Toward Real-Time Simulation of Cardiac Dynamics

Ezio Bartocci

SUNY Stony Brook

Joint work with

E. Cherry, J. Glimm, R. Grosu, S. A. Smolka, F. Fenton

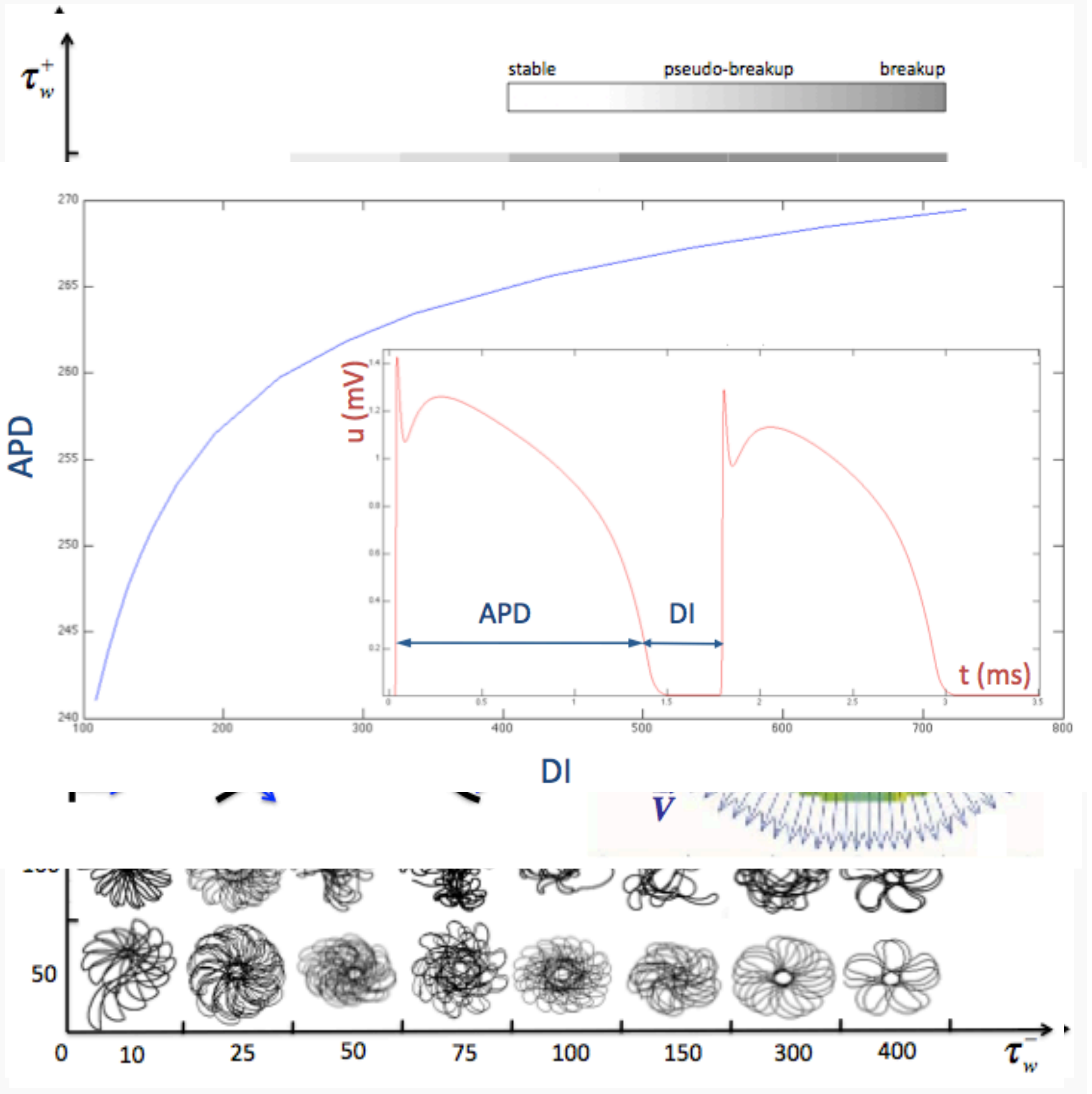
Outline

- Motivation
- Cardiac Models as Reaction Diffusion Systems
- CUDA Programming Model
- Reaction Diffusion in CUDA
- Case Studies
- Work in Progress

Motivation for our Work

Simulation-Based Analysis

- Spiral Formation
- Spiral Breakup
- Tip Tracking
- Front Wave Tracking
- Curvature Analysis
- Conduction Velocity
- Restitution Analysis



Cardiac Models as Reaction Diffusion Systems

Membrane's AP depends on:

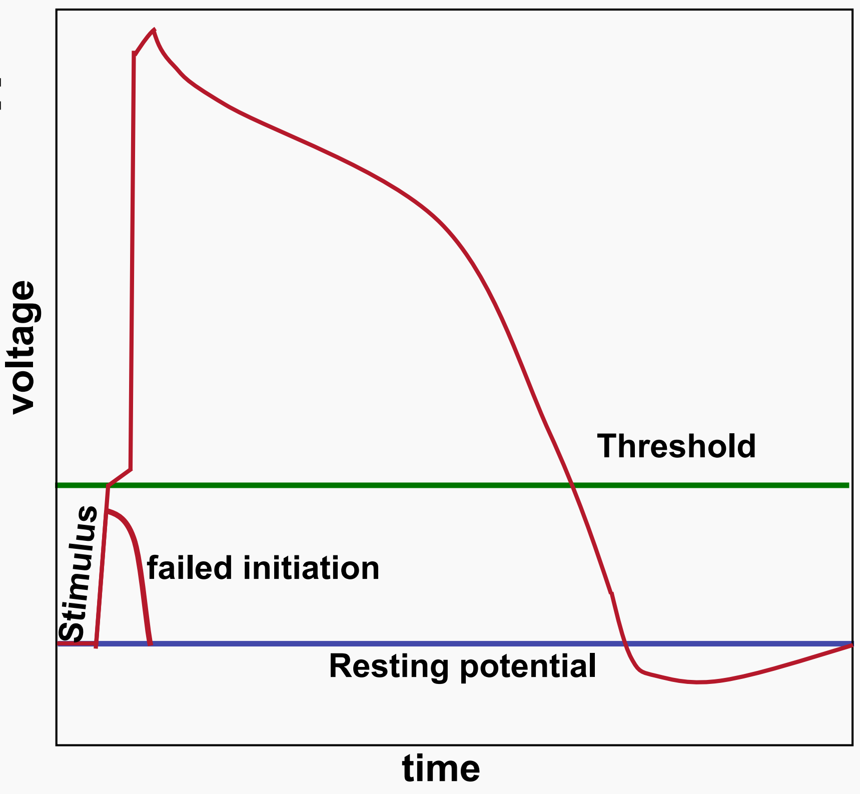
- **Stimulus (voltage or current):**
  - External / Neighboring cells
- **Cell's state**

AP has nonlinear behavior!

- **Reaction diffusion system:**

$$\underbrace{\frac{\partial \mathbf{u}}{\partial t}}_{\text{Behavior In time}} = \underbrace{R(\mathbf{u})}_{\text{Reaction}} + \underbrace{\nabla(D\nabla\mathbf{u})}_{\text{Diffusion}}$$

Schematic Action Potential

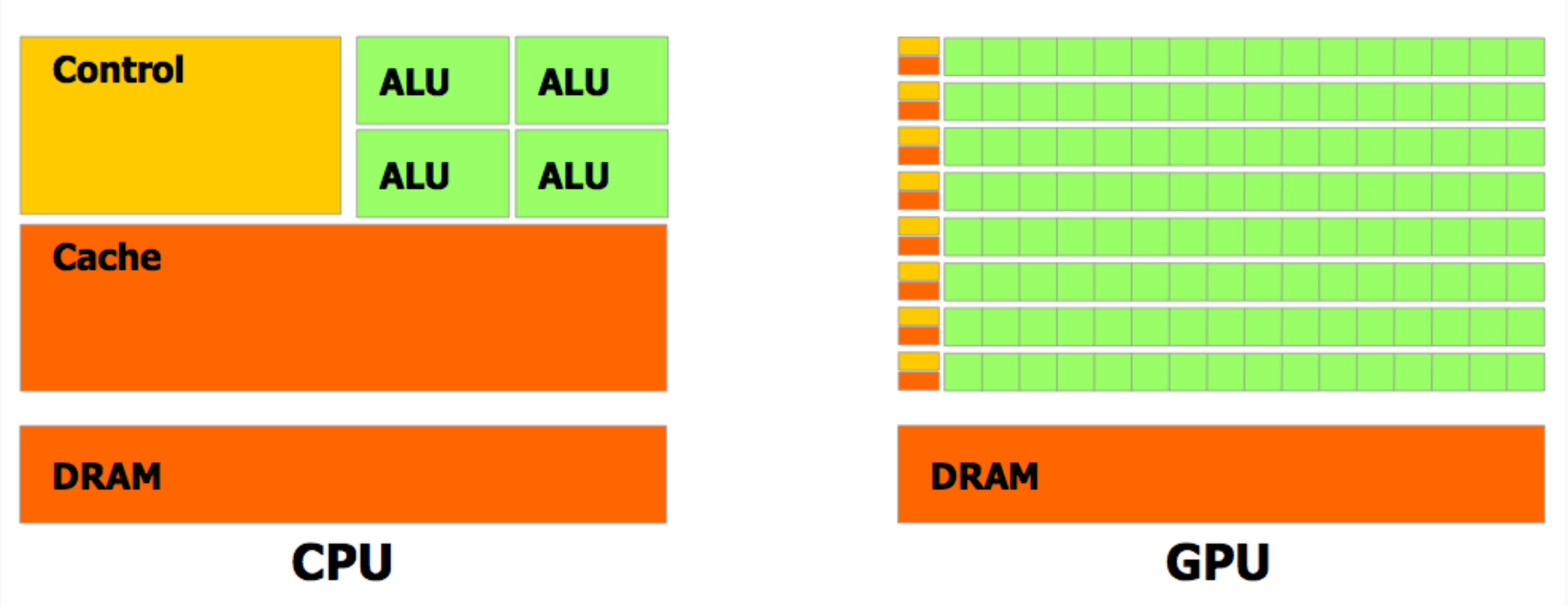


Cardiac Models

- Minimal Model (Flavio-Cherry) (4 v) Human
- Beeler-Reuter (8 v) Canine
- Ten Tussher Panfilov (19 v) Human
- Iyer (67 v) Human

Available Technologies

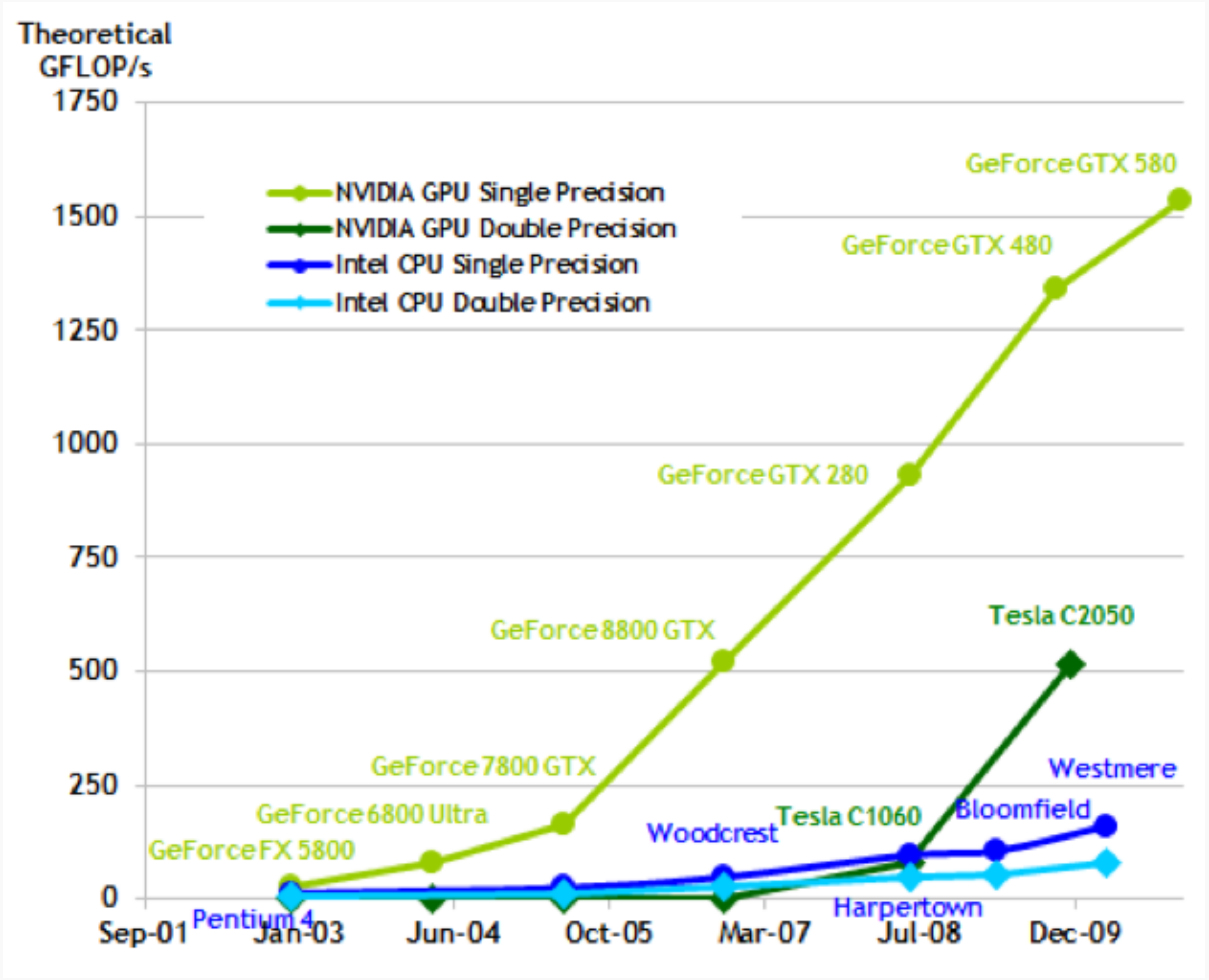
CPU based **GPU based**



The GPU devotes more transistors to data processing

This image is from CUDA programming guide

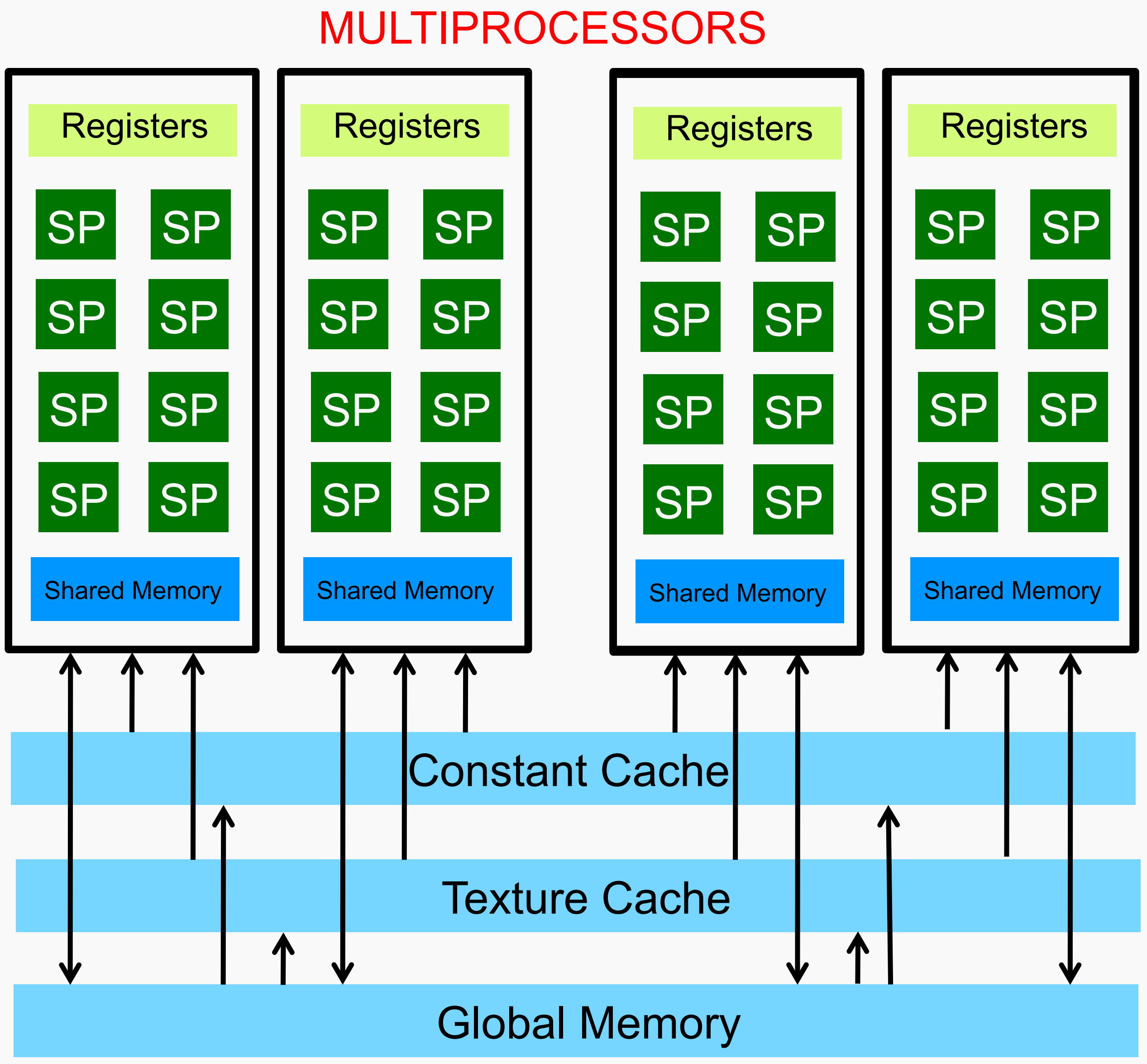
GPU vs CPU



Theoretical GFLOP/s
1750
1500
1250
1000
750
500
250
0
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Westmere
Bloomfield
Woodcrest
Harpertown
Pentium 4

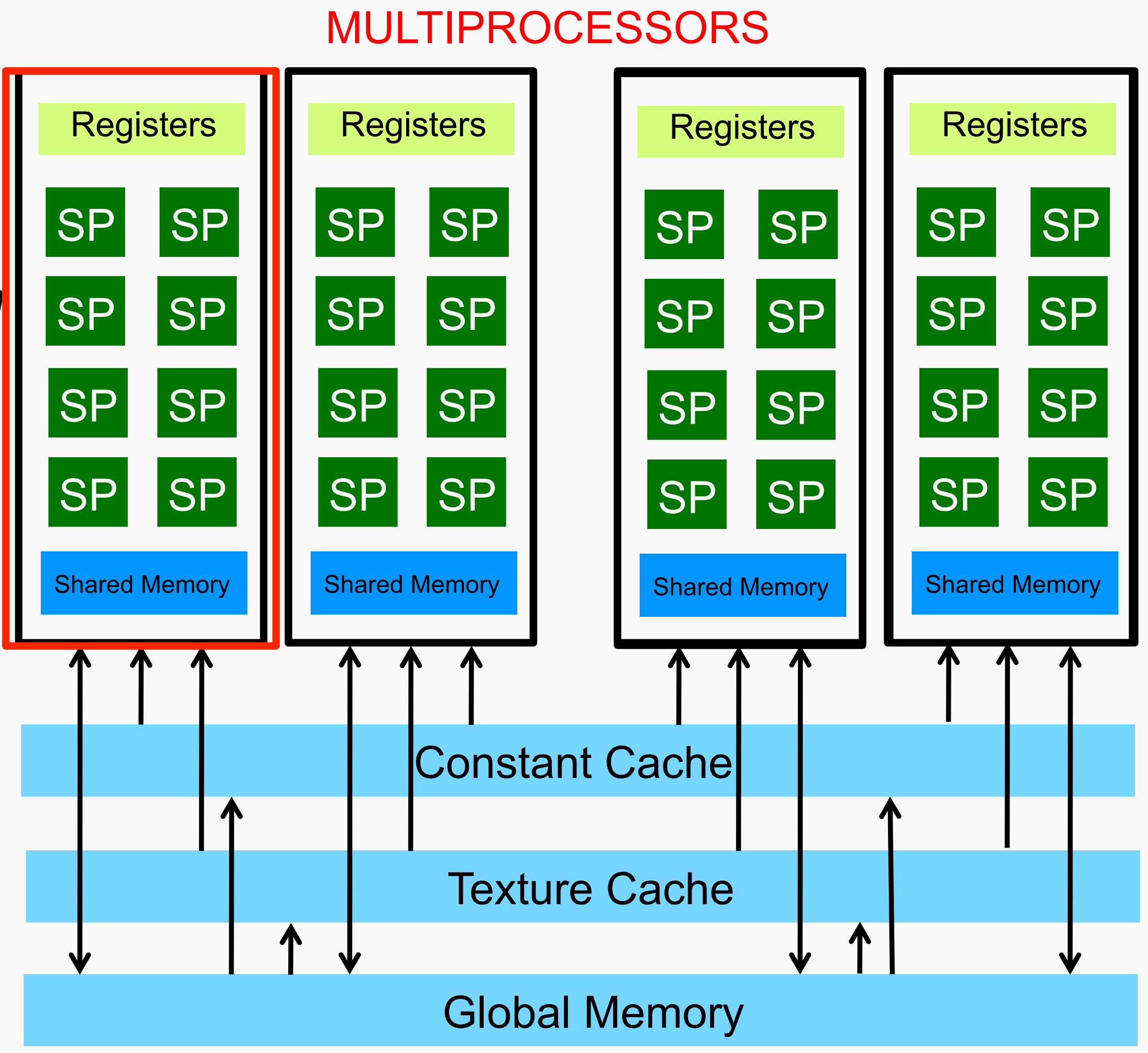
GPU Architecture

Each GPU consists of a Set of multiprocessors.



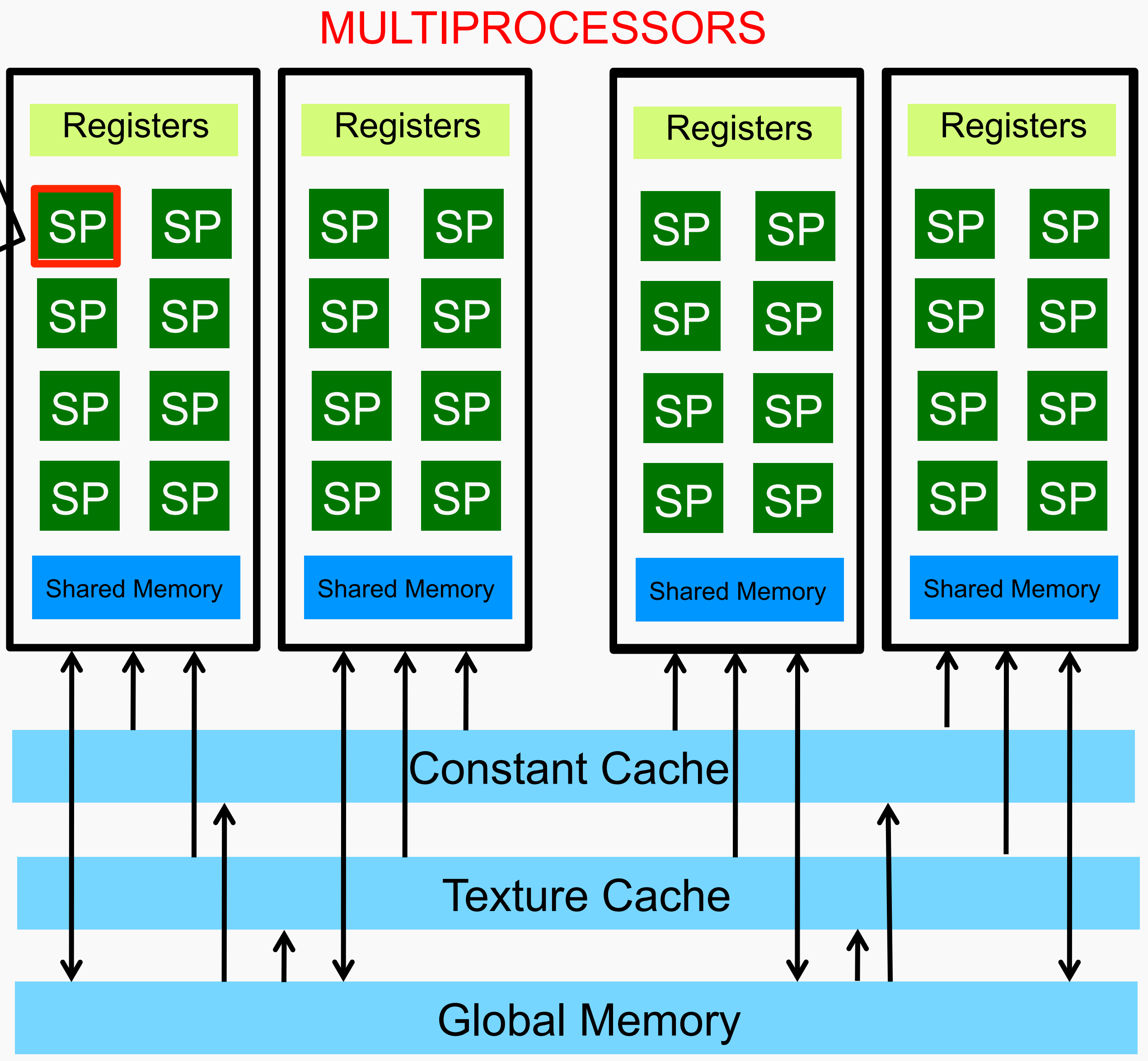
GPU Architecture

Each Multiprocesssor can have 8/32 Stream Processors (SP) (called by NVIDIA also cores) which share access to local memory.

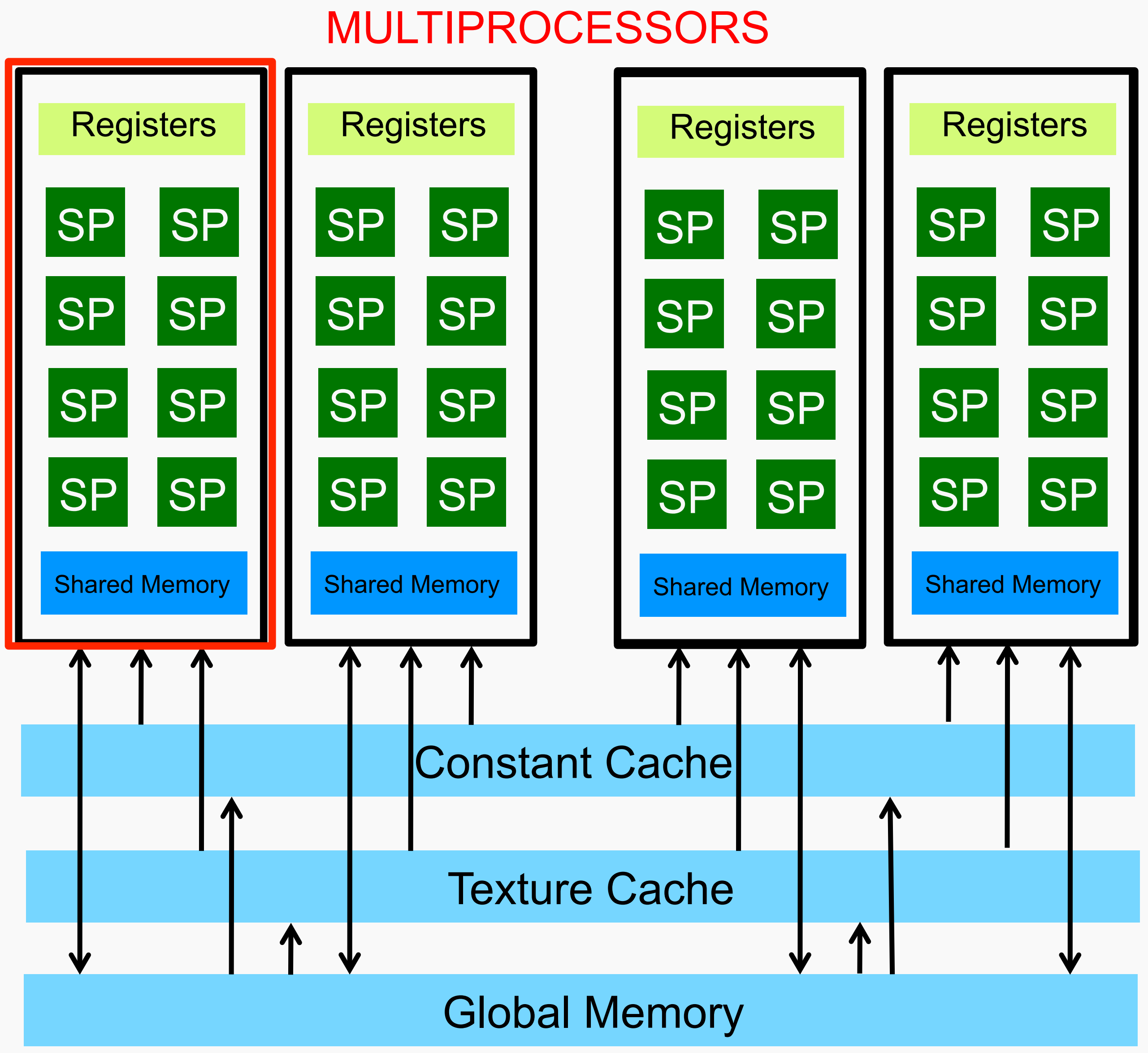


GPU Architecture

Each Stream Processor (core) contains a fused multiply-adder capable of single precision arithmetic. It is capable of completing 3 floating point operations per cycle - a fused MADD and a MUL.



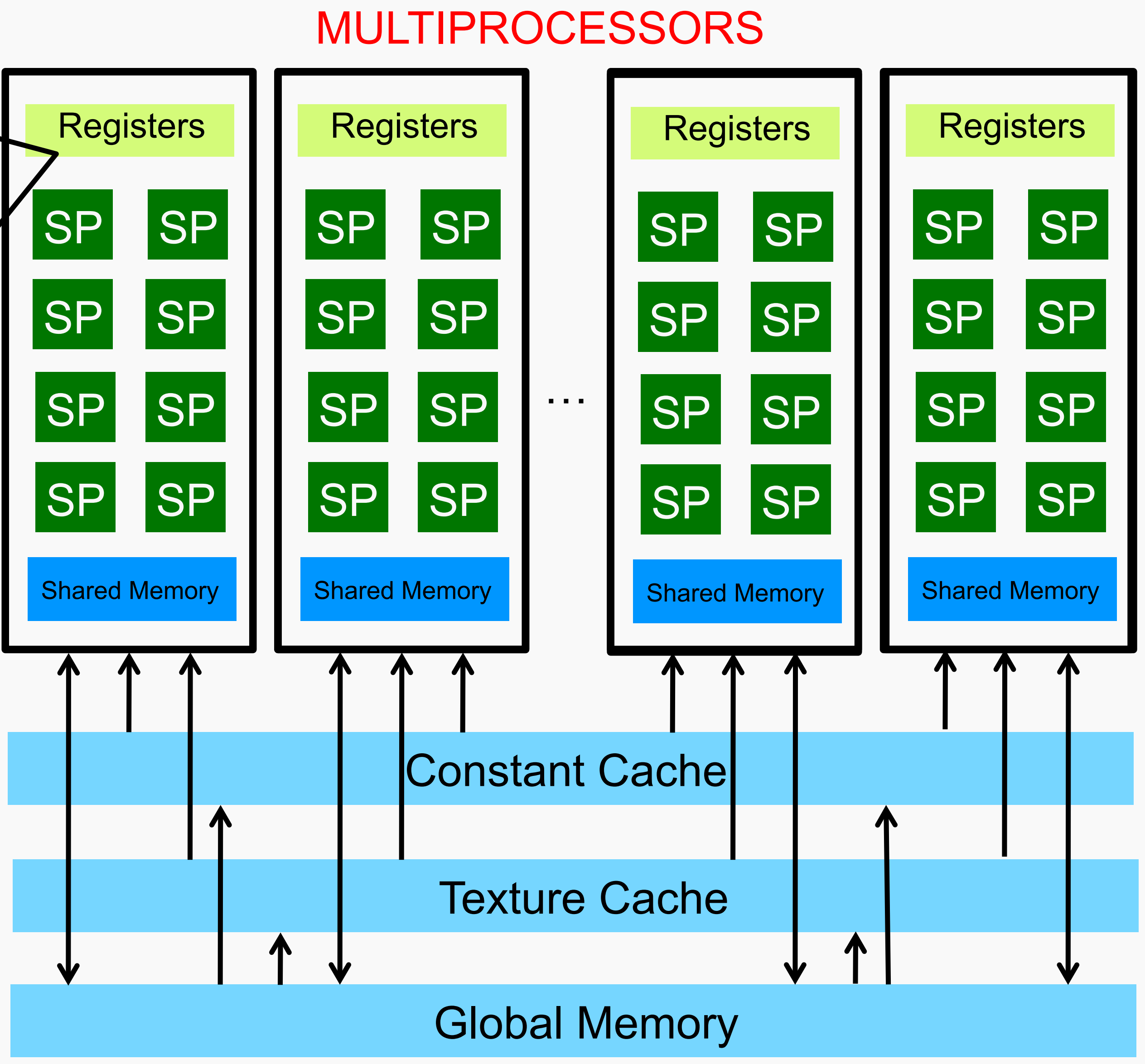
GPU Architecture



Each multiprocessor can contain one or more 64-bit fused multiple adder for double precision operations.

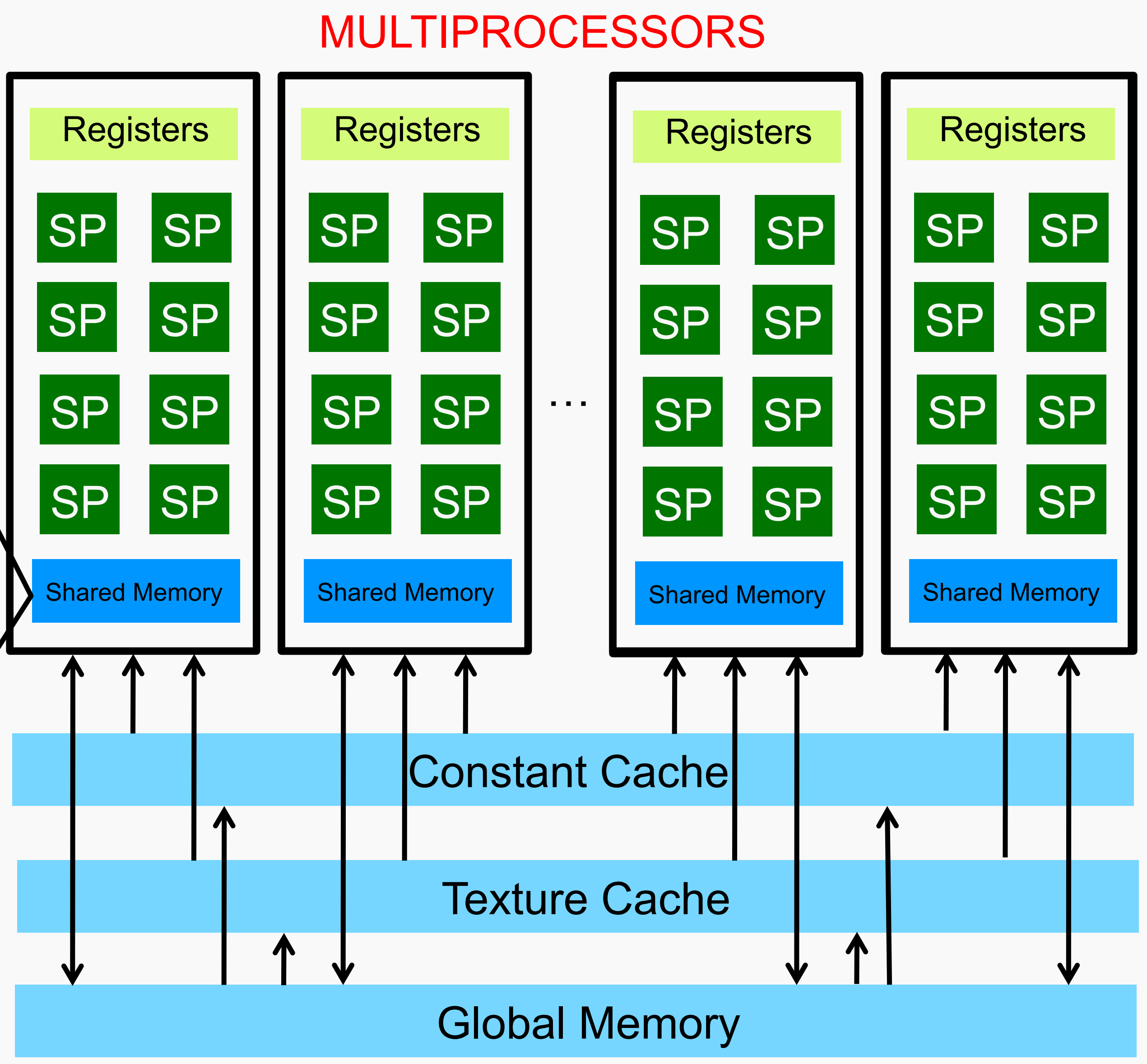
Memory Hierarchy

The fastest available Memory for GPU computation is device registers. Each multiprocessor contains 16KB of registers. The registers are partitioned among the MP-resident threads

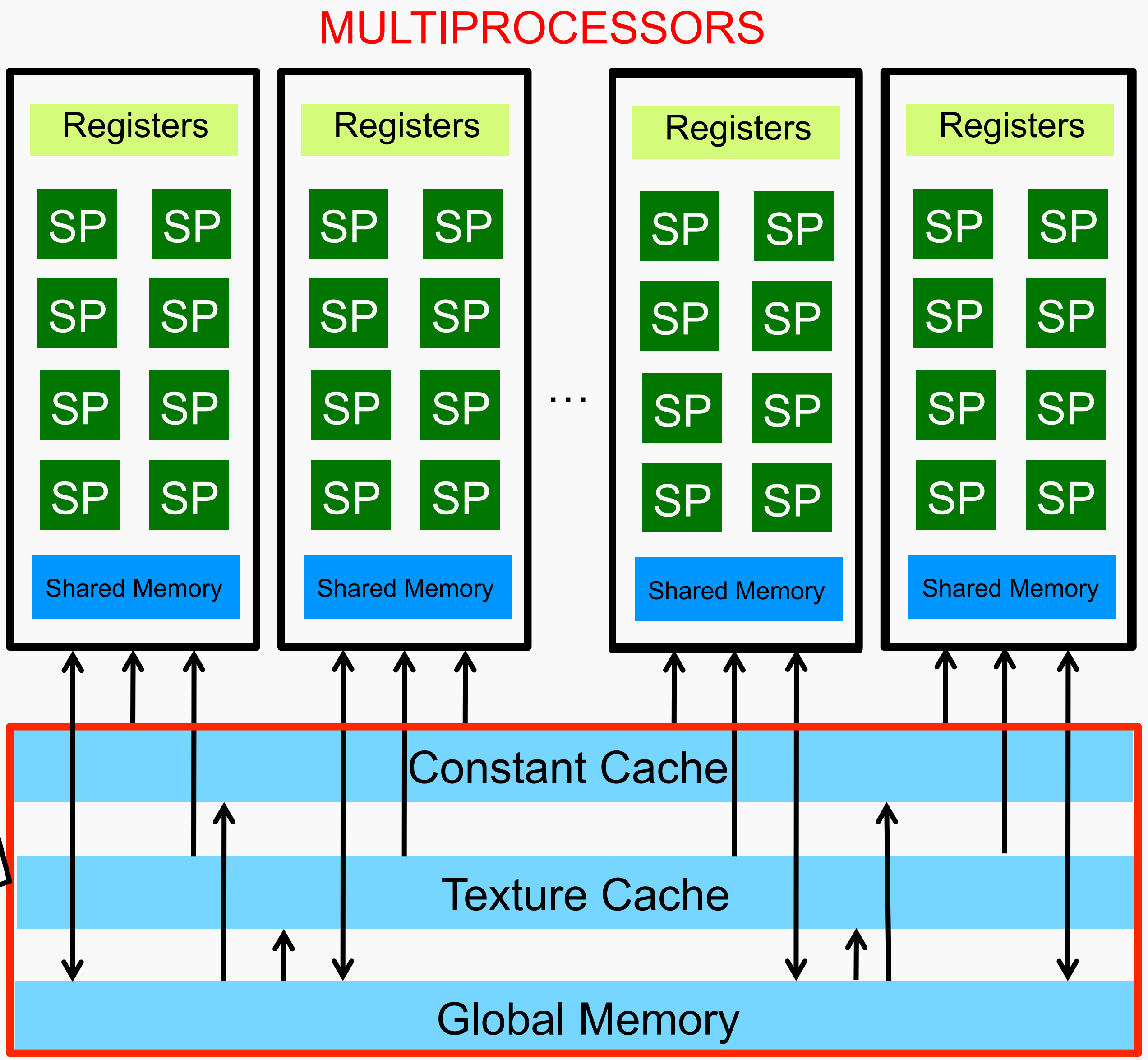


Memory Hierarchy

Shared memory (16KB) is primarily intended as a means to provide fast communication between threads of the executed by the same multiprocessor, although, due to its speed, it can also be used as a programmer controlled memory cache.

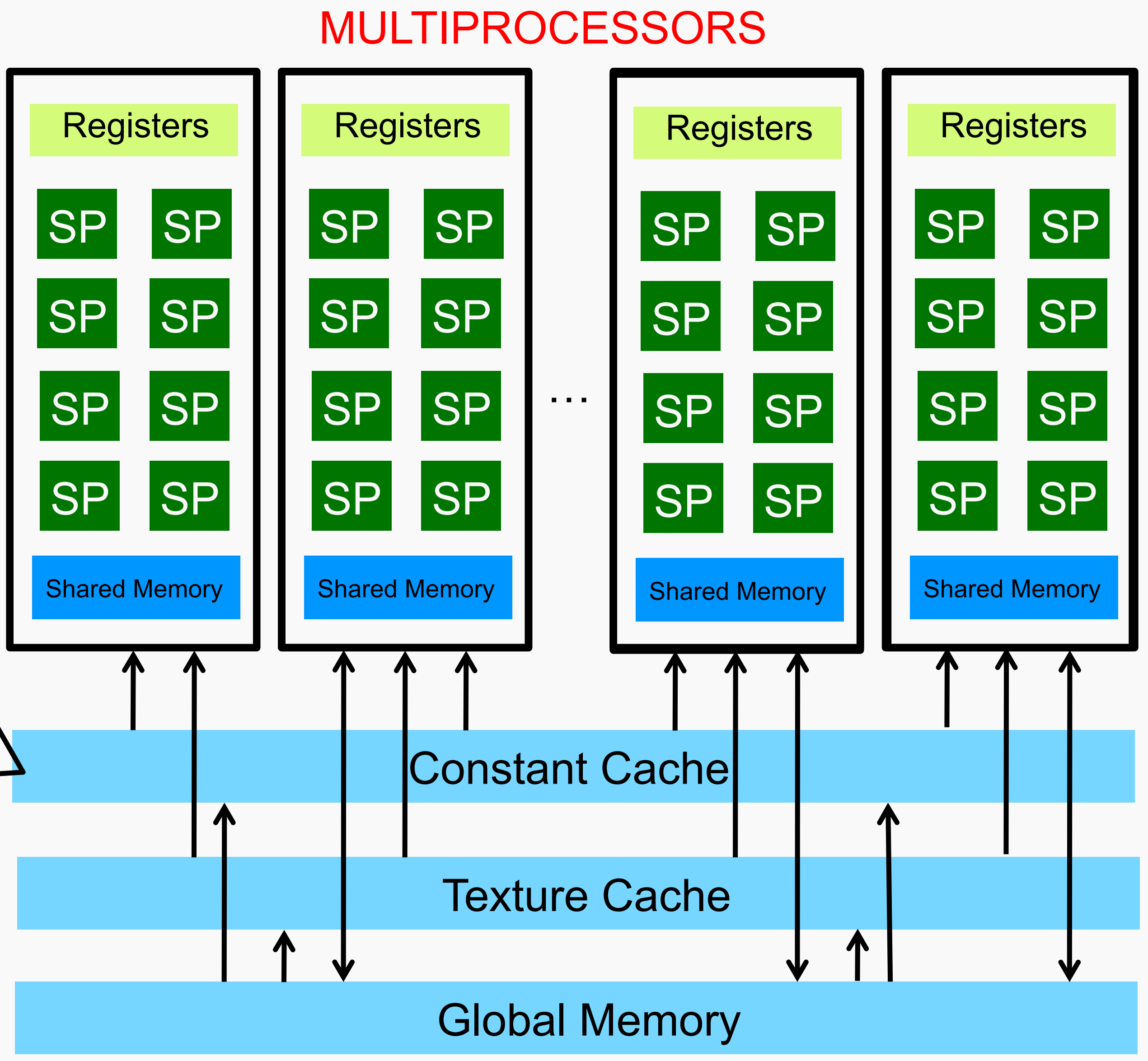


Memory Hierarchy



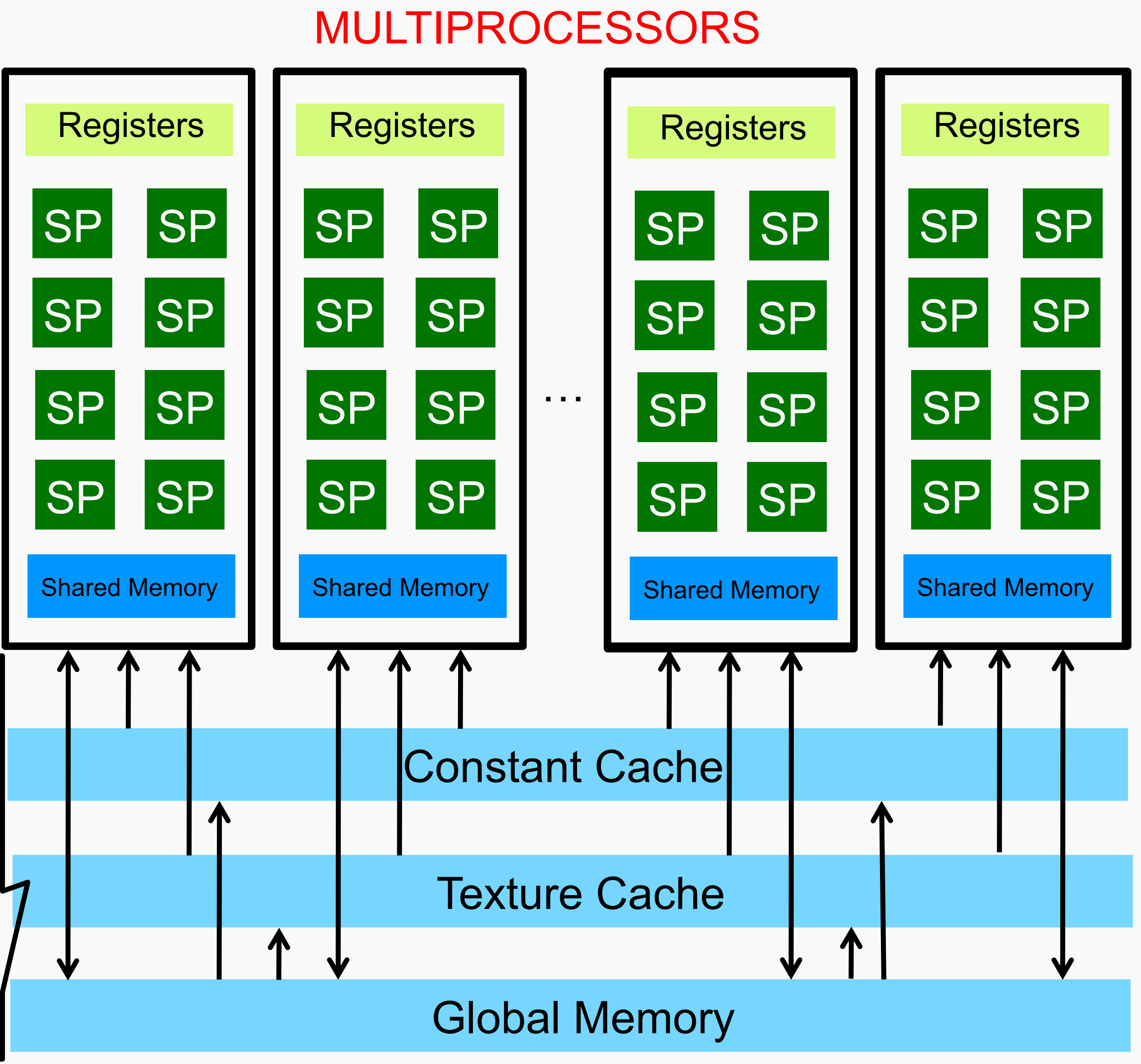
GPUs have also DRAM
The latency is 150x is slower then registers and shared memory

Memory Hierarchy



Constant memory, as the name implies, is a read-only region which also has a small cache.

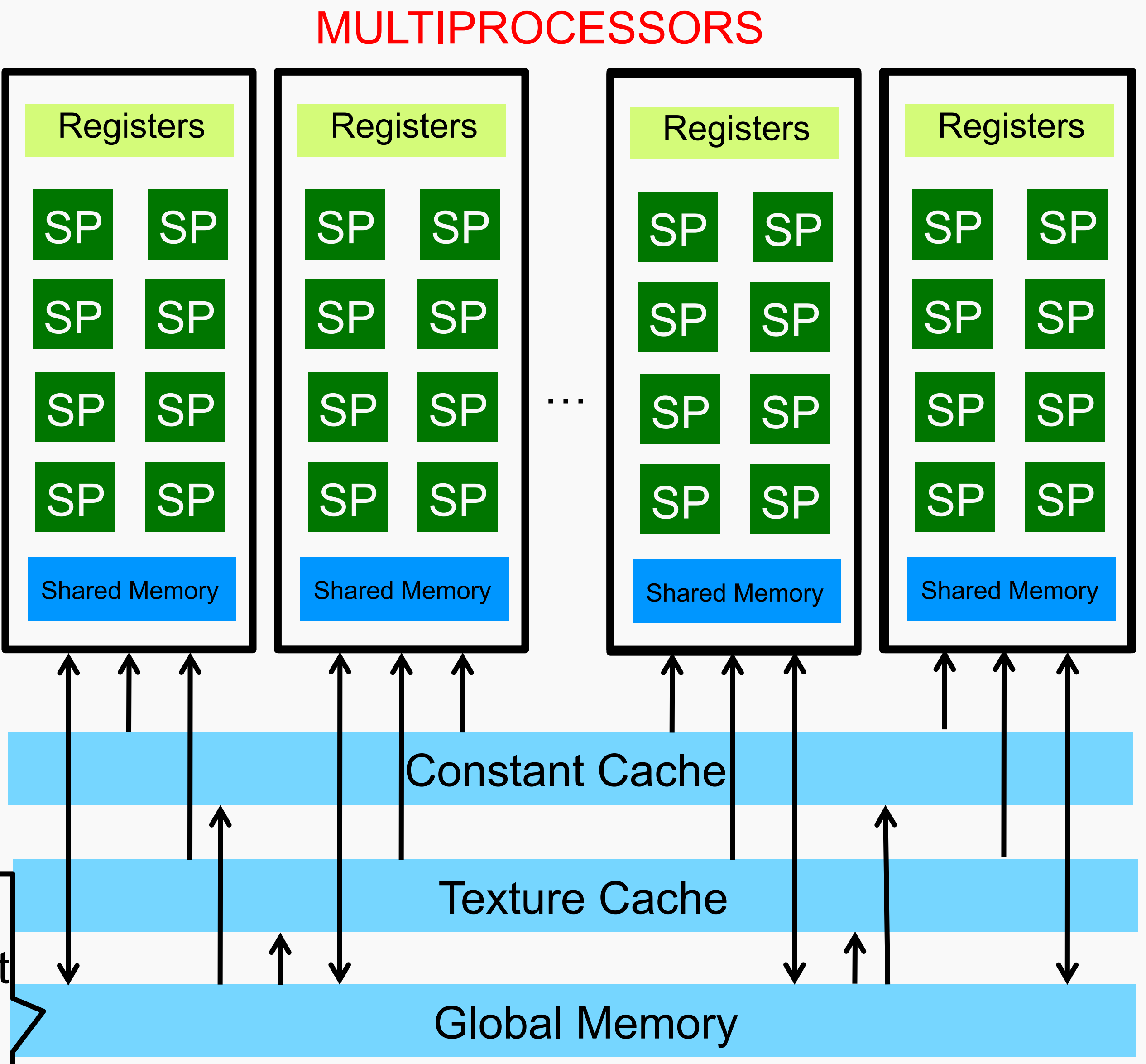
Memory Hierarchy



Texture memory is read-only with a small cache optimized for manipulation of textures. It also provides built-in linear interpolation of the data.

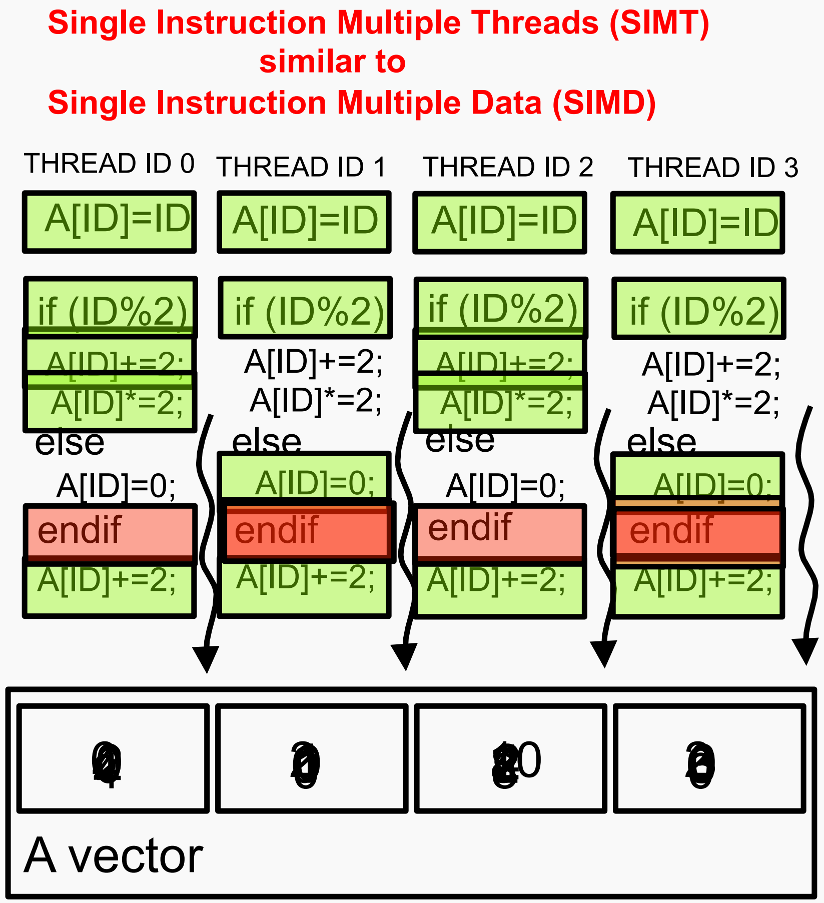
also provides built-in linear interpolation of the data.

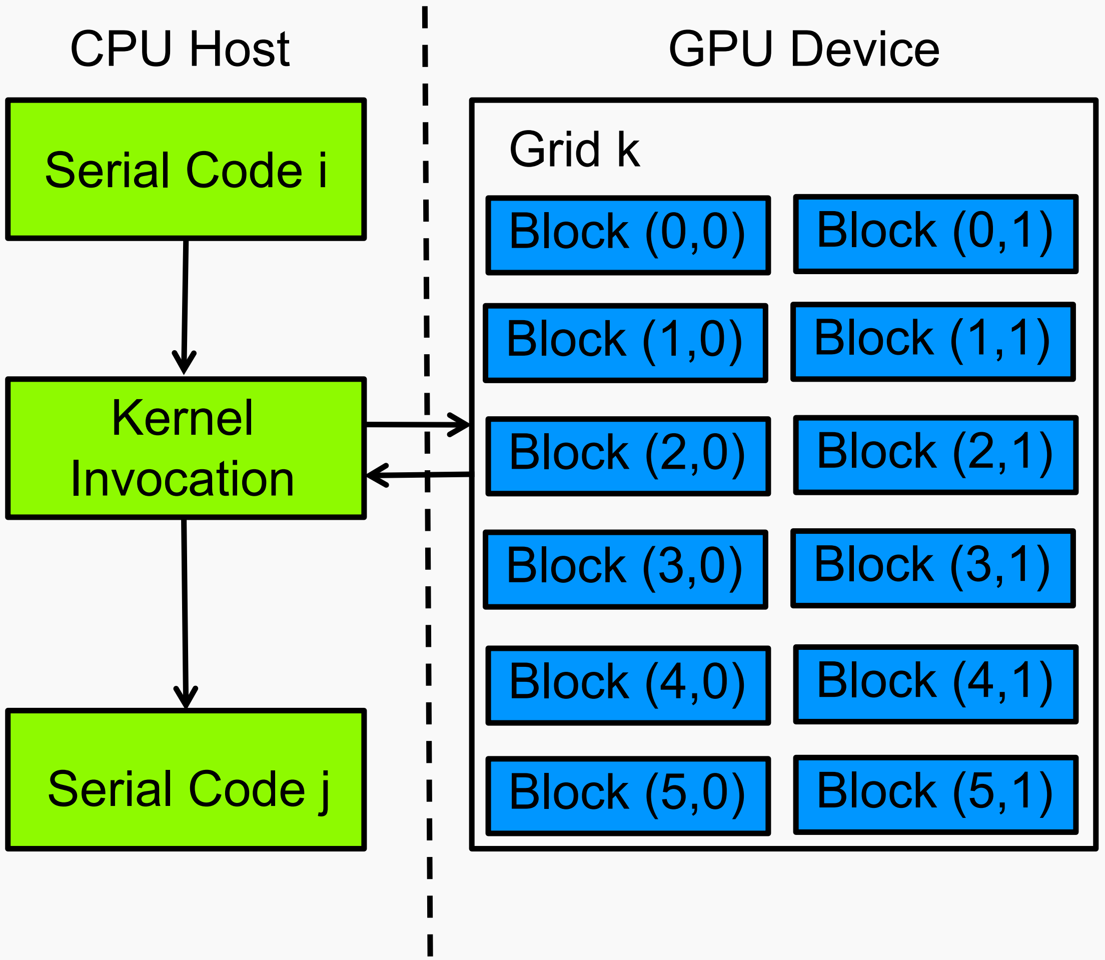
Memory Hierarchy



Global memory is available to all threads and is persistent between GPU calls.

CUDA Programming Model





When branches occur in the code (e.g. due to if statements) the divergent threads will become inactive until the conforming threads complete their separate execution.

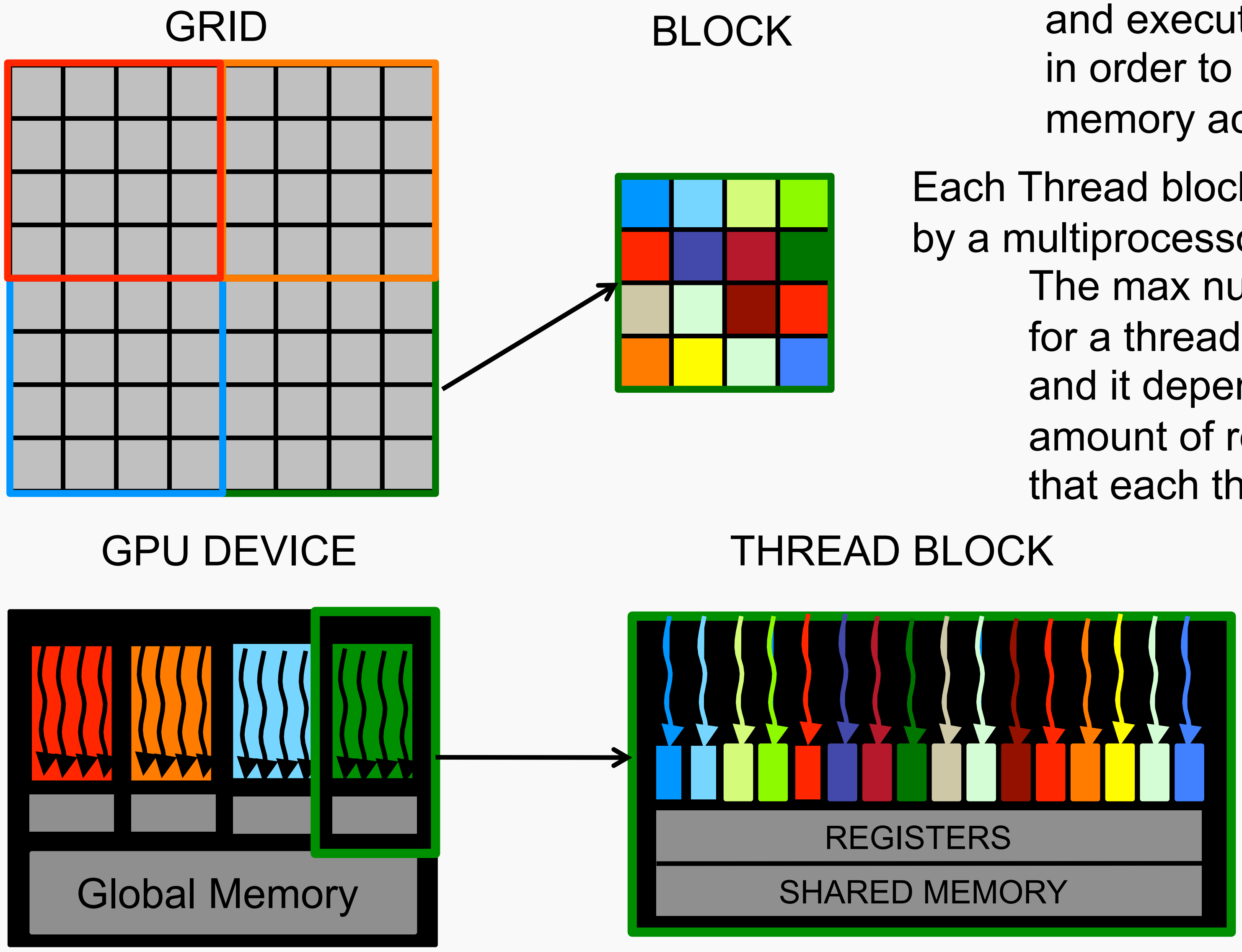
When execution merges, the threads can continue to operate in parallel.

CUDA Programming Model

Different threads are multiplexed and executed by the same core in order to reduce the latency of memory access.

Each Thread block is executed by a multiprocessors

The max number of threads for a thread block is 512 and it depends on the amount of registers that each thread may need.



Tesla C1060   Fermi C2070

30 Multiprocessors
240 Cores
Processor core clock: 1.296 GHz
933 Gigaflops (Single precision)
78 Gigaflops (Double Precision)
Max Bandwidth(102 Gigabytes/sec)
4 GB of DRAM

Cost: **1000 $**

14 Multiprocessors
448 Cores
Processor core clock: 1.15 GHz
1030 Gigaflops (Single precision)
515 Gigaflops (Double precision)
Max Bandwidth (144 GBytes/sec)
6 GB of DRAM

Cost: **3200 $**

Reaction-Diffusion in CUDA

$$\frac{\partial \mathbf{u}}{\partial t} = R(\mathbf{u}) + \nabla(D\nabla \mathbf{u})$$

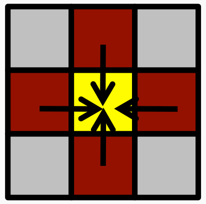
For each time step, a set of (ODEs) and Partial Differential Equations (PDEs) must be solved.

```
for (timestep=1; timestep < nend; i++){
        solveODEs <<grid, block>> (….);
        calcLaplacian <<grid, block>> (….);
}
```

Solving ODEs using different method depending on the model:

- -Runge-Kutta 4th order
- -Forward Euclid
- -Backward Euclid
- -Semi-Implicit
- ……

Solving PDEs (Calc the Laplacian)



$$\nabla(D\nabla \mathbf{u})_{i,j} = \frac{Ddt}{dx^2}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j}\right)$$

Optimize the Reaction Term in CUDA

Heaviside Simplification

- In order to avoid divergent branches among threads we substitute if then else condition of Heaviside functions in this way:

```
If (x > a){
   y = b;
}else{
   y = c;
}
```

```
y =b + (x > a)*(b_c)
```

a, b, b_c (b-c) are constant

Precomputing Lookup Tables using Texture

- We build tables where we precompute nonlinear part of the ODEs, we bind these table to textures and we exploit the built-in linear interpolation feature of the texture.

Optimize the Reaction Term in CUDA

Kernel splitting:

- For complex models, we need to split the ODEs solving in many Kernels in order to have enough registers for thread to perform our calculation.

Use –use_fast_math compiler option to substitute

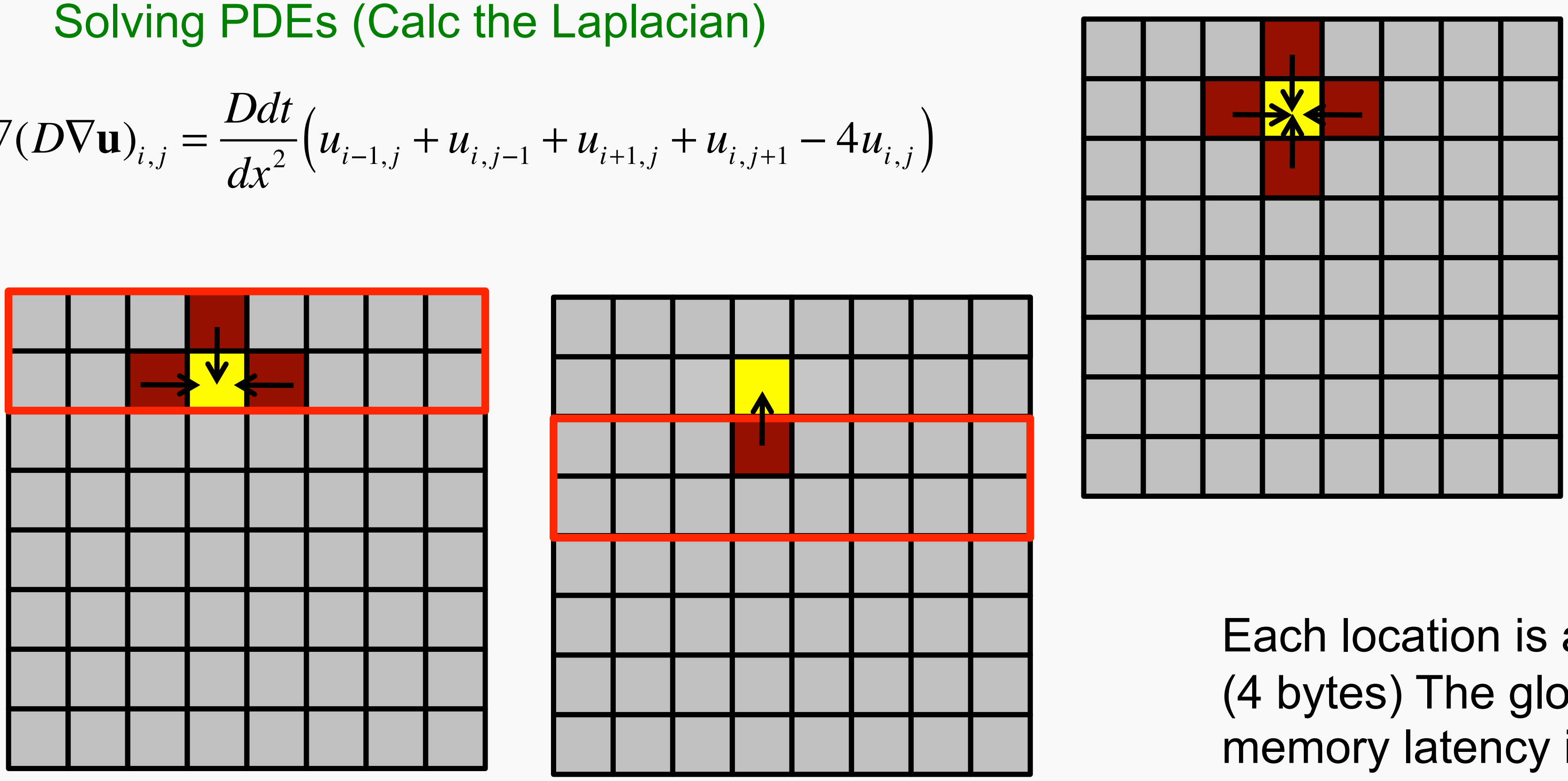
- In the models that use log, exp, sqrt, functions we substitute them with the GPU built-in functions.

Using Costant memory for common parameters (dt, dx, ..)

Optimize the Diffusion Term in CUDA

Solving PDEs (Calc the Laplacian)

$$\nabla(D\nabla\mathbf{u})_{i,j} = \frac{Ddt}{dx^2}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j}\right)$$



Using texture we can reduce the latency
In texture the data is cached (optimize for 2D Locality)
Drawback: It supports only single precision

Each location is a float (4 bytes) The global memory latency is very slow. The memory is accessed in multiples of 64 bytes

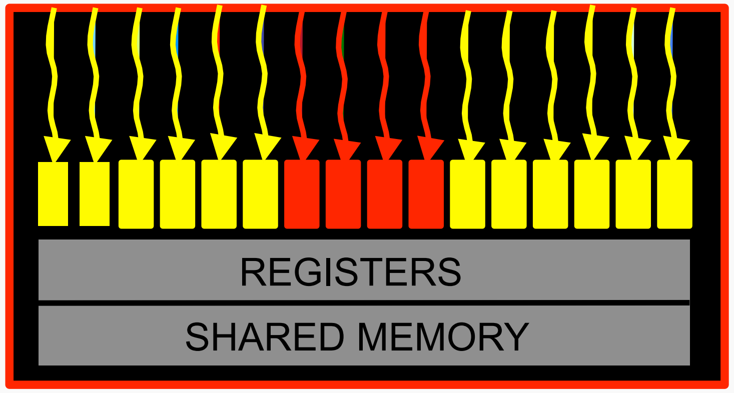
Optimize the Diffusion Term in CUDA

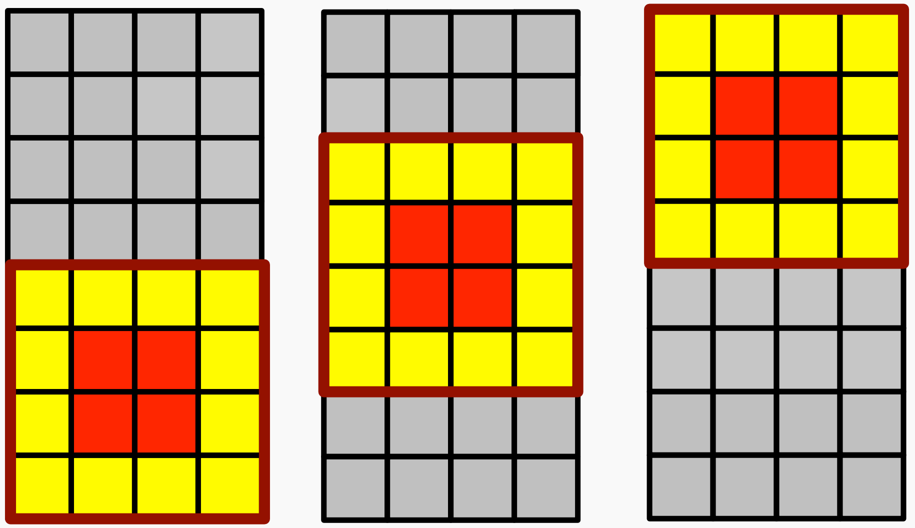
Another Technique is using SHARED MEMORY

$$\begin{tikzcd}\boxed{\text{The yellow and red threads read the location from the global memory into the shared memory.}} \arrow[r] & \boxed{\text{SYNCH}} \arrow[r] & \boxed{\text{The red threads calculates the laplacian using the values in the shared memory.}}\end{tikzcd}$$

$$\nabla(D\nabla\mathbf{u})_{i,j} = \frac{Ddt}{dx^2}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j}\right)$$

THREAD BLOCK

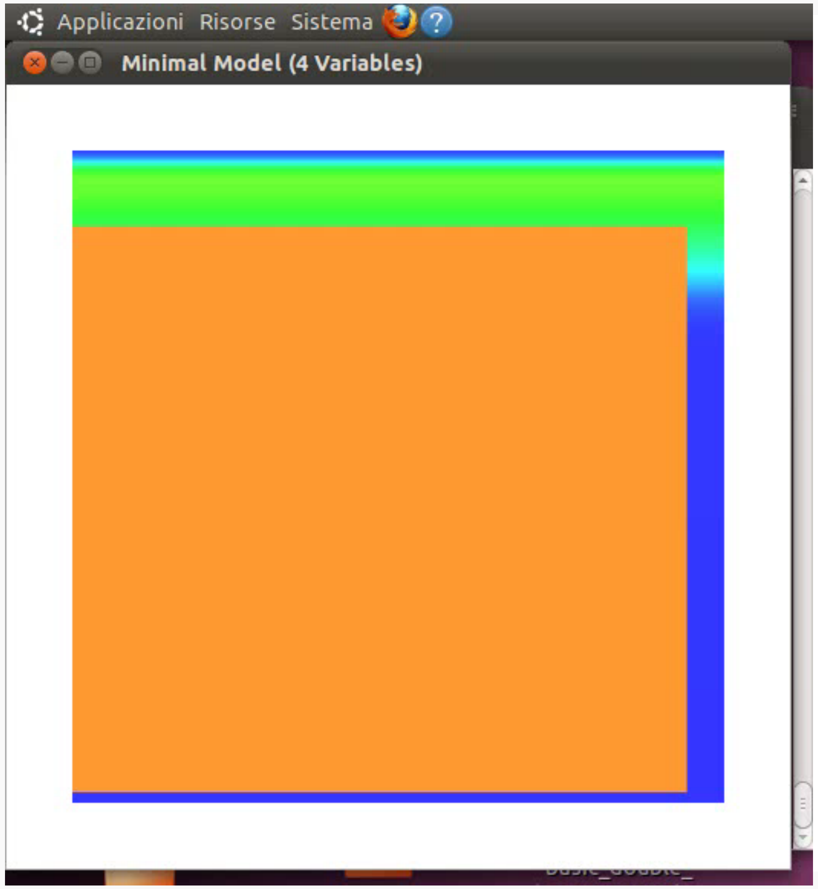


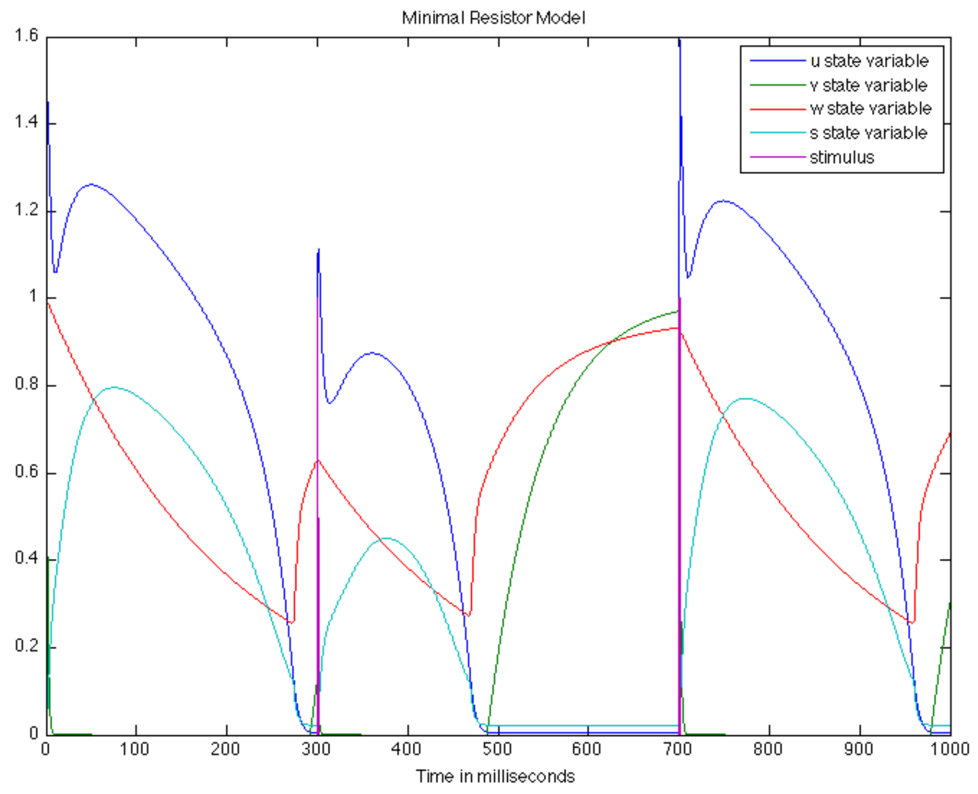


This technique supports single and double precision

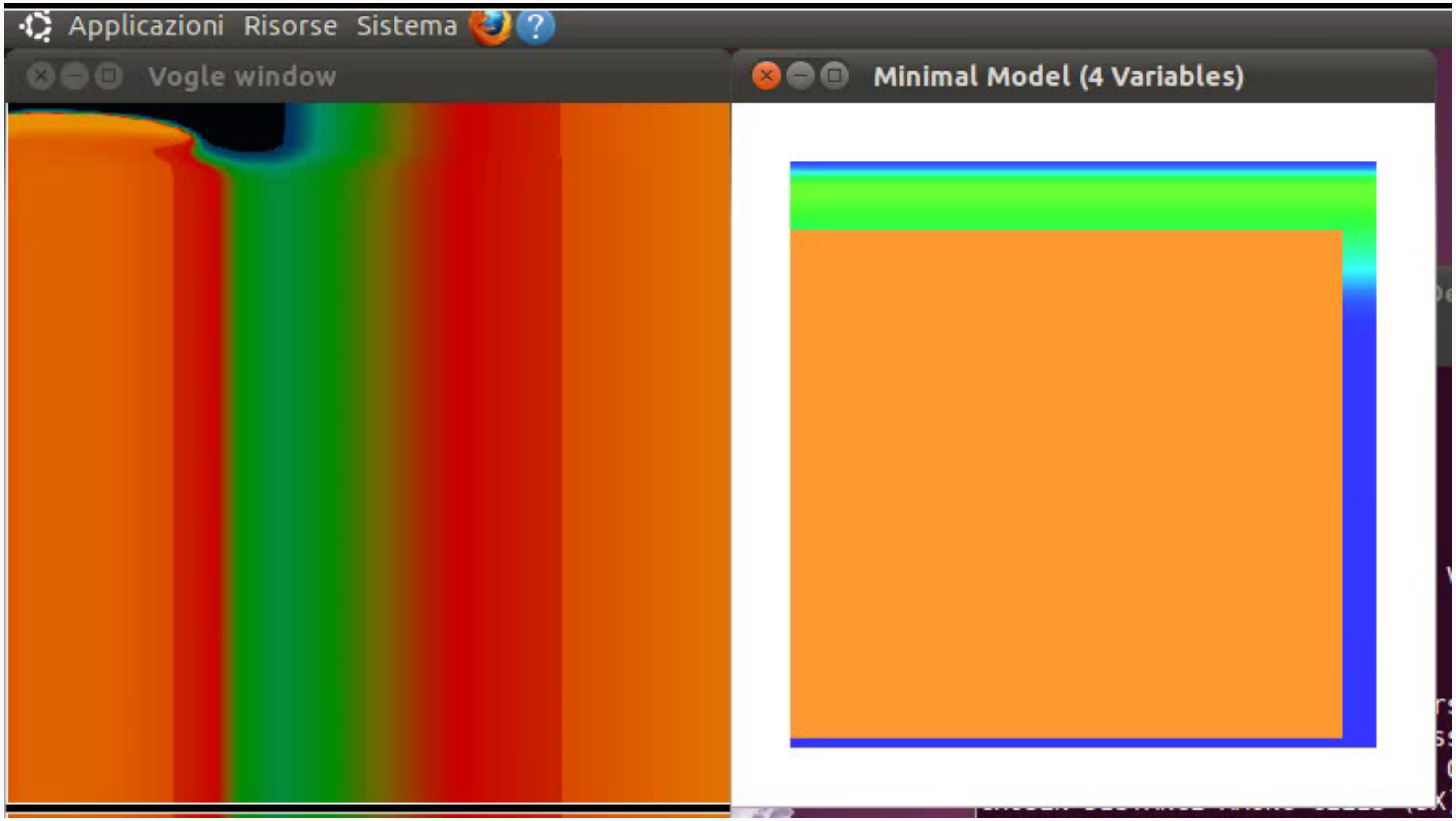
Drawback: The number of threads is greater than the number of elements

Case Study 1: Minimal Model (4V)



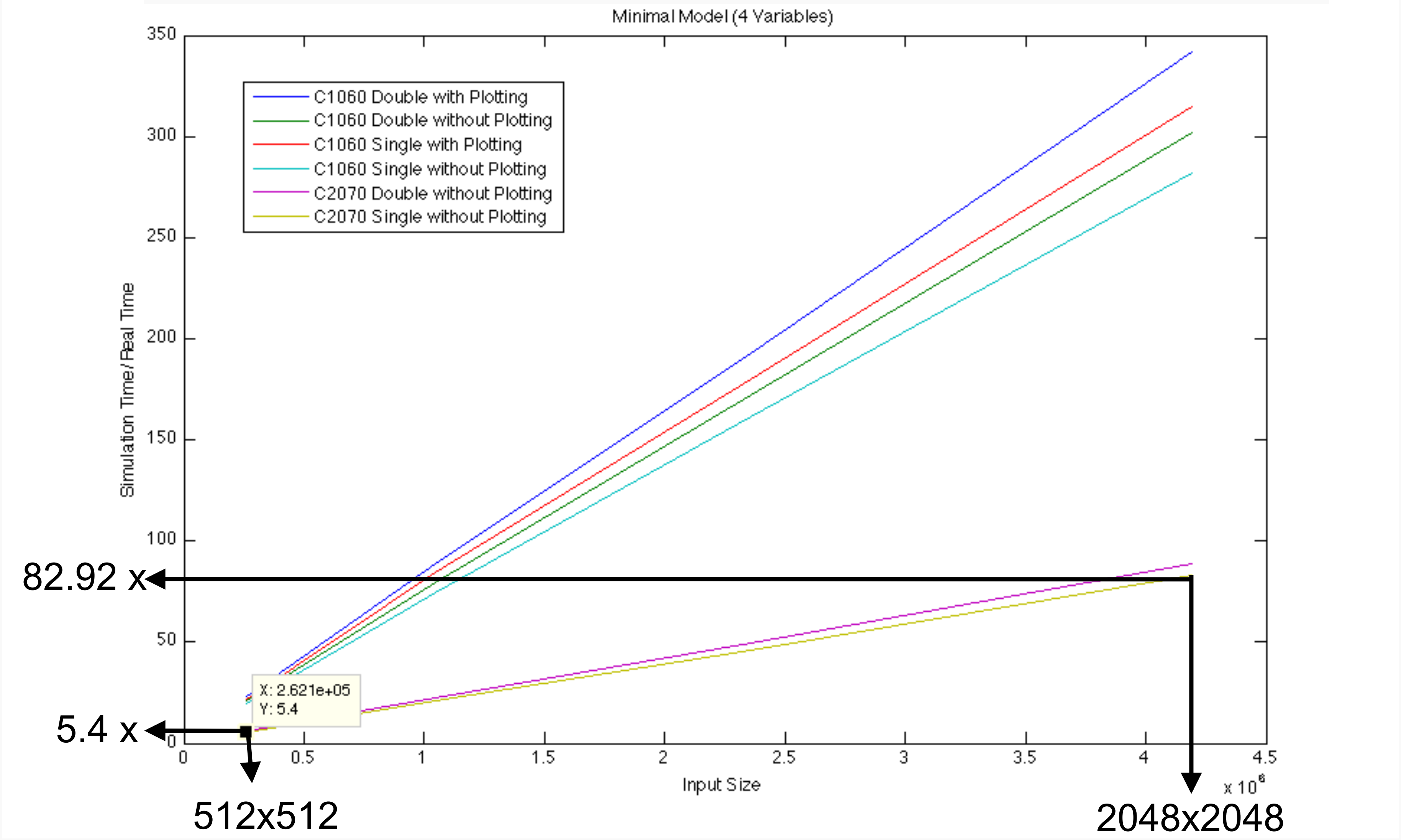


Perfomances

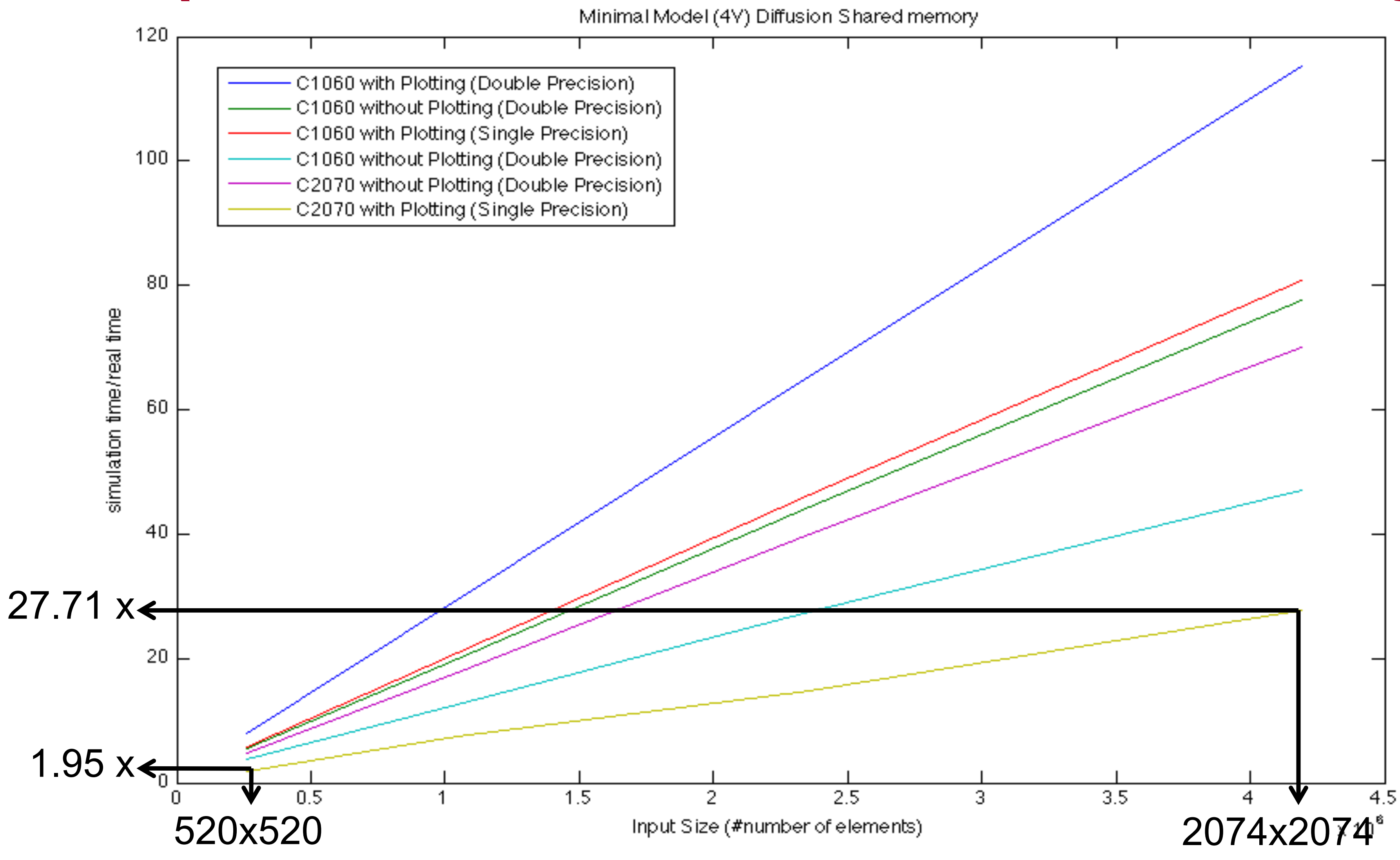


CPU Computation GPU Computation

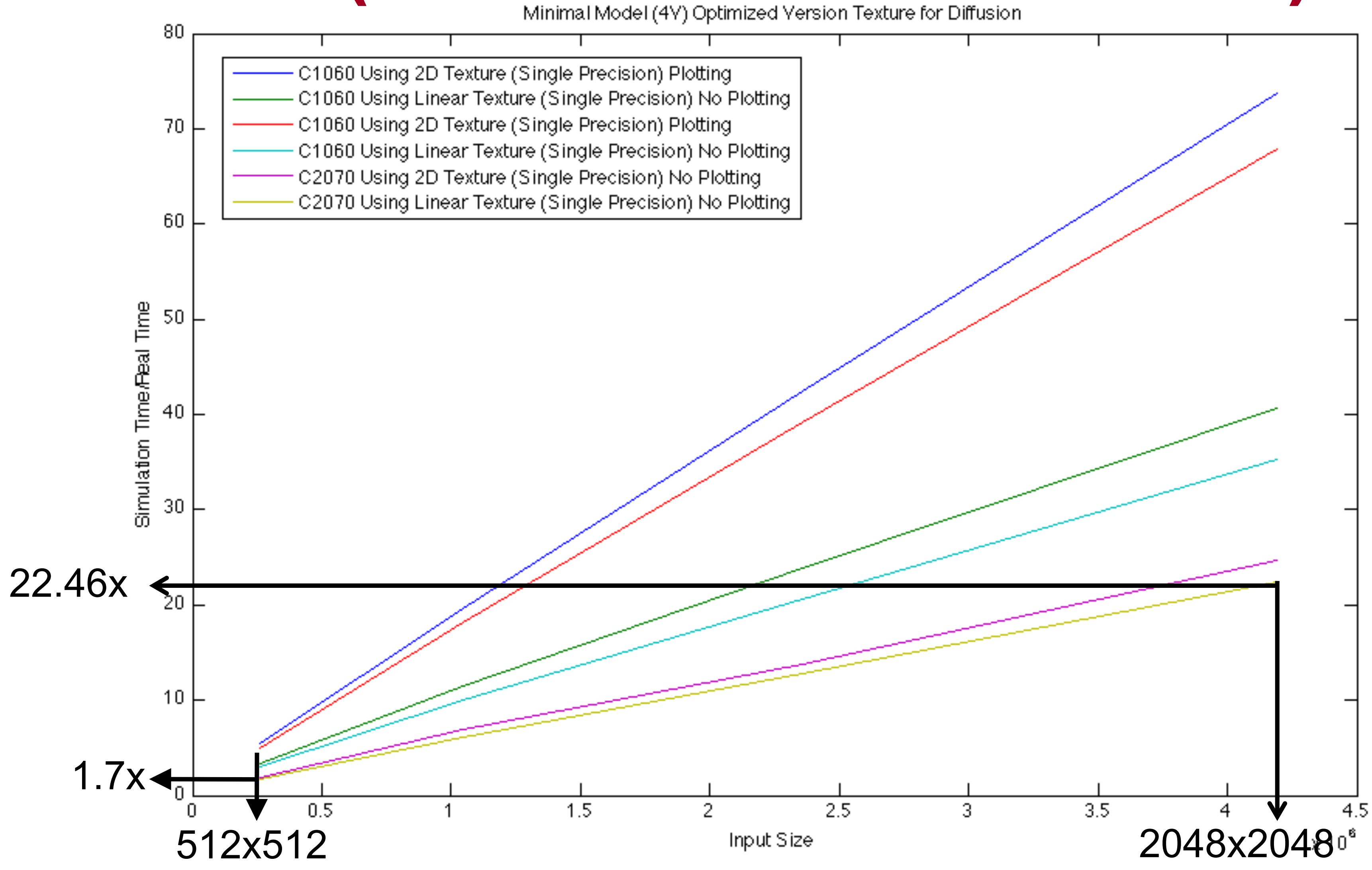
Naïve Implementation



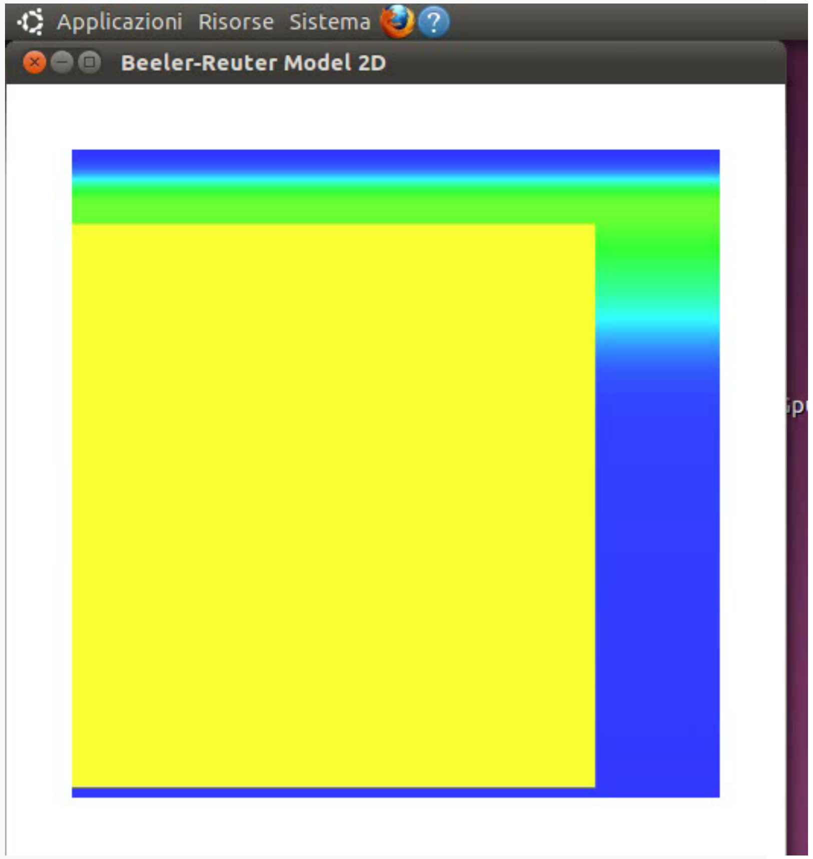
Reaction optimized (Diffusion with Shared Memory)

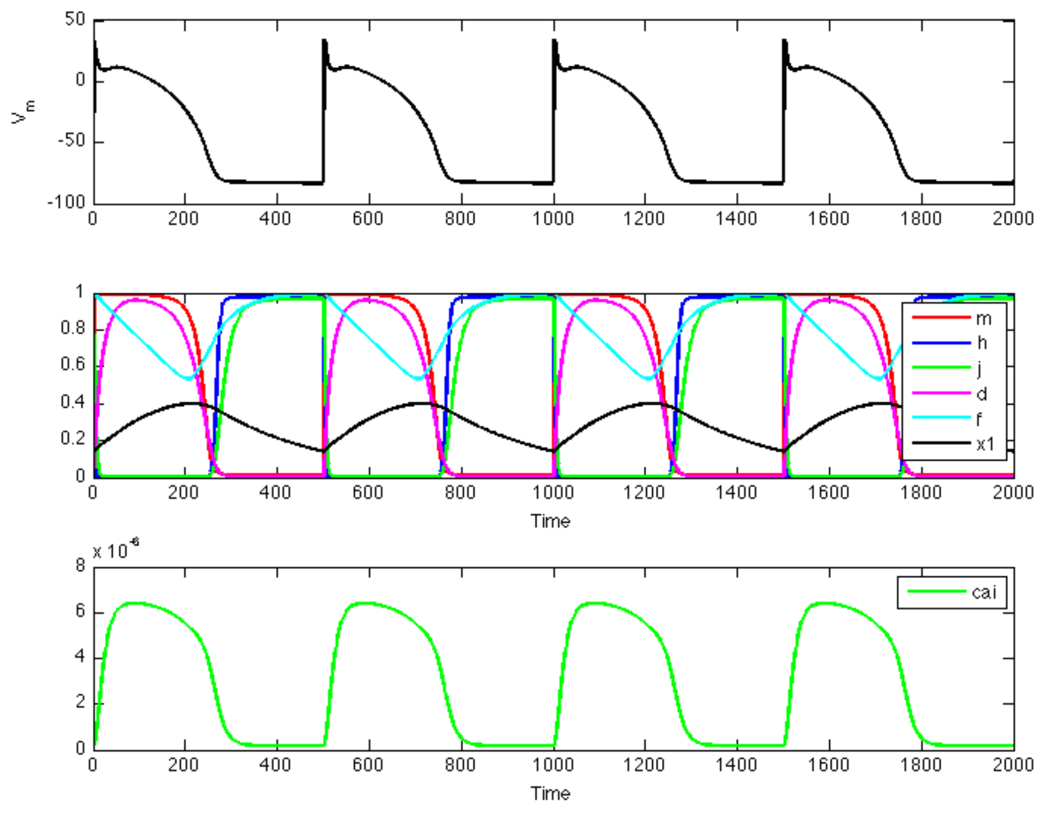


Reaction optimized (Diffusion with Texture)

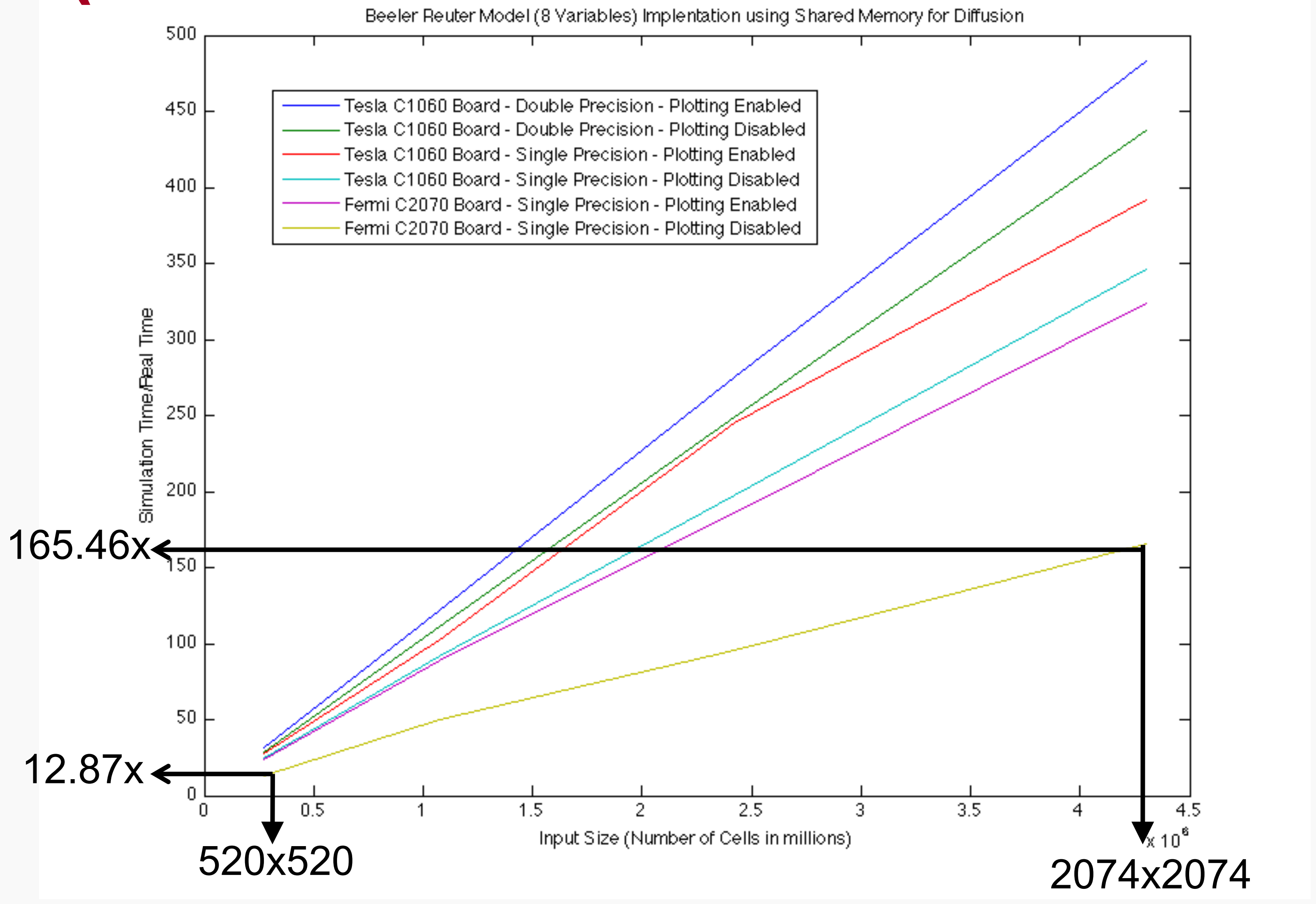


Beeler-Reuter Model (8V)

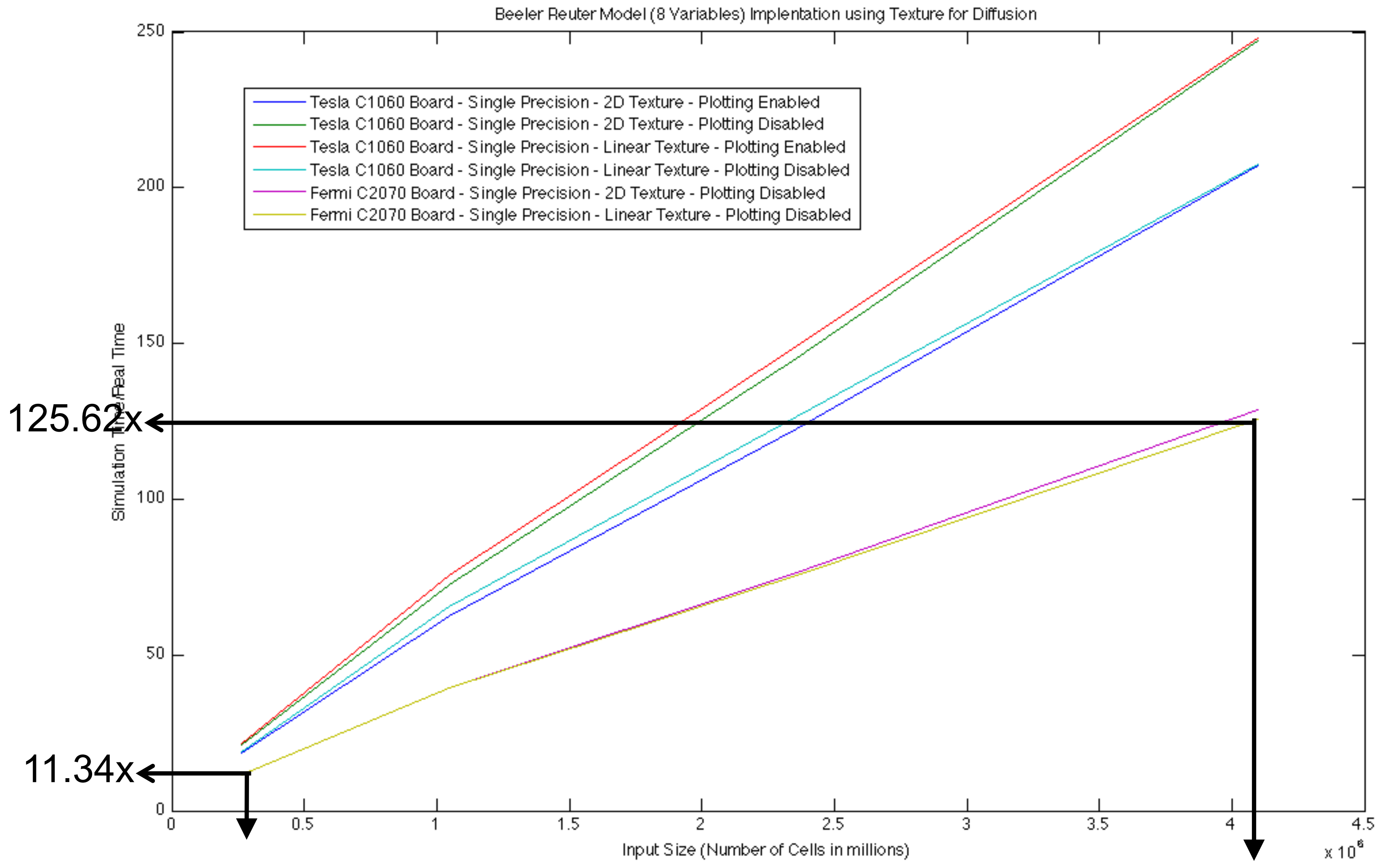




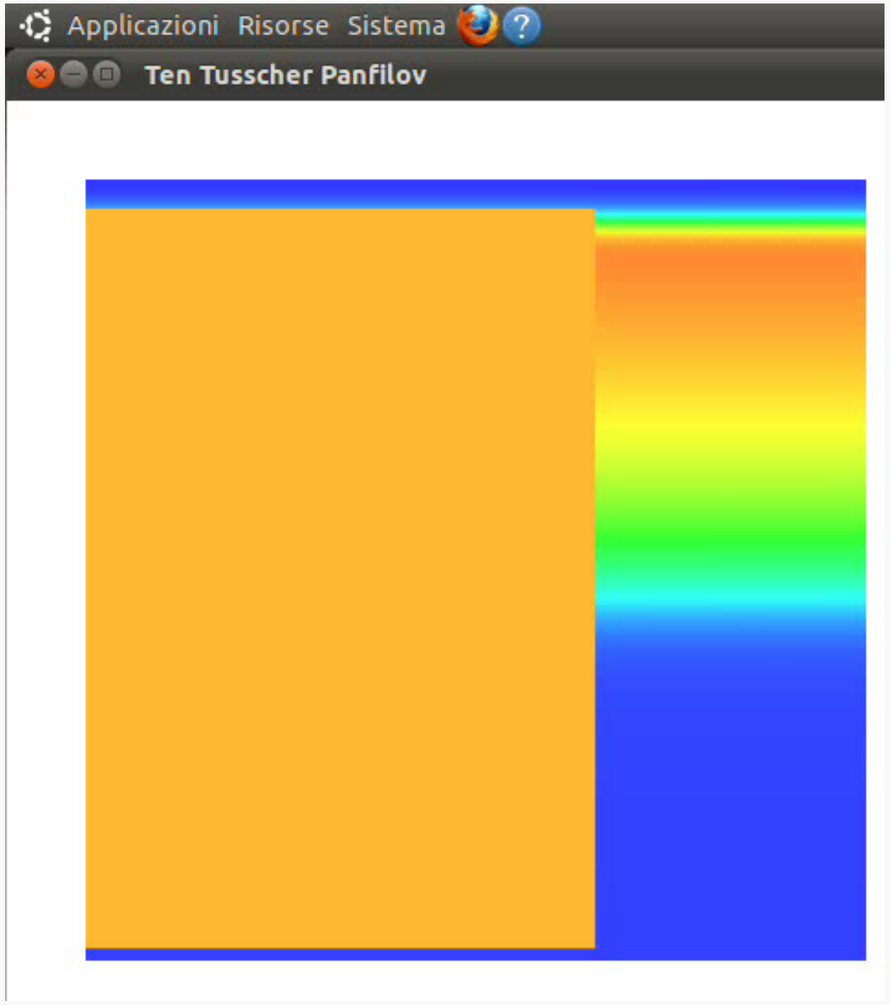
Reaction optimized (Diffusion with Shared Memory)



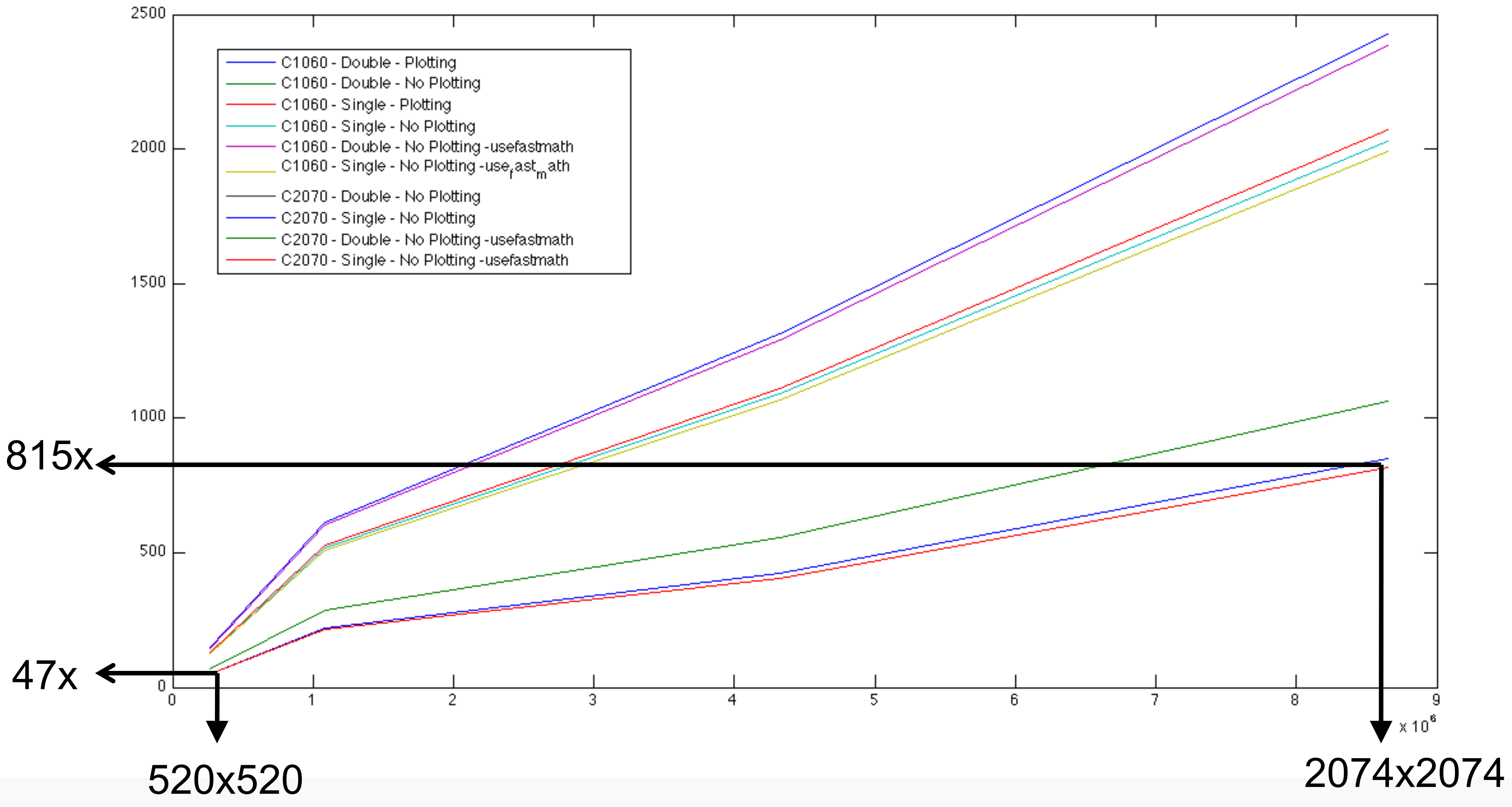
Reaction optimized (Diffusion with Texture)



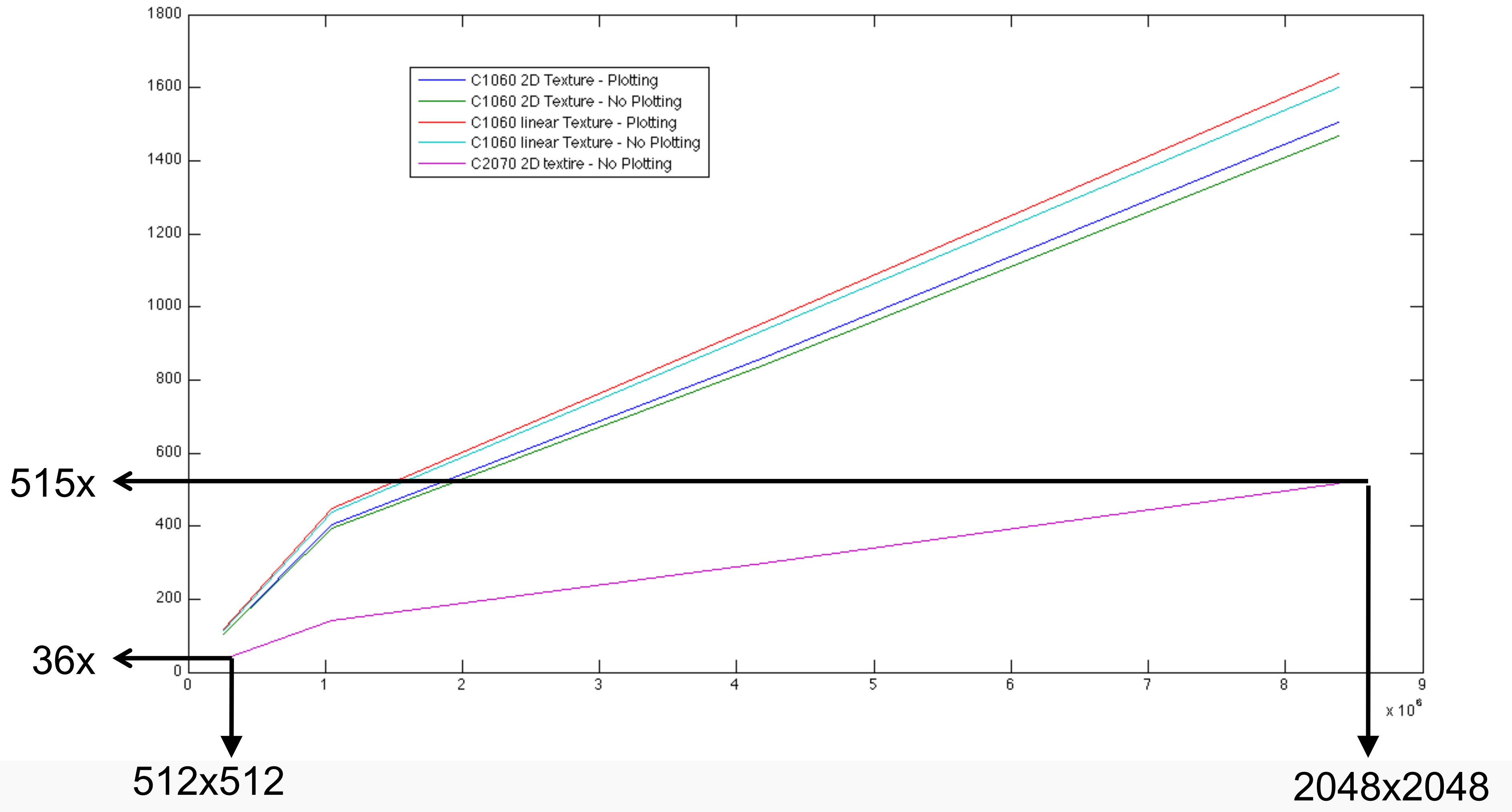
Ten Tusscher Panfilov Model (19V)



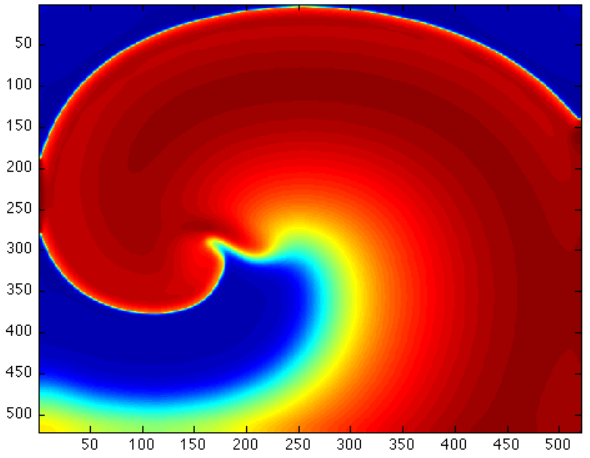
Reaction optimized (Diffusion with Shared Memory)

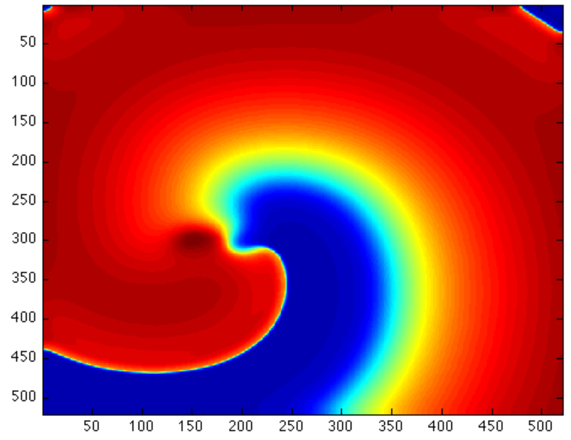


Reaction optimized (Diffusion with Texture)

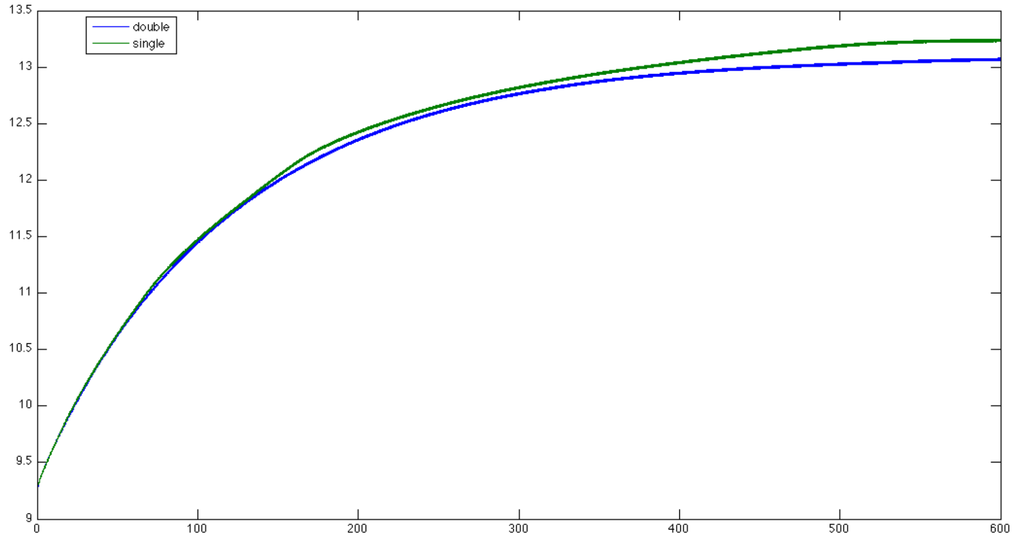


Double vs Single



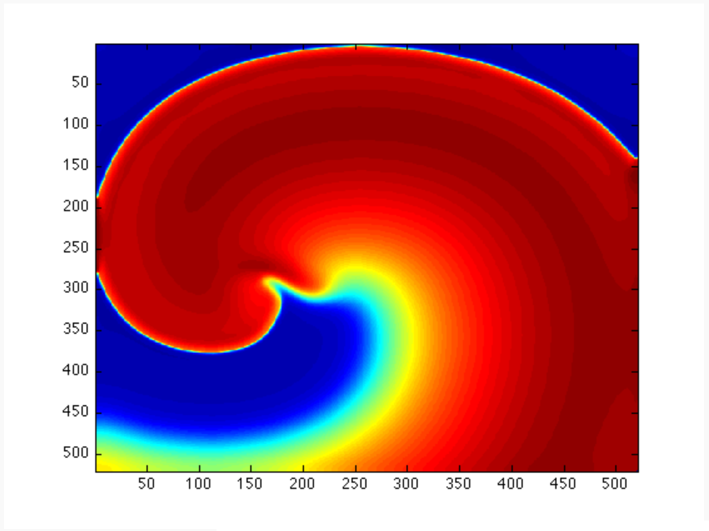


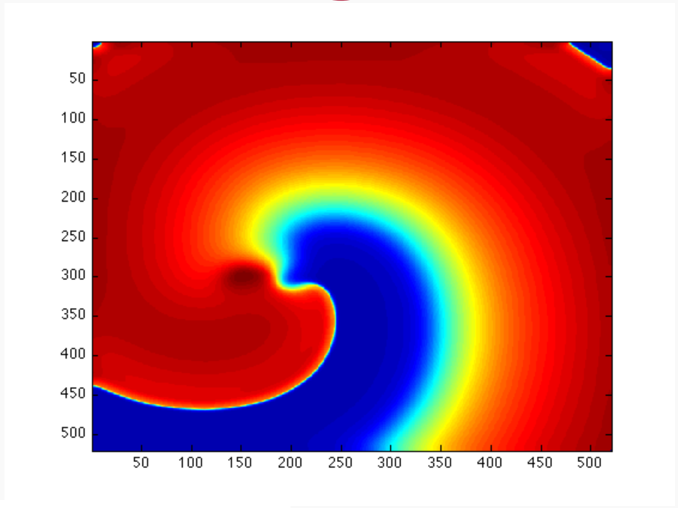
Nai



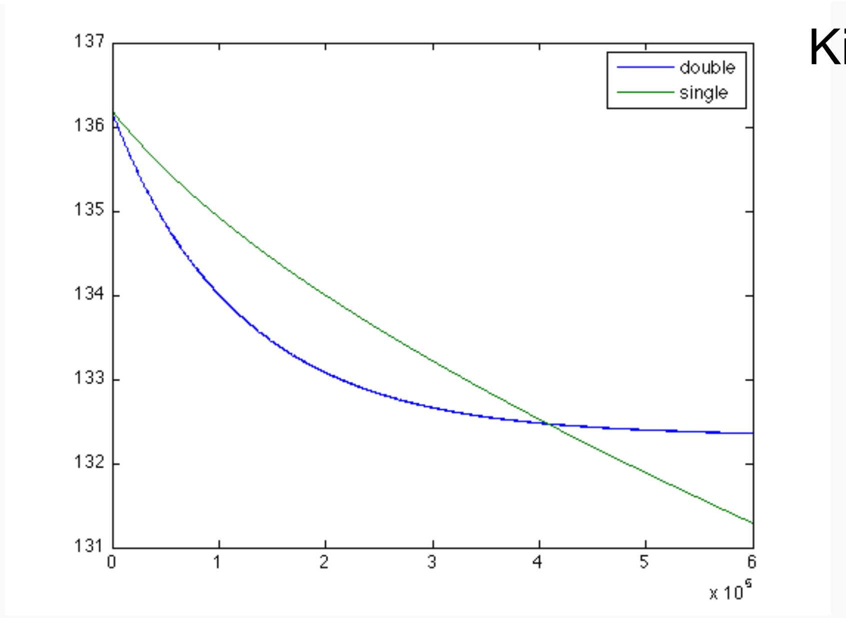
After 10 minutes of simulation:

Double vs Single





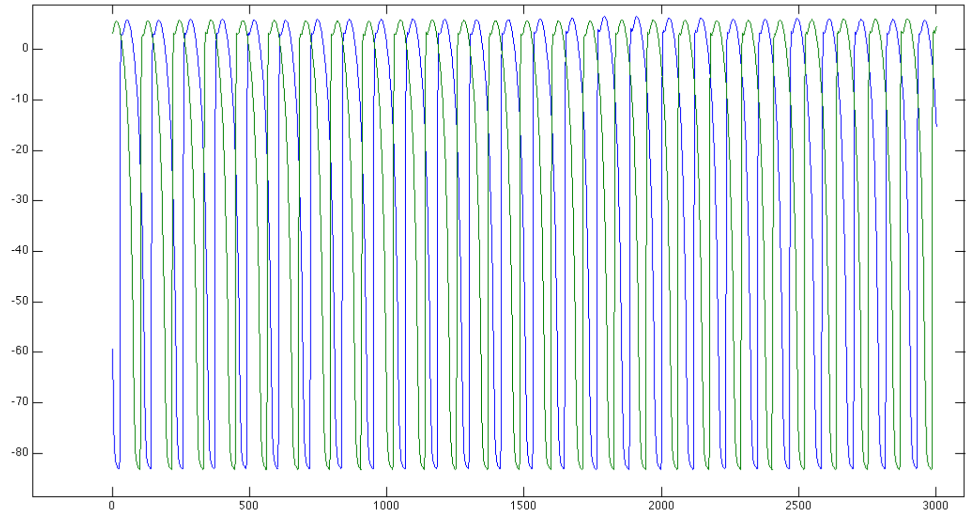
Ki

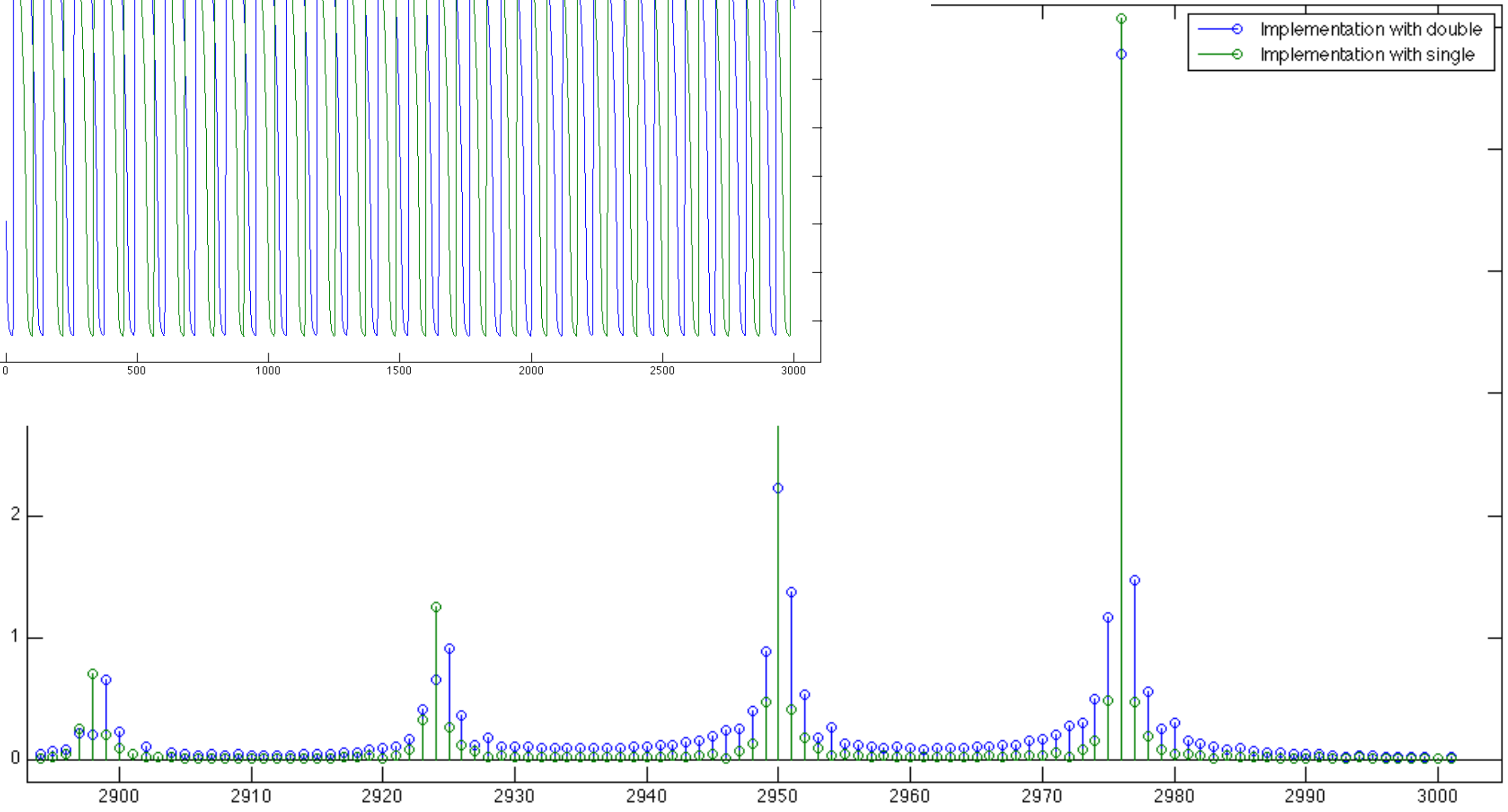


After 10 minutes of simulation:

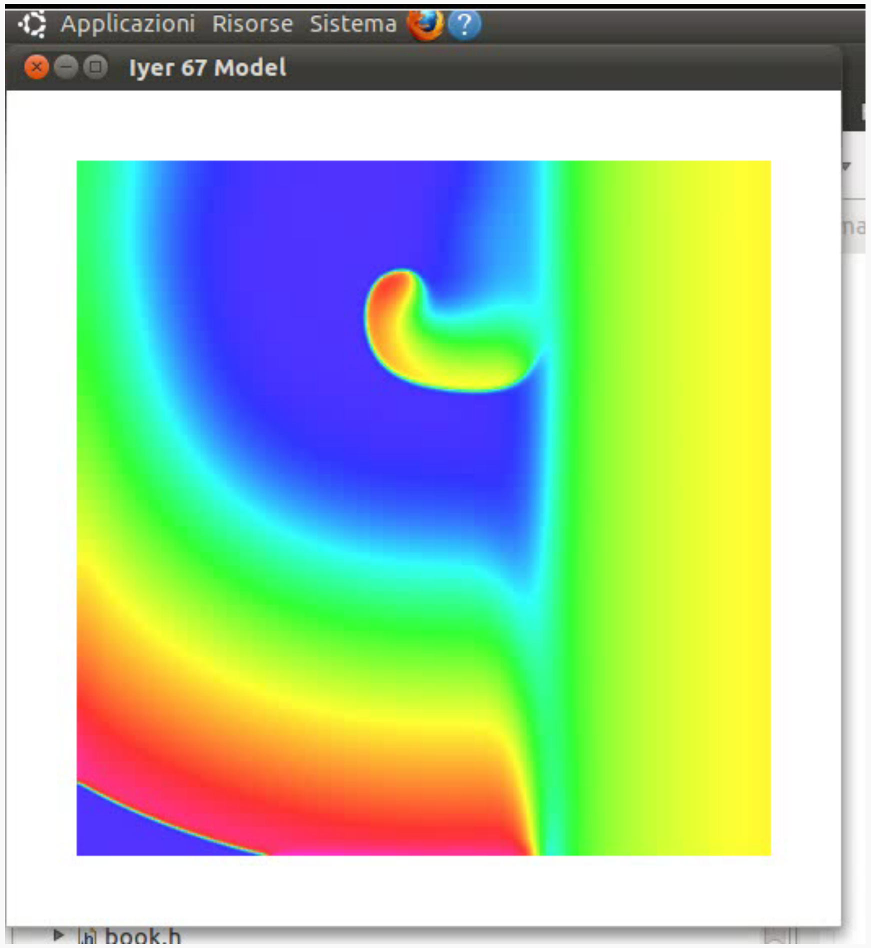
Double vs Single

After 10 minutes of simulation:

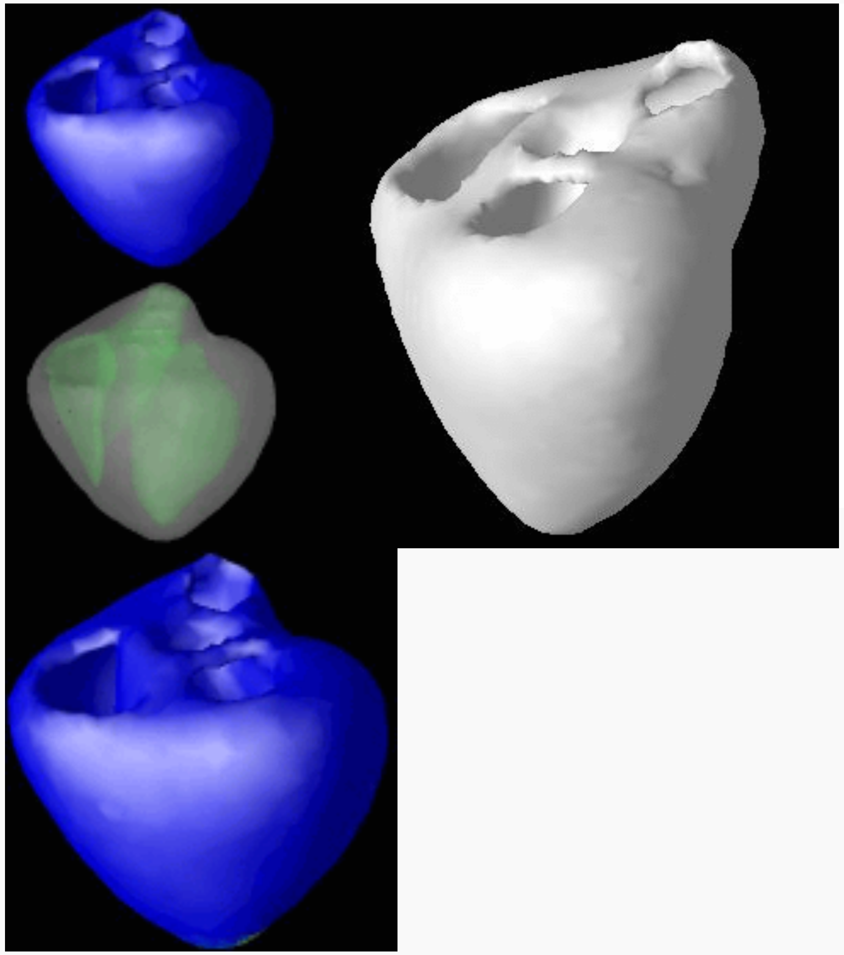


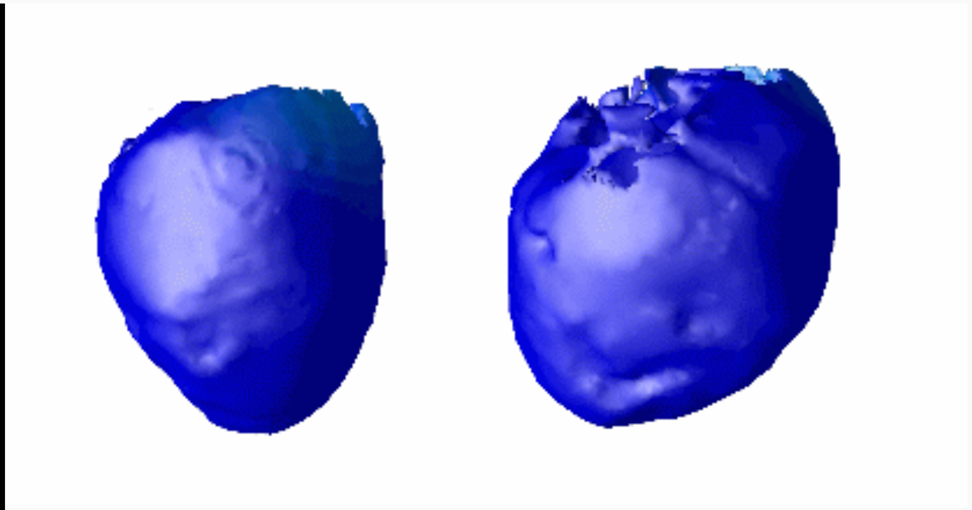


Work in Progress Iyer Model (67V)



Work in Progress 3D Models





Conclusions

- Many other challenge problems of CMACS expedition can take advantage of GPU technologies.
- The curve of developing of these technologies seems very promising for the future years.
- We are definitely interesting to collaborate with the other teams of the CMACS expedition in order to develop new revolutionary highly scalable GPU-based analysis tools for complex systems.

Thank you