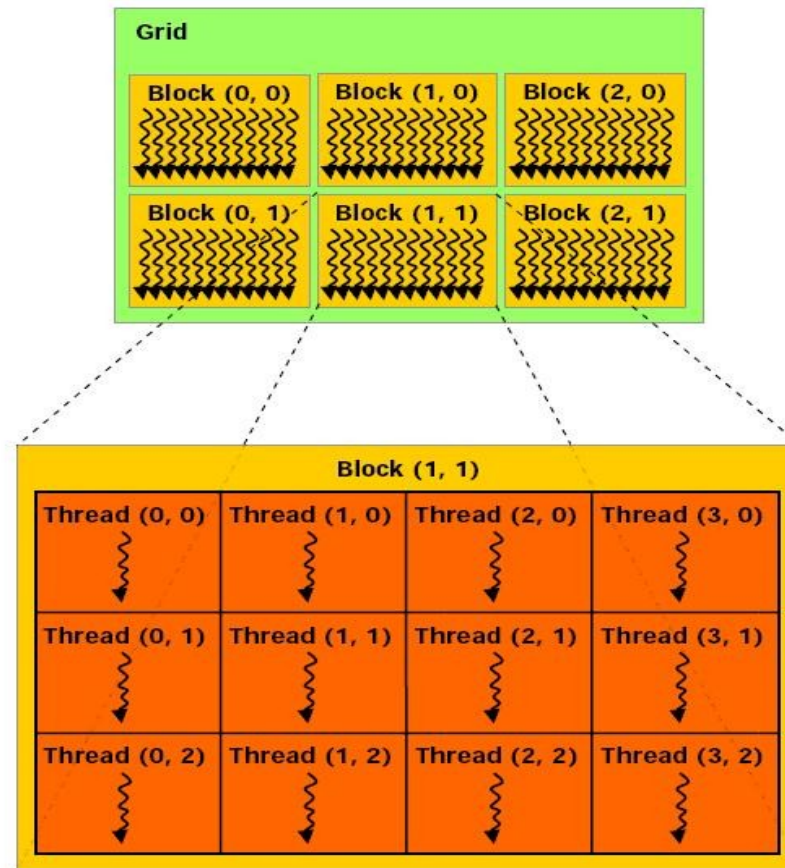


Figure 2-1. Grid of Thread Blocks

## Code the GPU part (device) inside a « kernel »

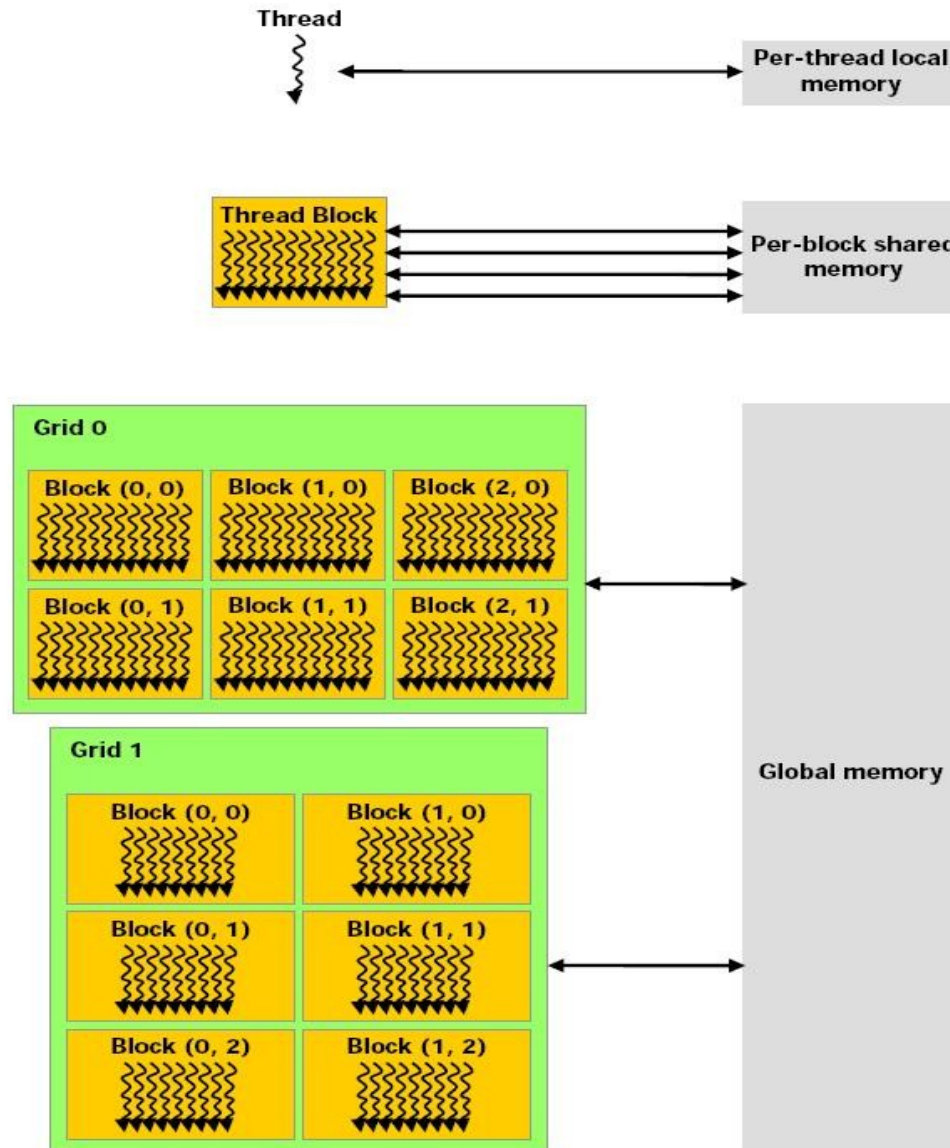
```
__global__ void matAdd(float A[N][N], float B[N][N],  
                      float C[N][N])  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
    if (i < N && j < N)  
        C[i][j] = A[i][j] + B[i][j];  
}
```

## Code the CPU part (host)

```
int main()  
{  
    // Kernel invocation  
    dim3 dimBlock(16, 16);  
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,  
                (N + dimBlock.y - 1) / dimBlock.y);  
    matAdd<<<dimGrid, dimBlock>>>(A, B, C);  
}
```

C extensions -> .cu files compiled with nvcc

# Memory Hierarchy



# Progress of a program

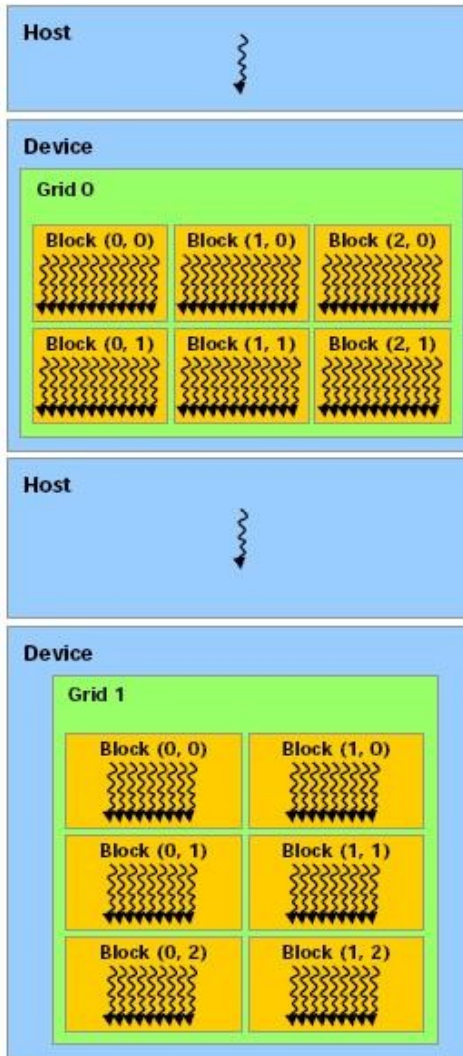
C Program  
Sequential  
Execution

Serial code

Parallel kernel  
Kernel0<<<>>()

Serial code

Parallel kernel  
Kernel1<<<>>()



# Remarks

Kernel programming at a low level (though in C extension)

Keep all data in the GPU, transfer result only at the end

Data: CPU<-> GPU

```
float* ha; ha=cudaMallocHost((void**)ha,...);  
float* da; cudaMalloc(da,.....);  
cudaMemcpy2D(ha,da,...);  
.....  
cudaMemcpy(da,ha,...);
```

# CUBLAS

Blas functions on GPU

Depending on the graphic card, only single precision

User does not need to call cuda functions (wrapper). Standard C code

Blas1 single, double function -> CPU

Blas2-Blas3 result stays in the GPU

# Example: sgemm

```
int main(int argc, char** argv)
{
    cublasStatus status;
    float *h_A, *h_B, *h_C;
    float *d_A, *d_B, *d_C;
    .....
    /* Initialize CUBLAS */
    status = cublasInit();

    /* Allocate host memory for the matrices */
    h_A = (float*)malloc(n2 * sizeof(float));
    h_B = (float*)malloc(n2 * sizeof(float));
    h_C = (float*)malloc(n2 * sizeof(float));
    /* Fill the matrices with test data */
    .....
    /* Allocate device memory for the matrices */
    status = cublasAlloc(n2, sizeof(float), (void**)&d_A);
    status = cublasAlloc(n2, sizeof(float), (void**)&d_B);
    status = cublasAlloc(n2, sizeof(float), (void**)&d_C);

    /* Initialize the device matrices with the host matrices */
    status = cublasSetVector(n2, sizeof(float), h_A, 1, d_A, 1);
    status = cublasSetVector(n2, sizeof(float), h_B, 1, d_B, 1);
    status = cublasSetVector(n2, sizeof(float), h_C, 1, d_C, 1);

    /* Performs sgemm using cublas */
    cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);

    /* Read the result back */
    status = cublasGetVector(n2, sizeof(float), d_C, 1, h_C, 1);

    /* Host and device memory clean up */
    .....
    /* Shutdown */
    status = cublasShutdown();
    return 0;
}
```

# Results

## SGEMM

### LINEAR ALGEBRA TEST ON GPU

#### HARDWARE

##### Lenovo T61:

Intel Core Duo 2.20 Ghz, 4Mb

Nvidia Quadro 140M 2x8=16 cores 0.8Ghz

##### Imac

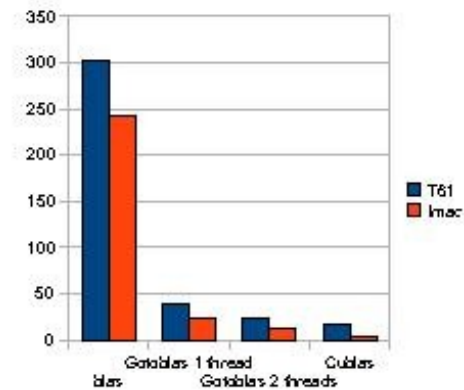
Intel Core Duo 3.06 Ghz, 4Mb

Nvidia Geforce 8800 GS 8x8=64 cores 0.4GHz

SGEMM n=512, 2000 iterations

Time (s)

	T61	Imac
blas	302	243
Gotoblas 1 thread	39.5	24.7
Gotoblas 2 threads	22.9	13.1
Cublas	17.2	3.27



# MAGMABLAS

Subset of Cublas

More optimized:

Auto-tuners

Pointer redirecting (remove performance oscillations)

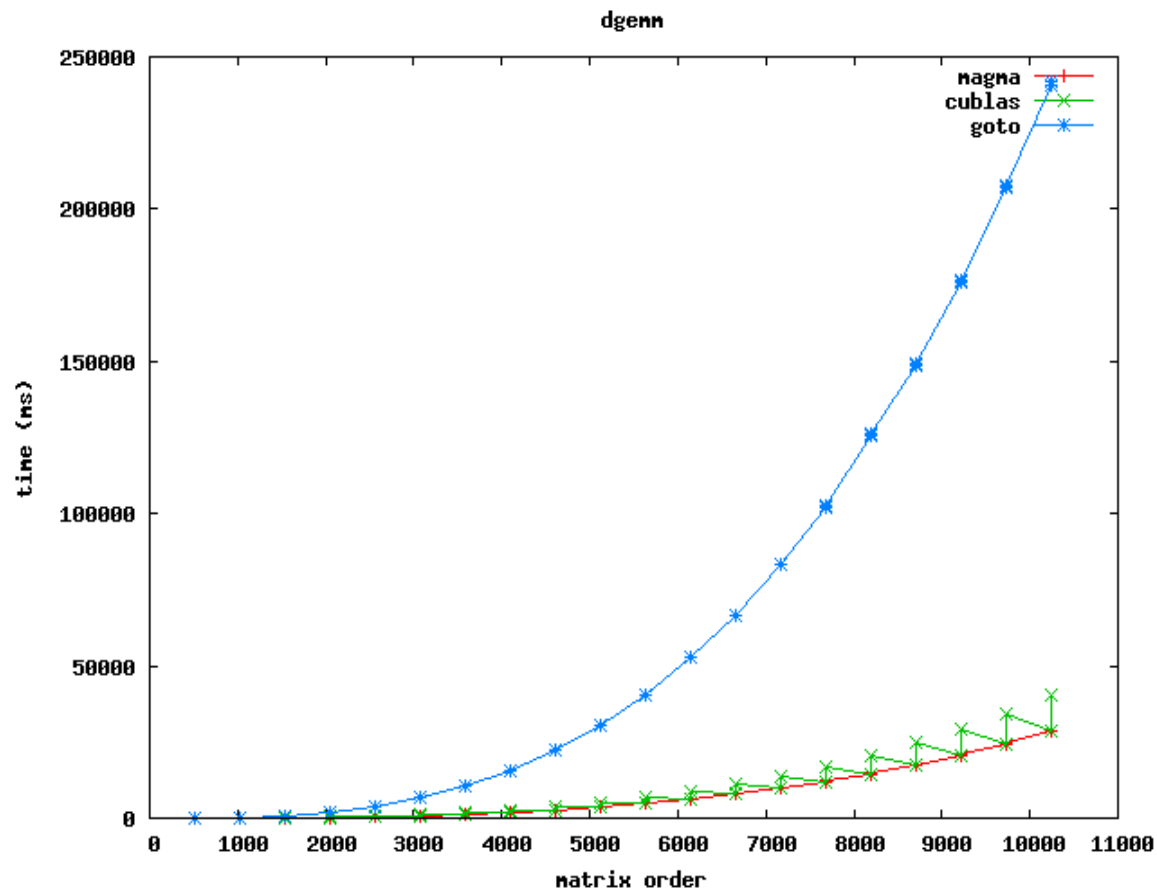
sgemv, dgemv

Sgemm, dgemm

....

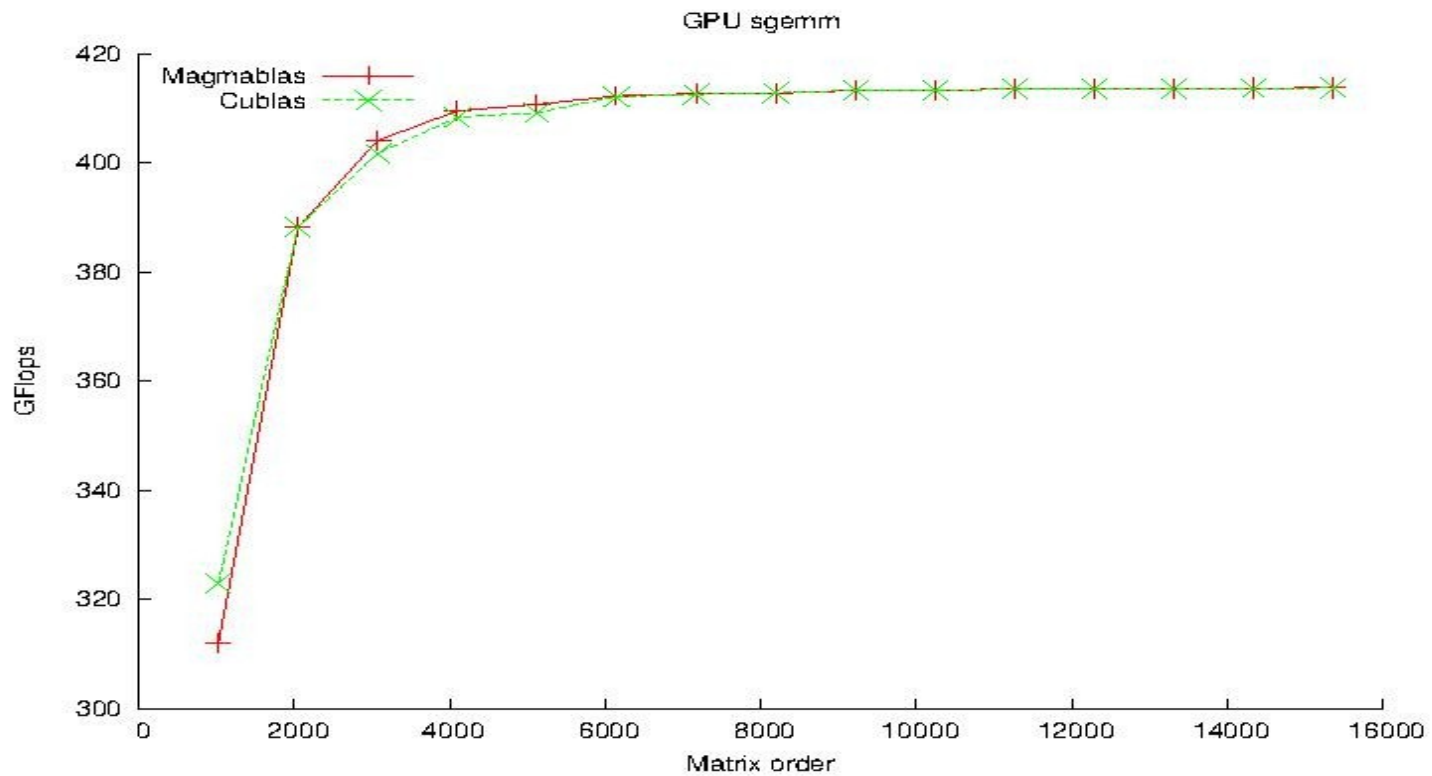
# Results

Dit-exp-cluster: 4 8cores-nodes, each 2 Nvidia Tesla S1070  
(4GB, 4x240=960 cores)

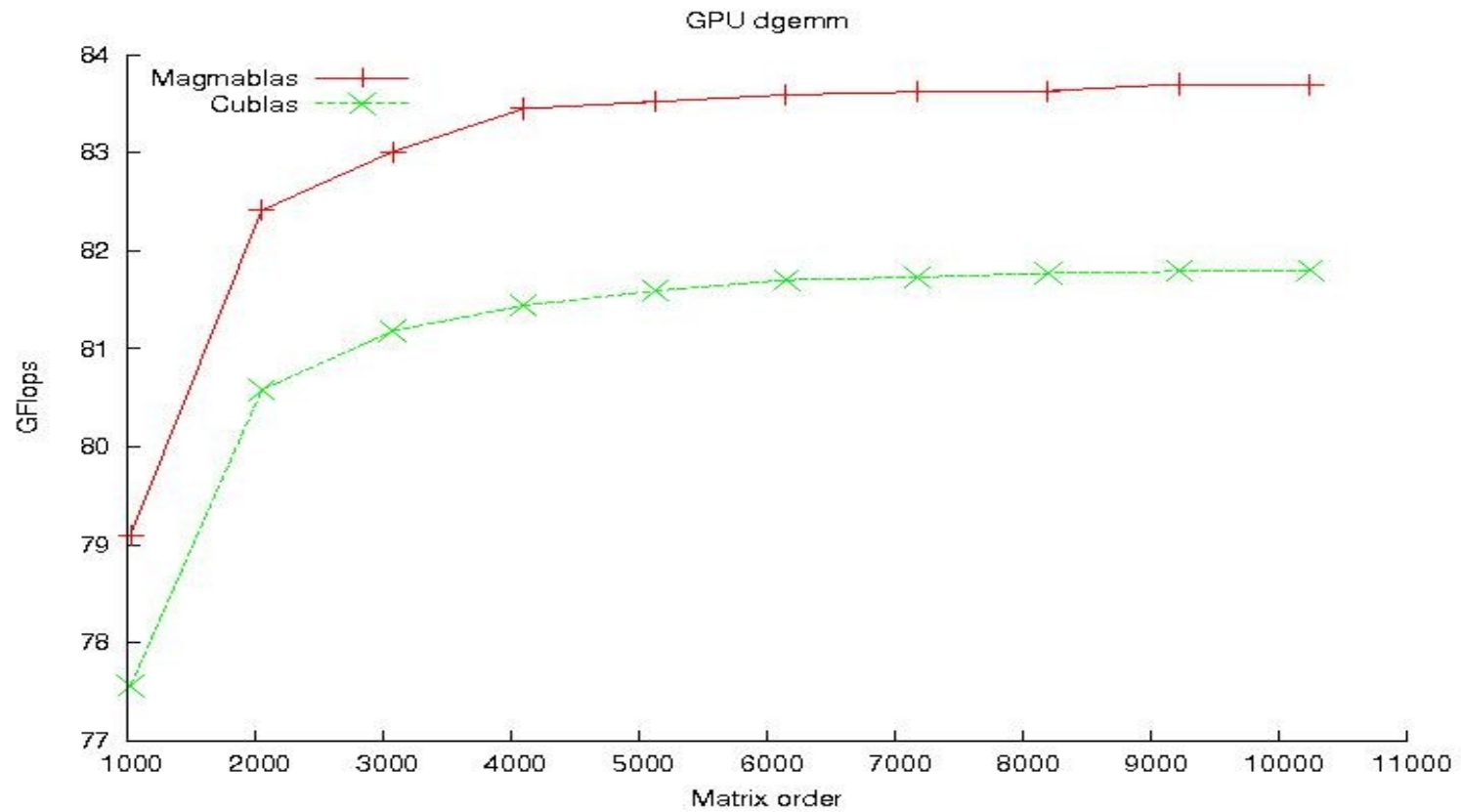


# Electra

24 8cores-nodes, each 2 Nvidia Tesla T10  
(4GB, 30MPx8cores=240 cores)



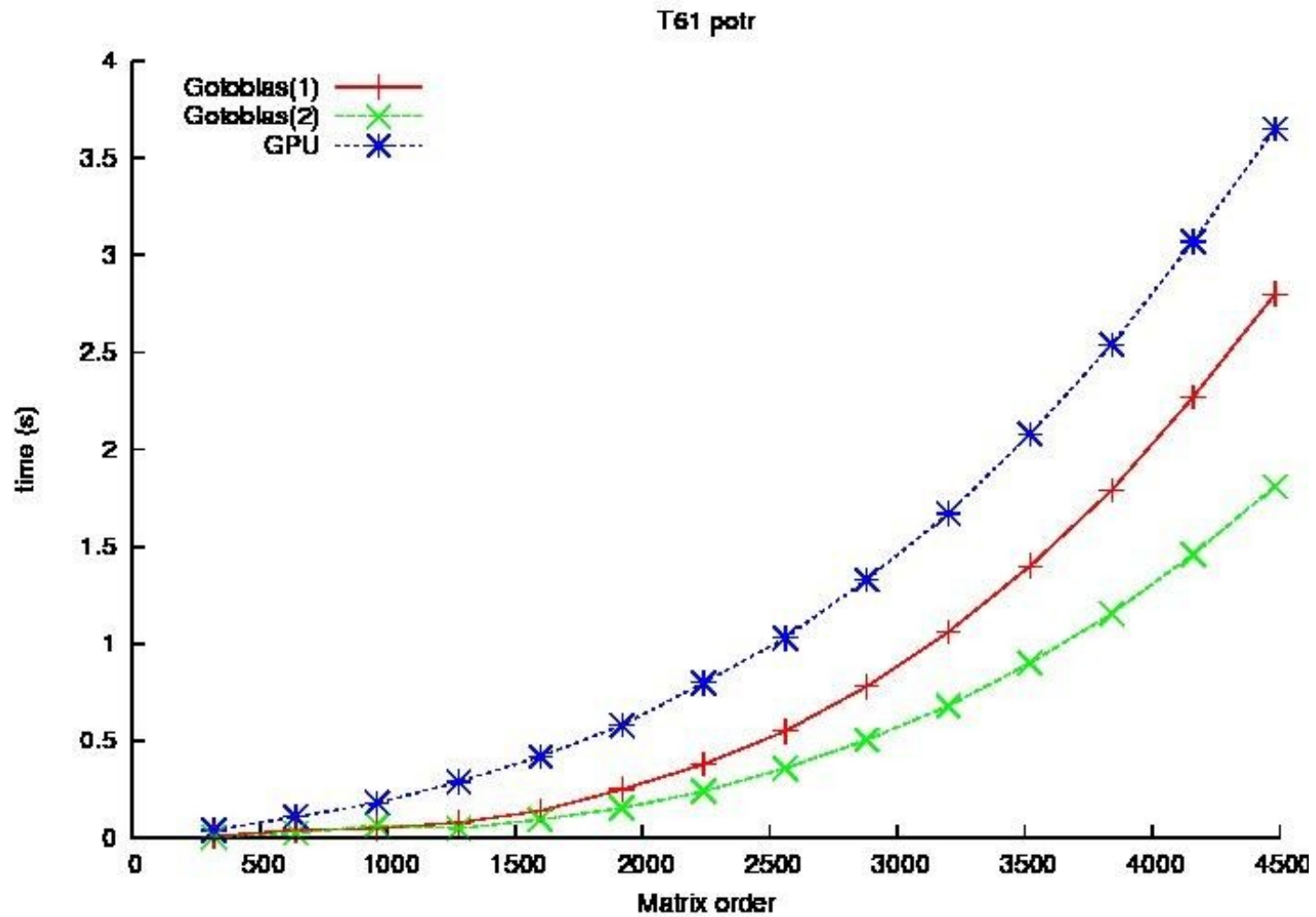
# Electra dgemm



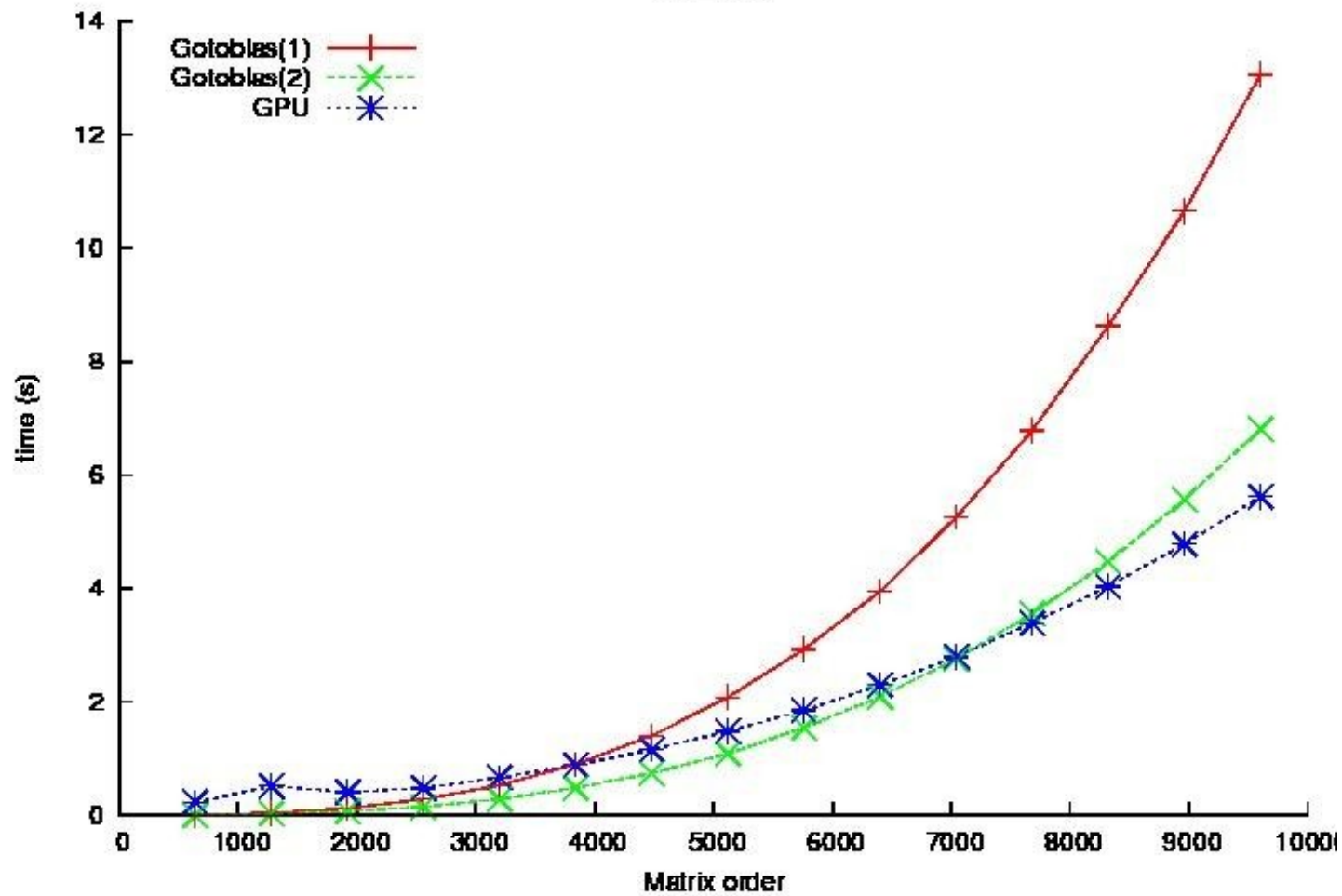
# Lapack

- ◆ Two years ago: no gpu version
- ◆ Replace blas functions by the ones of cublas in the Lapack code. All arrays stay in the gpu
- ◆ Equivalent to use the gpu as a coprocessor
- ◆ Problem: with some cards, only single precision!

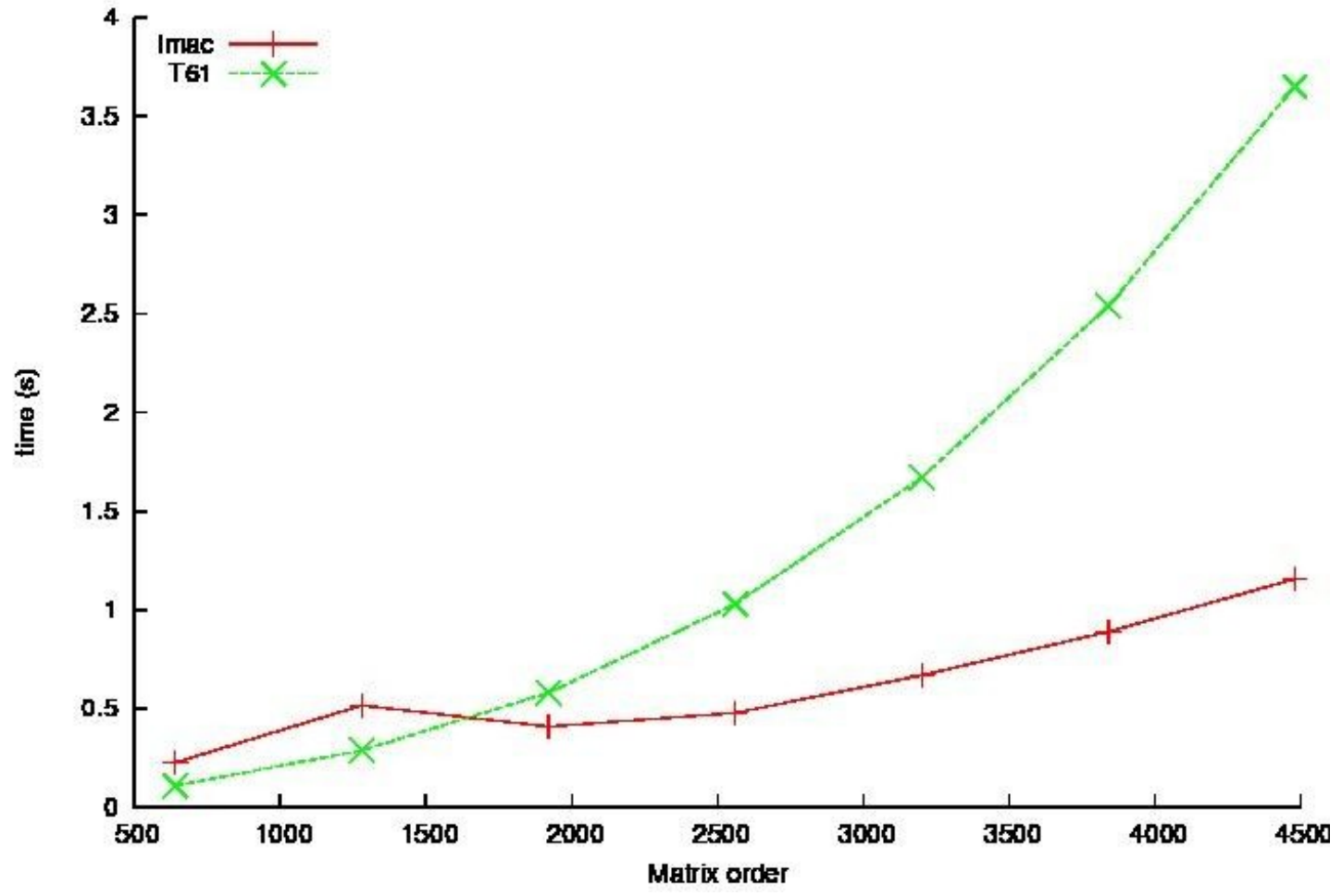
# POTR: $Ax=b$ , $A$ sdp



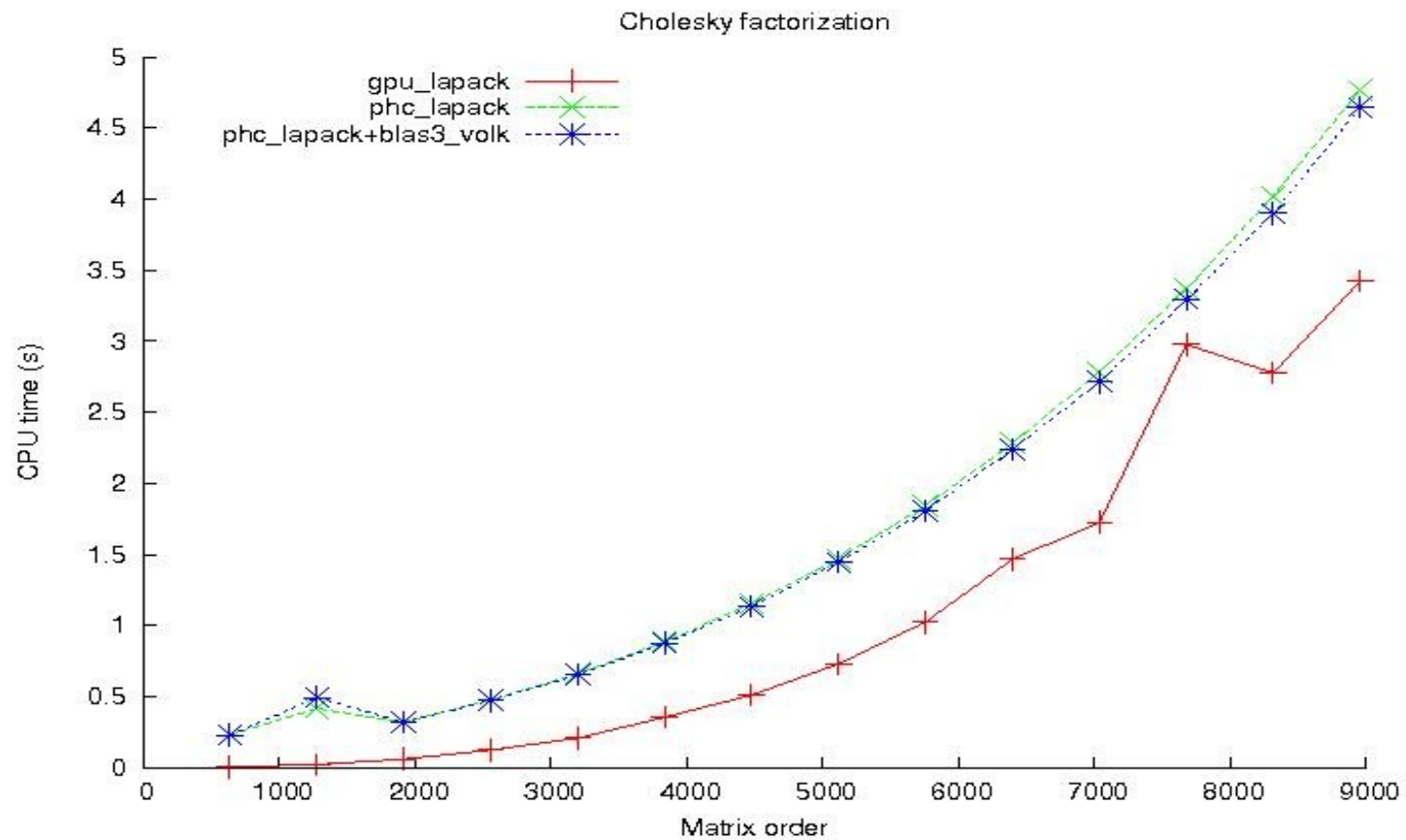
Imac potr



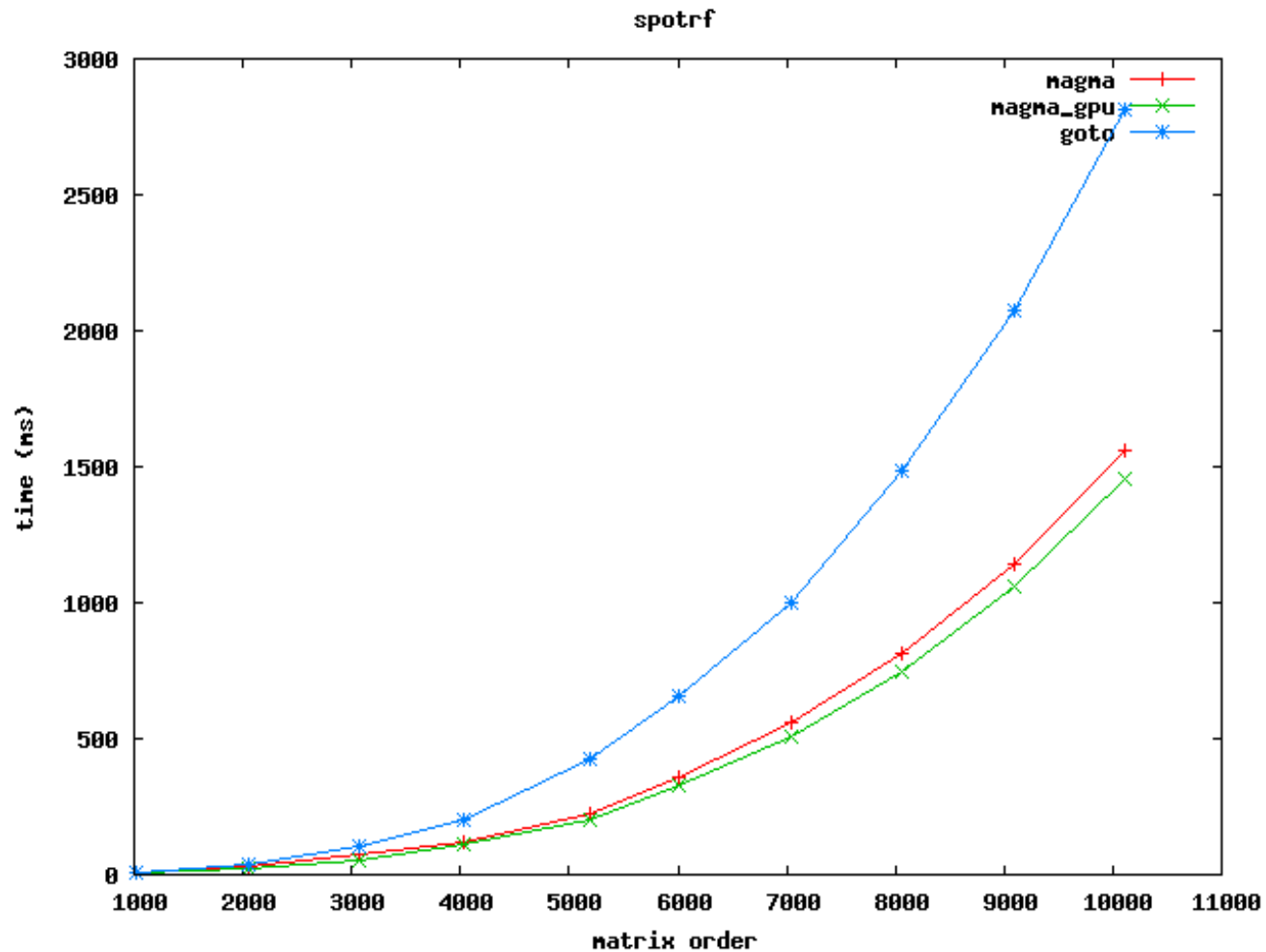
GPU ptr

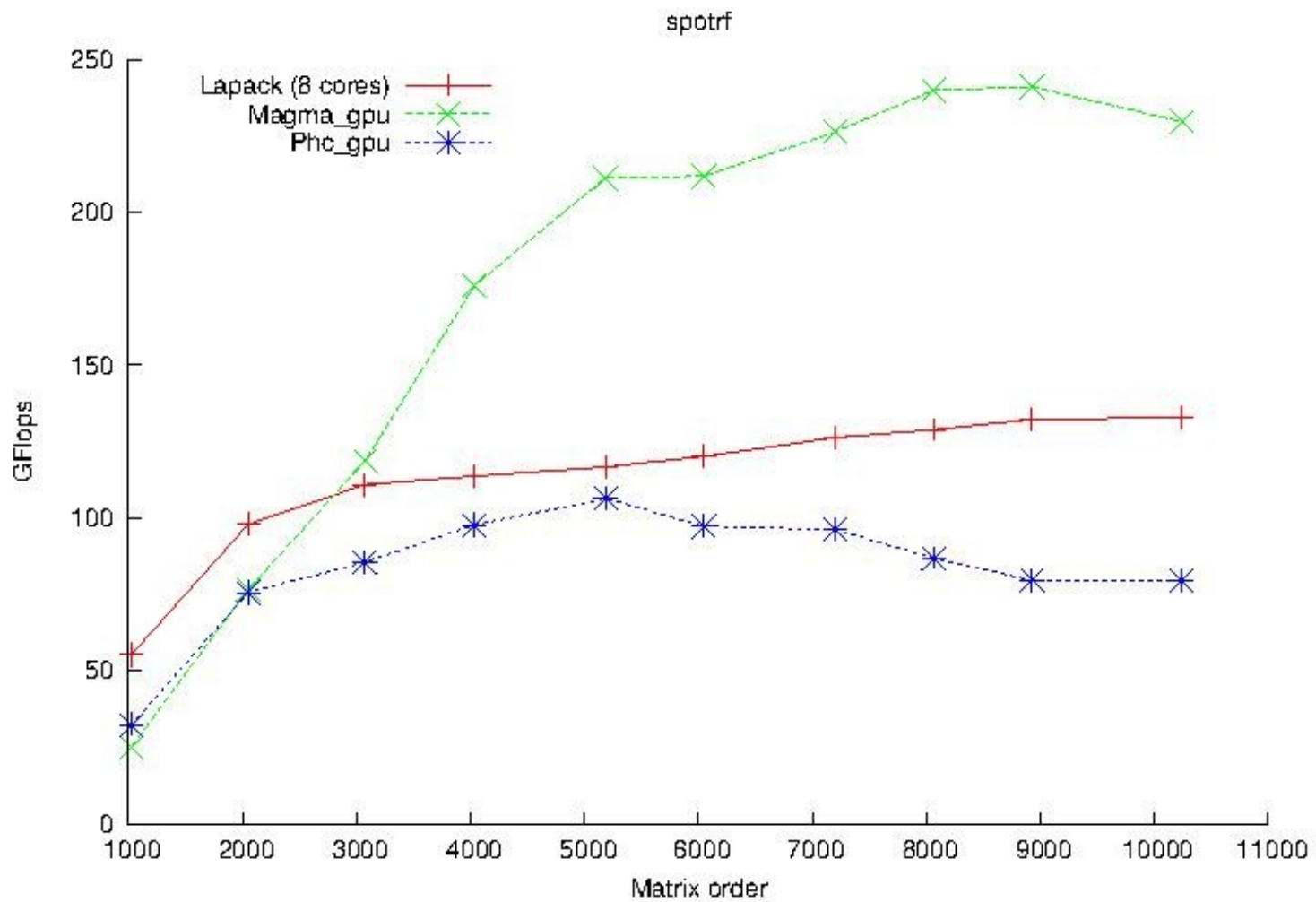


# PhC\_lapack <-> Cula

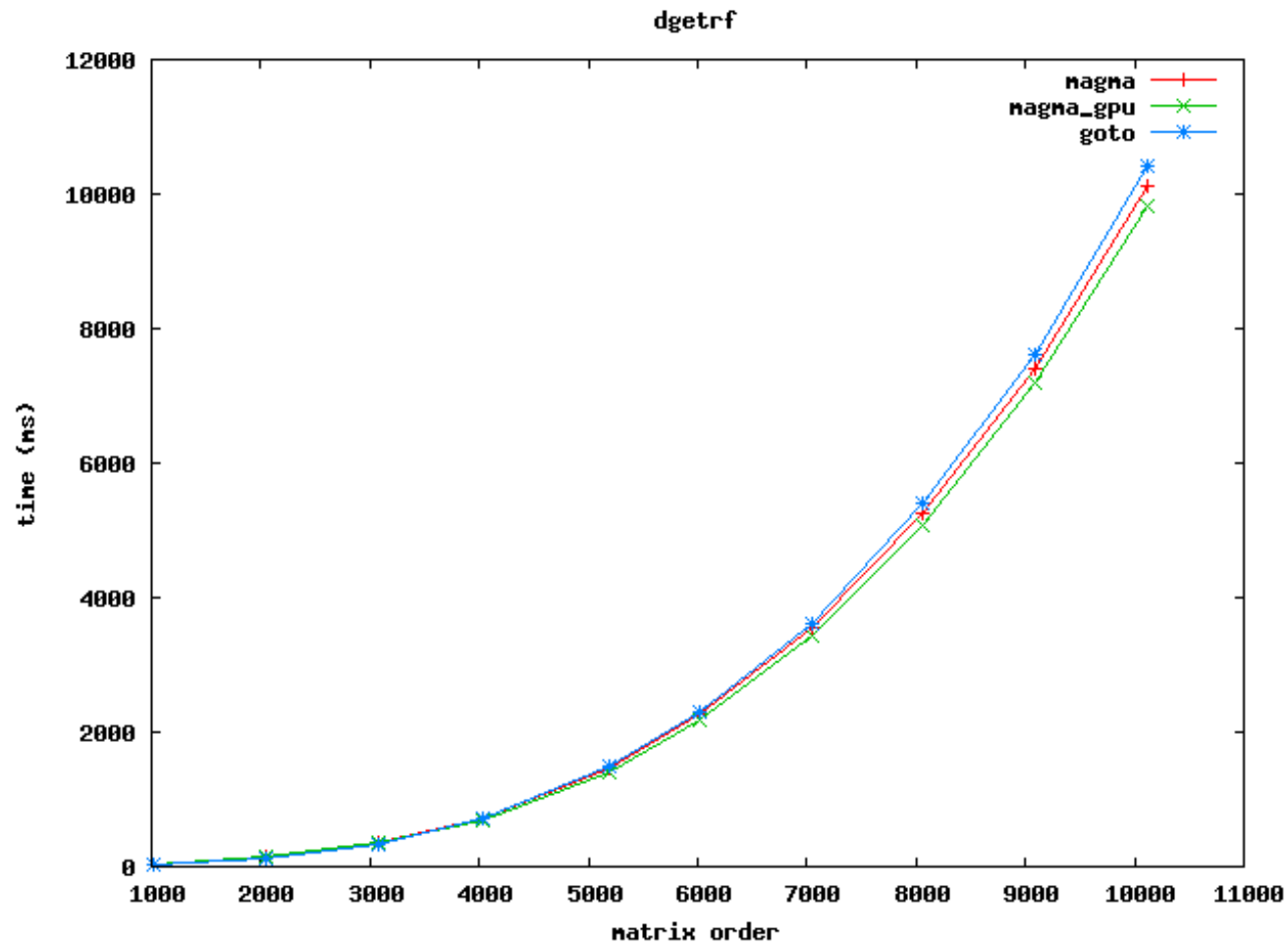


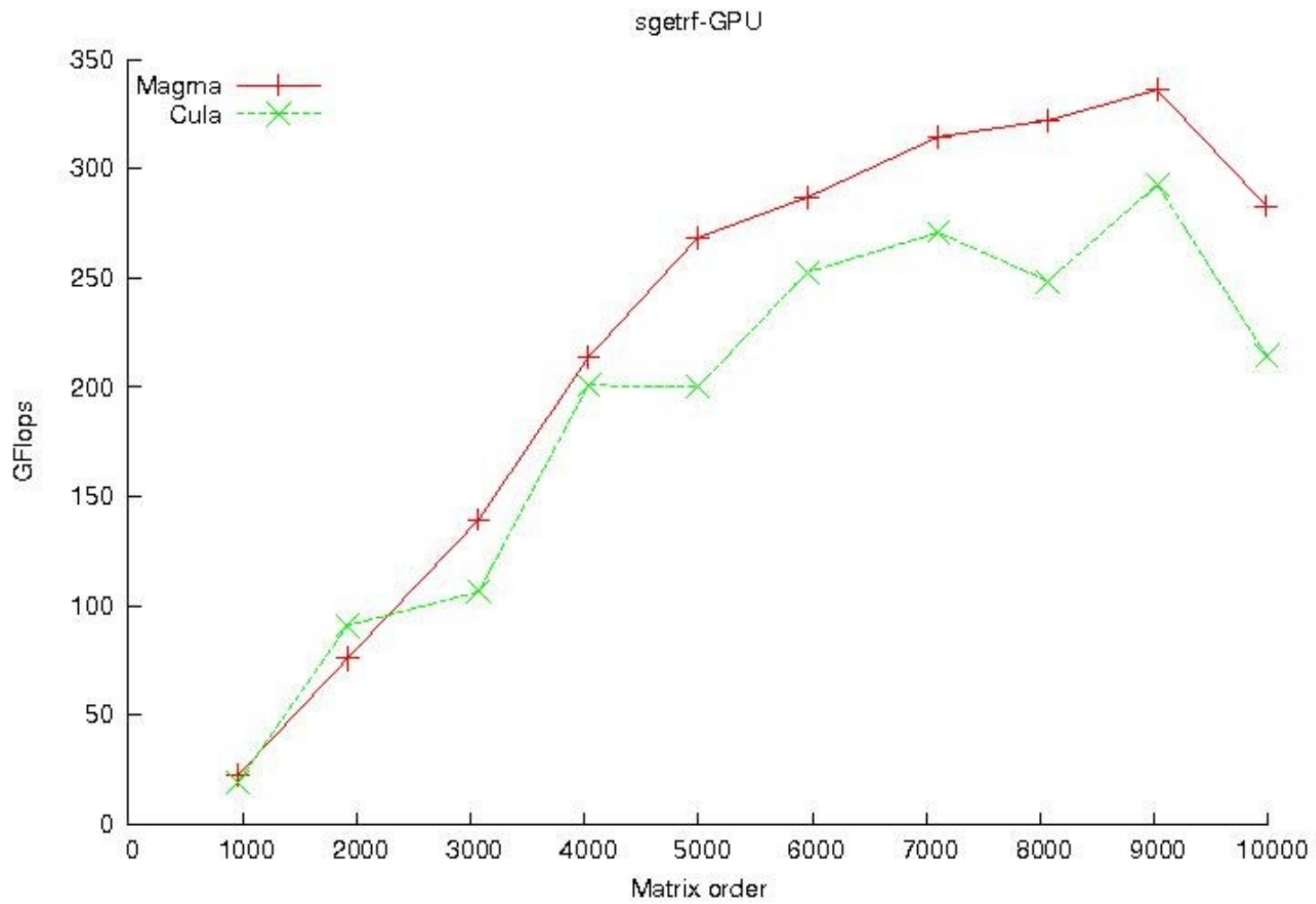
# Cula<->Magma Cholesky





# LU Decomposition

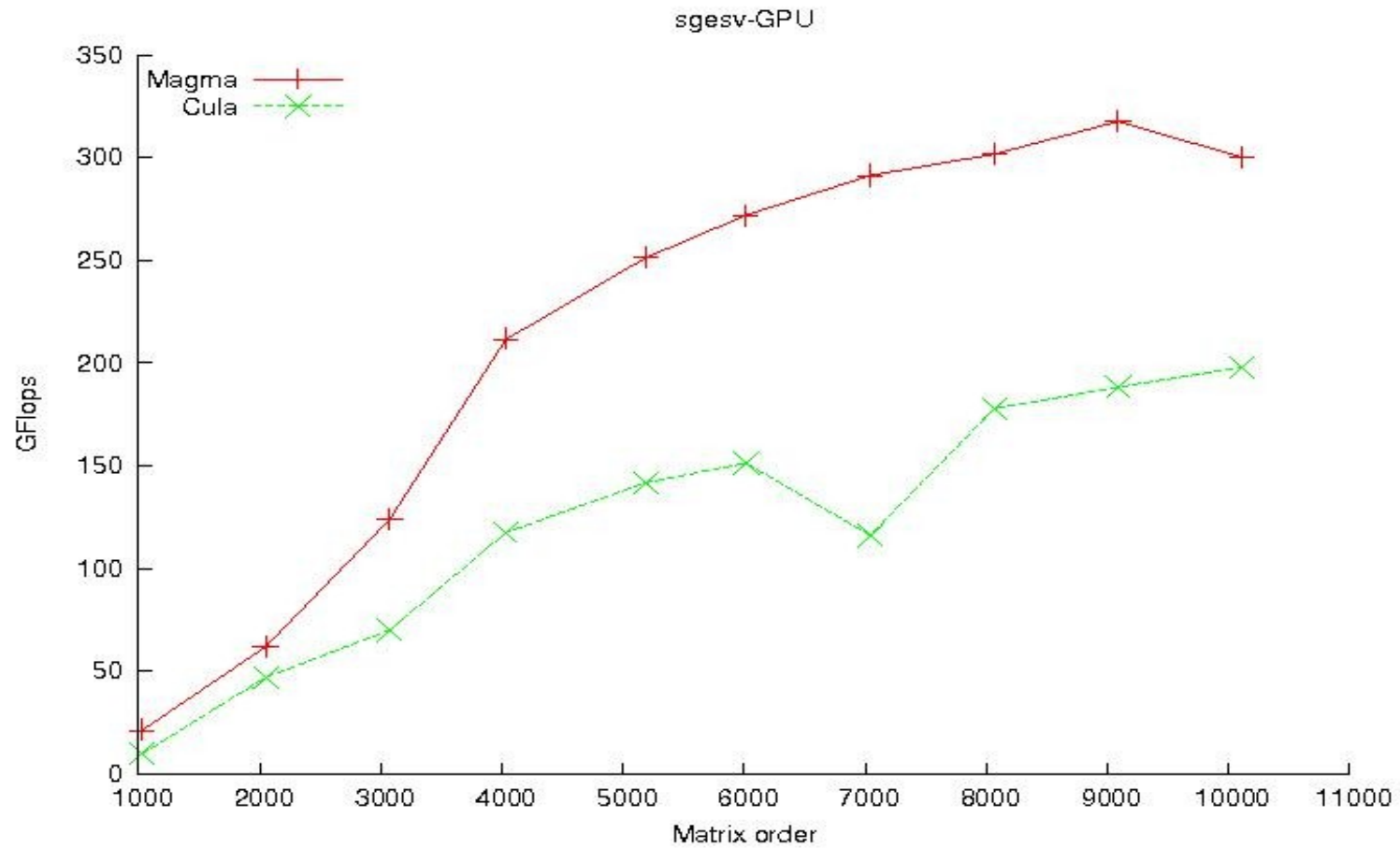




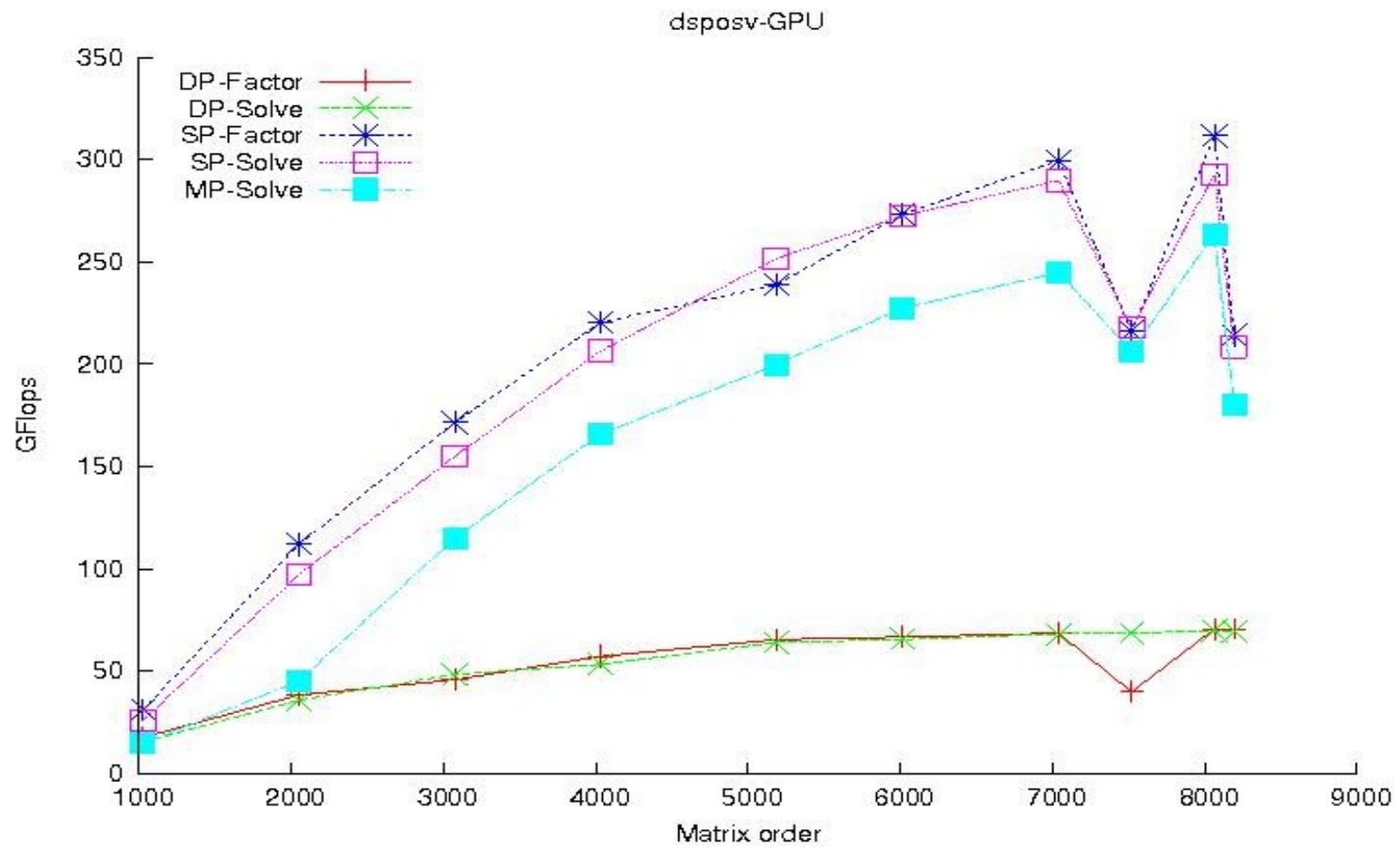
# Matrix factorization

- ◆ GPU better than CPU with few cores for large  $n$ :  
Expected, because block algorithms better for large  $n$
- ◆ The GPU with more processors is more efficient for  $n$  large
- ◆ CPU with many cores almost as good as GPU, but much more money expensive

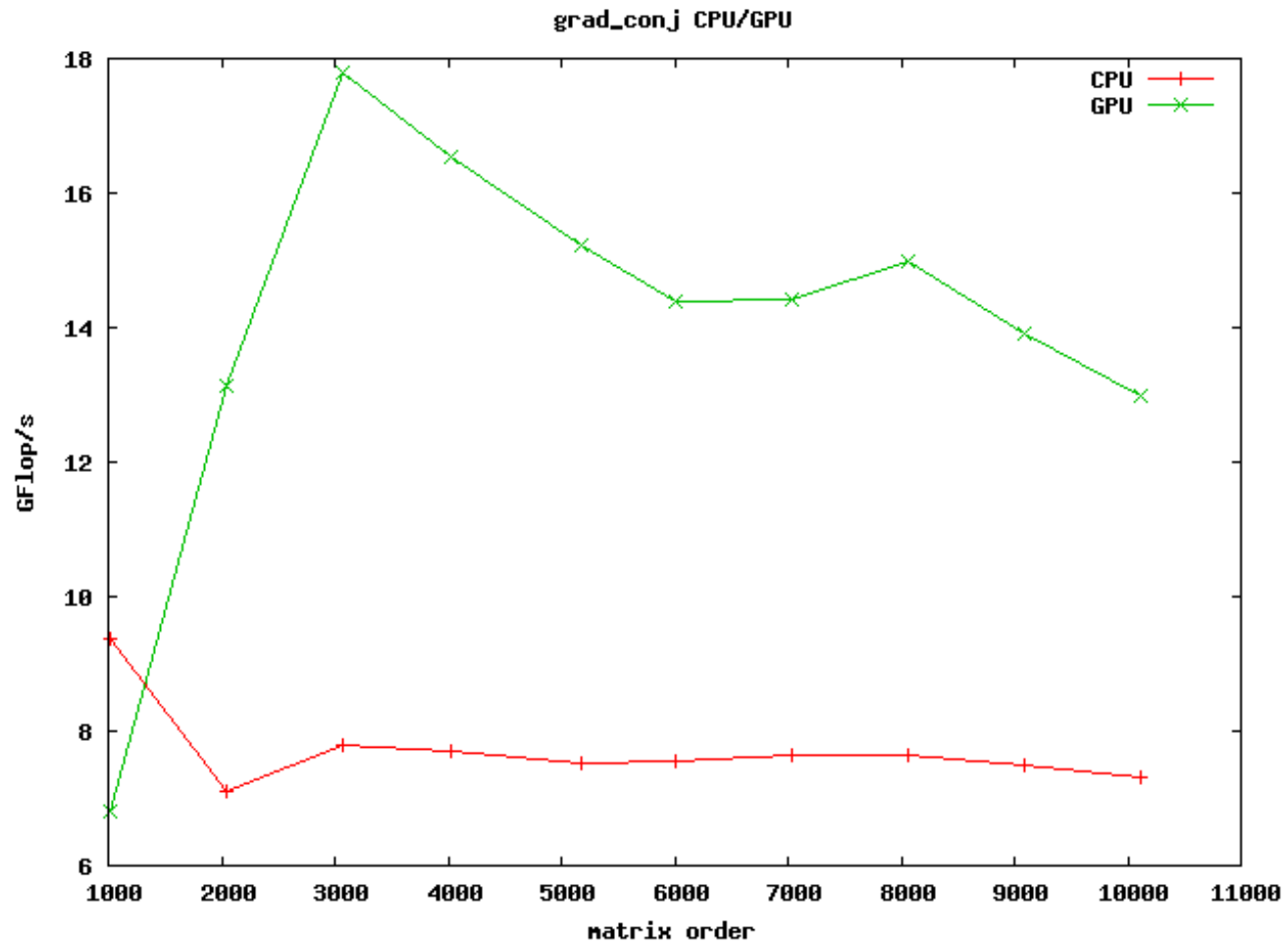
# Linear system



# Sposv, Dposv



# Conjugate gradient



# Conclusion

- ◆ Complete Lapack GPU?
- ◆ For maximal efficiency: code the GPU!
- ◆ For linear algebra (and EDP solving):  
Not enough basic software yet!
- ◆ Molecular dynamics starts quite strongly  
Namd, Amber pmemd