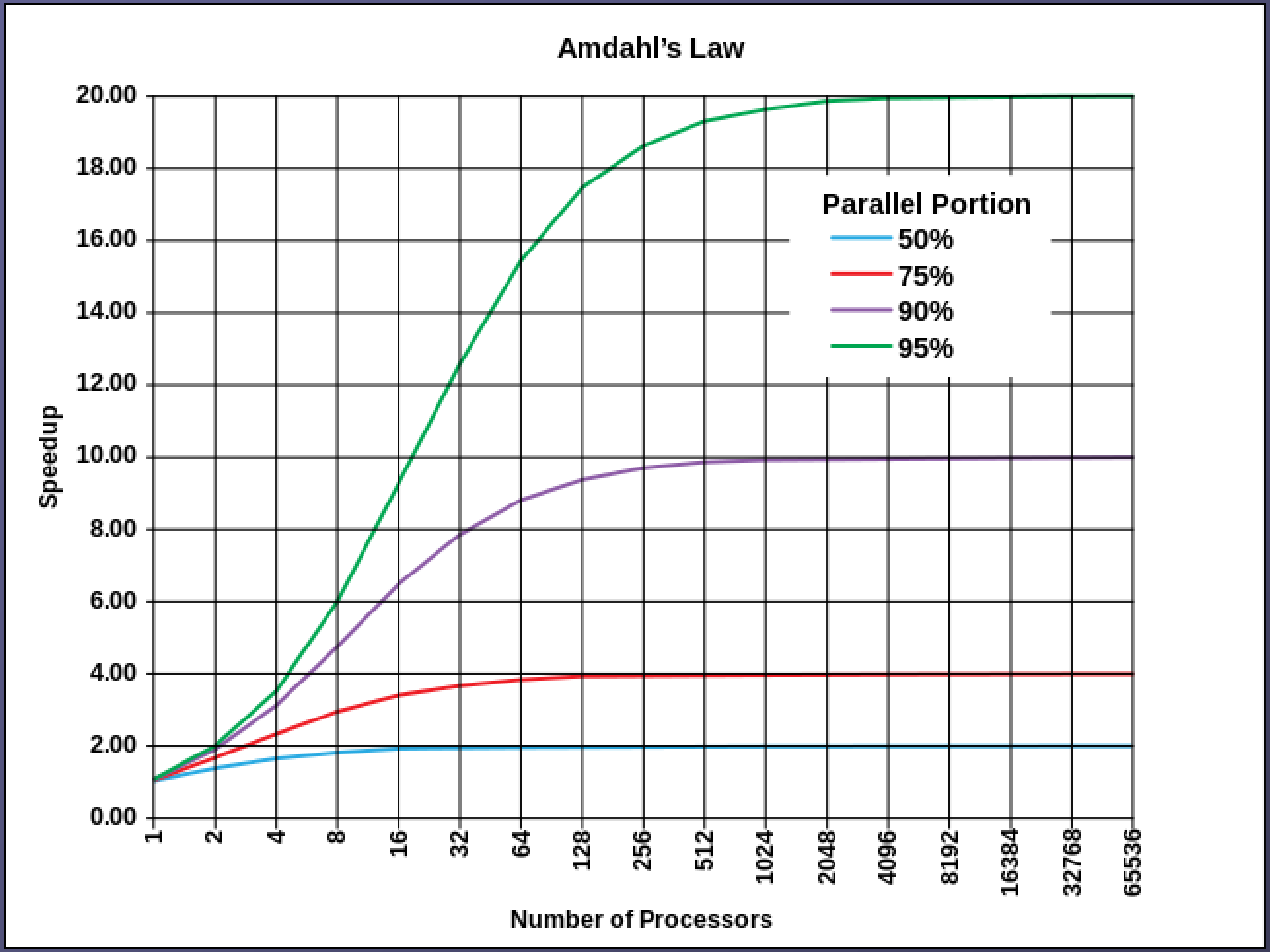
Parallel Computing

Some basic ideas

Amdahl's Law (Gene Amdahl 1967)



Evolution according to Amdahl's law of the theoretical speedup of the execution of a program in function of the number of processors executing it, for different values of p. The speedup is limited by the serial part of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

Calculate Amdahl's Law:

Let X be the part of my program (in terms of computing time) which can be parallelised.
The sequential computing time T_{seq} is normalized to unity (1), and can be expressed as:

$$T_{seq} = 1 = X + (1-X)$$

The parallel computing time Tpar under ideal conditions (ideal load balancing, ultrafast communication):

$$T_{par} = X/p + (1-X)$$

with processor number (core number) p ;
Then the speed-up of the program $S = T_{seq} / T_{par}$:

$$S = 1 / (1-X+X/p) \quad ;$$

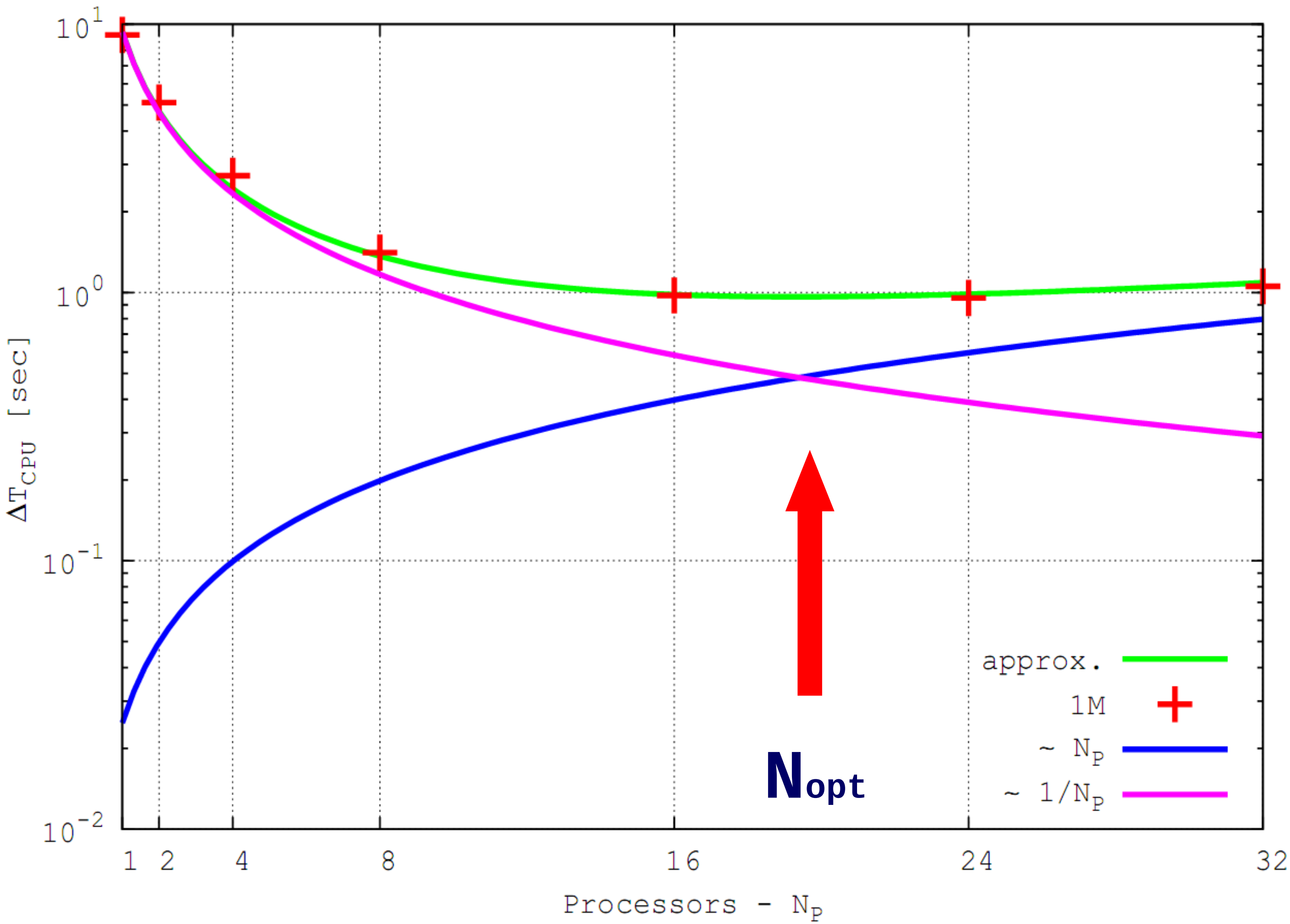
$$\text{Note: } T_{par}/T_{seq} = 1/S \text{ (sometimes also plotted)}$$

Note the limit of S for large p is very large: S = 1/(1-X). And if X ~ 1: S ~ p
With communication overhead:

$$T_{par} = X/p + (1-X) + T_{comm} \quad \rightarrow \quad S = 1 / (1-X+X/p+T_{comm})$$

If T_{comm} independent of p we have for large p: $S = 1 / (1-X + T_{comm}) = \text{const.}$

Parallel code on cluster



Strong and Soft Scaling

- Strong Scaling: Fixed Problem size, increase p
- Soft Scaling: Increase Problem size, increase p (constant amount of work per processing element)

Ansatz for Soft Scaling:

- $T_{seq} = p\,(X + (1-X))$
- $T_{par} = X + p\,(1-X)$
- $S = T_{seq}/T_{par} = p \;/\; (X+p\,(1-X))$

If $X \sim 1$: $S = p$; $T_{par} = X = const.$

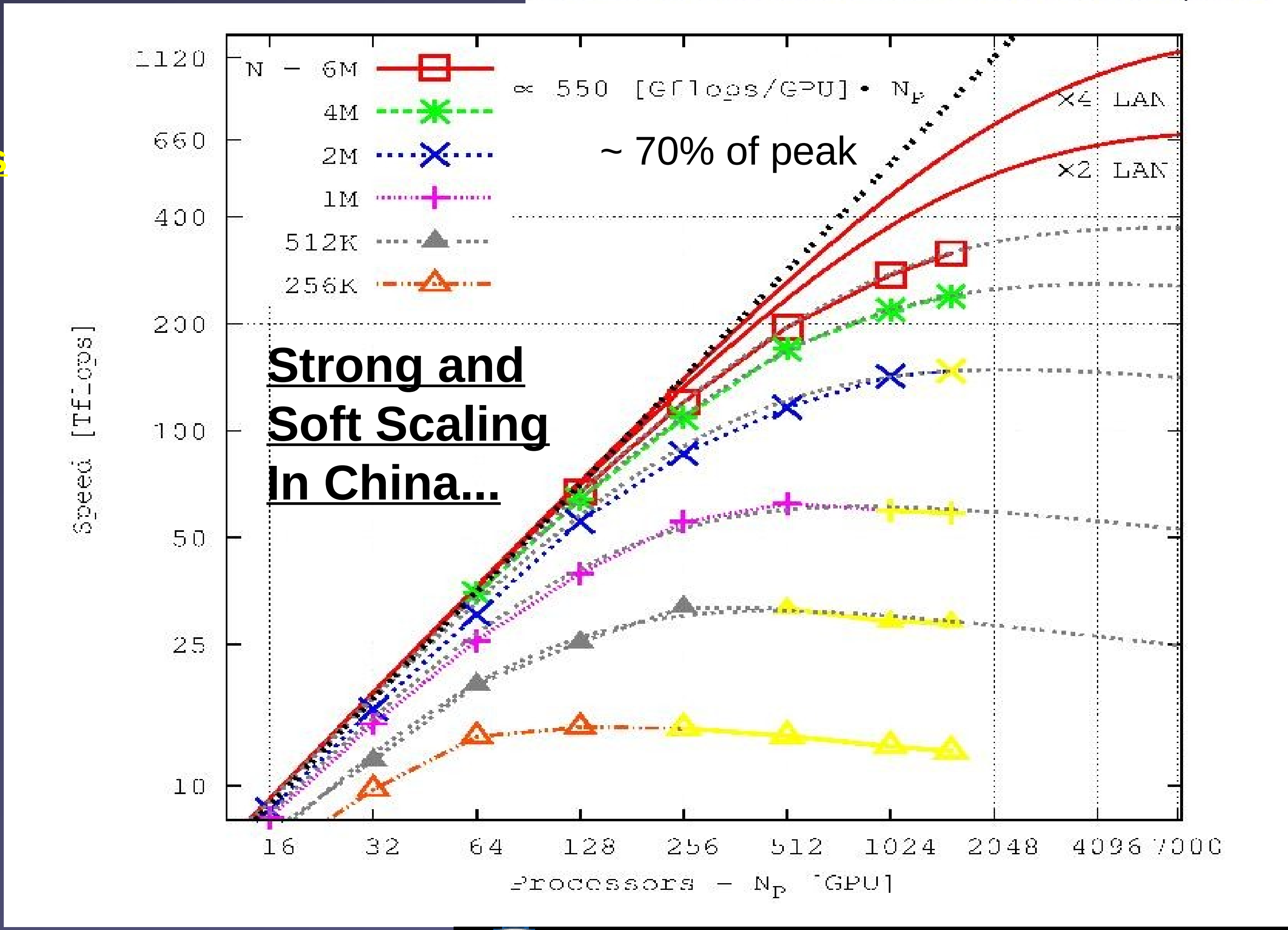
ΦGPU – NBODY Code

中国科学院国家天文台
National Astronomical Observatories, CAS

350 Teraflop/s
1600 GPUs .
440 cores
= 704.000
GPU-Cores

Using Mole-8.5 of IPE/CAS Beijing

Berczik et al. 2013



中国科学院过程工程研究所
Institute Of Process Engineering, Chinese Academy Of Sciences

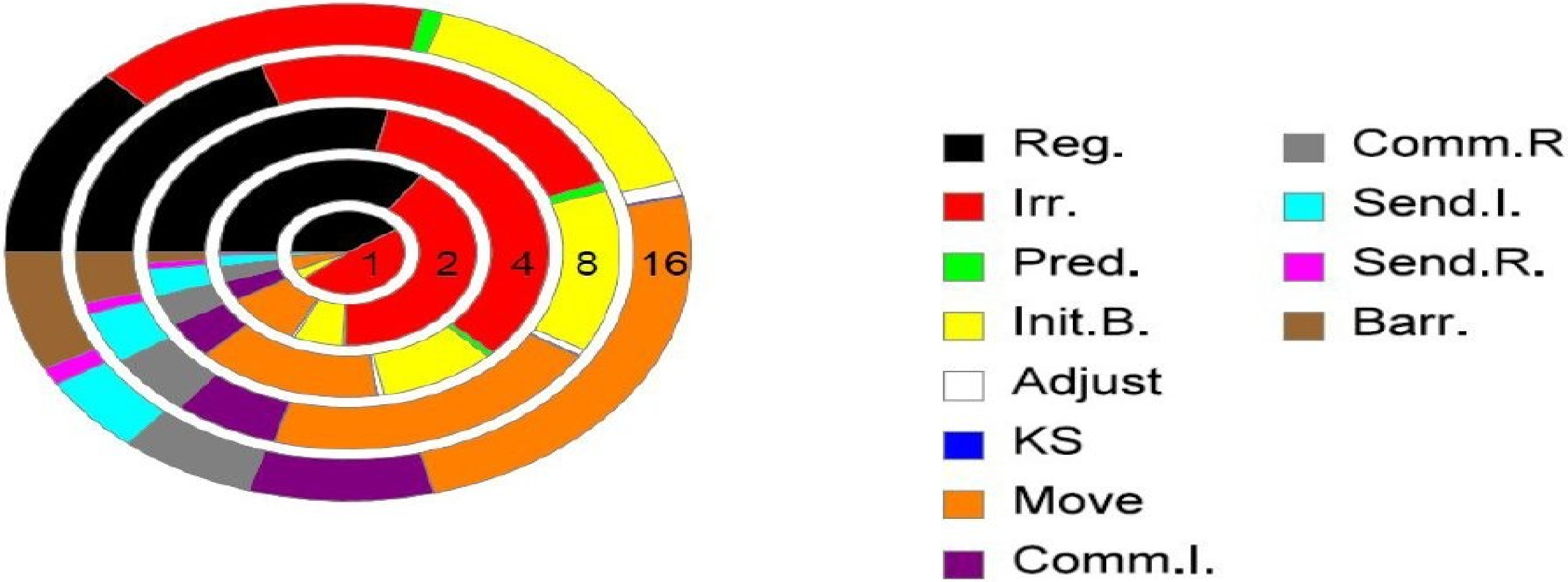
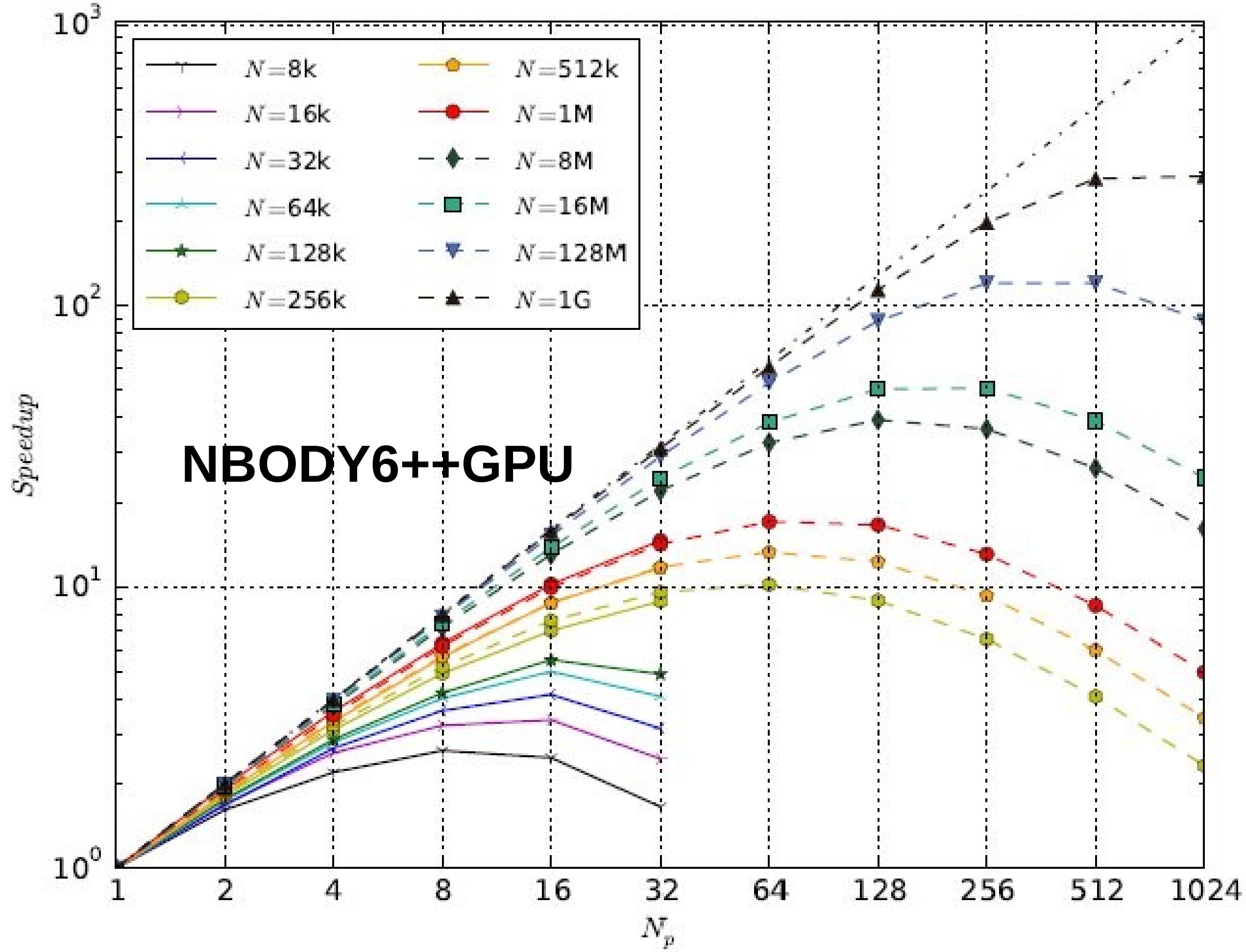


Table 1 Main components of NBODY6++

Description	Timing variable	Expected scaling N	Expected scaling N_p	Fitting value [sec]
Regular force computation	$T_{\rm reg}$	$\mathcal{O}(N_{\rm reg} \cdot N)$	$\mathcal{O}(N_p^{-1})$	$(2.2 \cdot 10^{-9} \cdot N^{2.11} + 10.43) \cdot N_p^{-1}$
Irregular force computation	$T_{\rm irr}$	$\mathcal{O}(N_{\rm irr} \cdot \langle N_{nb} \rangle)$	$\mathcal{O}(N_p^{-1})$	$(3.9 \cdot 10^{-7} \cdot N^{1.76} - 16.47) \cdot N_p^{-1}$
Prediction	$T_{\rm pre}$	$\mathcal{O}(N^{kn_p})$	$\mathcal{O}(N_p^{-kp_p})$	$(1.2 \cdot 10^{-6} \cdot N^{1.51} - 3.58) \cdot N_p^{-0.5}$
Data moving	$T_{\rm mov}$	$\mathcal{O}(N^{kn_{m1}})$	$\mathcal{O}(1)$	$2.5 \cdot 10^{-6} \cdot N^{1.29} - 0.28$
MPI communication (regular)	$T_{\rm mcr}$	$\mathcal{O}(N^{kn_{cr}})$	$\mathcal{O}(kp_{cr} \cdot \frac{N_p-1}{N_p})$	$(3.3 \cdot 10^{-6} \cdot N^{1.18} + 0.12)(1.5 \cdot \frac{N_p-1}{N_p})$
MPI communication (irregular)	$T_{\rm mci}$	$\mathcal{O}(N^{kn_{ci}})$	$\mathcal{O}(kp_{ci} \cdot \frac{N_p-1}{N_p})$	$(3.6 \cdot 10^{-7} \cdot N^{1.40} + 0.56)(1.5 \cdot \frac{N_p-1}{N_p})$
Synchronization	$T_{\rm syn}$	$\mathcal{O}(N^{kn_s})$	$\mathcal{O}(N_p^{kp_s})$	$(4.1 \cdot 10^{-8} \cdot N^{1.34} + 0.07) \cdot N_p$
Sequential parts on host	$T_{\rm host}$	$\mathcal{O}(N^{kn_h})$	$\mathcal{O}(1)$	$4.4 \cdot 10^{-7} \cdot N^{1.49} + 1.23$



Huang, Berczik, Spurzem, Res. Astron. Astroph. 2016, 16, 11.

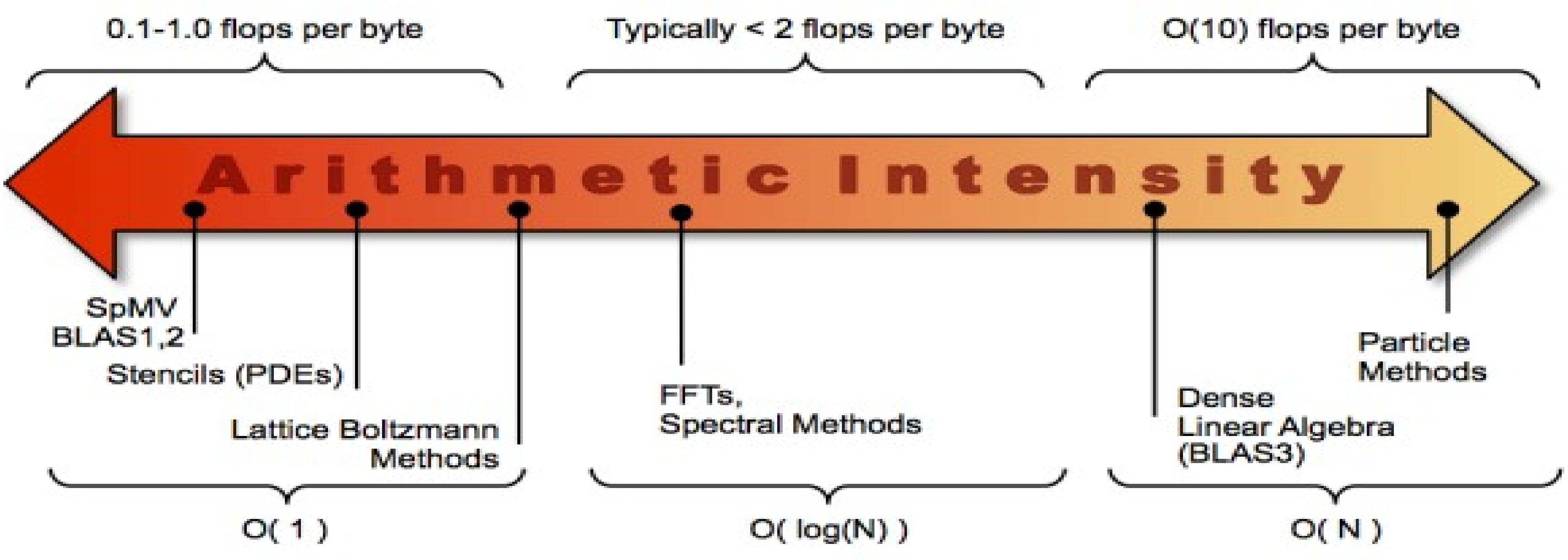
Fig. 2 The speed-up (S) of NBODY6++ as a function of particle number (N) and processor number (N_p). Solid points are the measured speed-up ratio between sequential and parallel wall-clock time, dash lines predict the performance of larger scale simulations further. The symbols used in figure have the magnitudes: $1k = 1,024$, $1M = 1k^2$ and $1G = 1k^3$.

Roofline Performance Model (LBL)

http://crd.lbl.gov/departments/computer-science/PAR/research/roofline

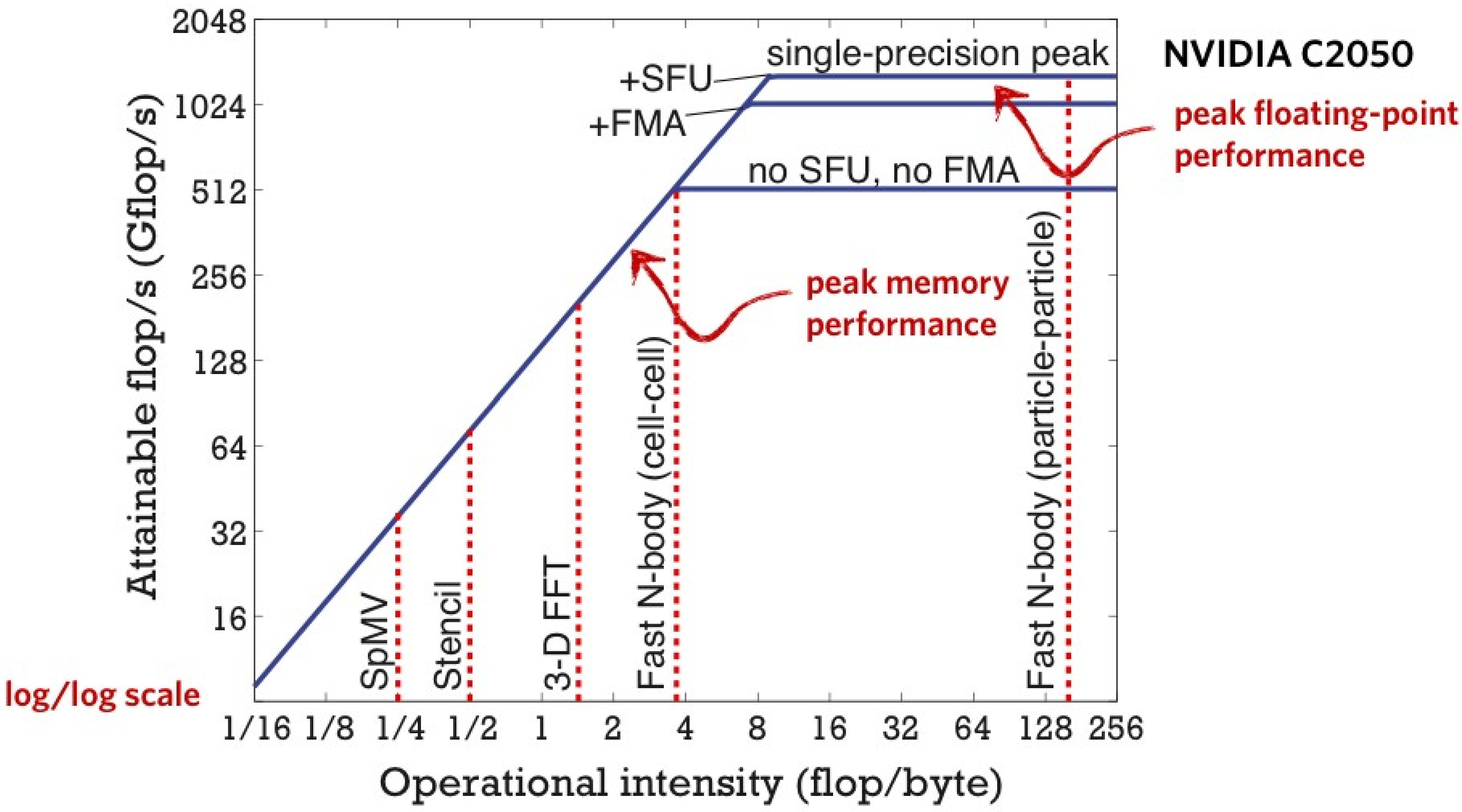
Arithmetic Intensity

The core parameter behind the Roofline model is Arithmetic Intensity. Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes).



Roofline Performance Model (LBL)

http://lorenabarba.com/wp-content/uploads/2012/01/roofline_slide.png



Parallel Computing

Timing and Debugging

Wrap-Up of CUDA

Histogram

Matrix Multiplication (expect Friday)

Before we start...

Some nice ideas:

/home/Tit4/lecture60/gpu-course/00_error/

/home/Tit4/lecture60/gpu-course/4_dot/dot_special.cu

Recap of 6: dot_perfect.cu :

Fat Threads! New variable gridDim.x !

Block Reduction on Host instead of AtomicAdd!

Also used for histogram later.

Timing with CUDA Event API

```
int main ()
{
     cudaEvent_t start, stop;
     float time;

     cudaEventCreate (&start);
     cudaEventCreate (&stop);

     cudaEventRecord (start, 0);

     someKernel <<<grids, blocks, 0, 0>>> (...);

     cudaEventRecord (stop, 0);
     cudaEventSynchronize (stop);

     cudaEventElapsedTime (&time, start, stop);

     cudaEventDestroy (start);
     cudaEventDestroy (stop);

     printf ("Elapsed time %f sec\n", time*.001);

     return 1;
}
```

← Ensures kernel execution has completed (cudaEventSynchronize)

CUDA Event API Timer are,

- OS independent
- High resolution
- Useful for timing asynchronous calls

Standard CPU timers will not measure the timing information of the device.

CUDA – GNU Debugger – CUDA-gdb

http://docs.nvidia.com/cuda/cuda-gdb/index.html

NVIDIA DEVELOPER ZONE CUDA TOOLKIT DOCUMENTATION Search

CUDA-GDB (PDF) - v7.5 (older) - Last updated September 1, 2015 - Send Feedback -

CUDA-GDB

1. Introduction

This document introduces CUDA-GDB, the NVIDIA® CUDA® debugger for Linux and Mac OS.

1.1. What is CUDA-GDB?

CUDA-GDB is the NVIDIA tool for debugging CUDA applications running on Linux and Mac. CUDA-GDB is an extension to the x86-64 port of GDB, the GNU Project debugger. The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables developers to debug applications without the potential variations introduced by simulation and emulation environments.

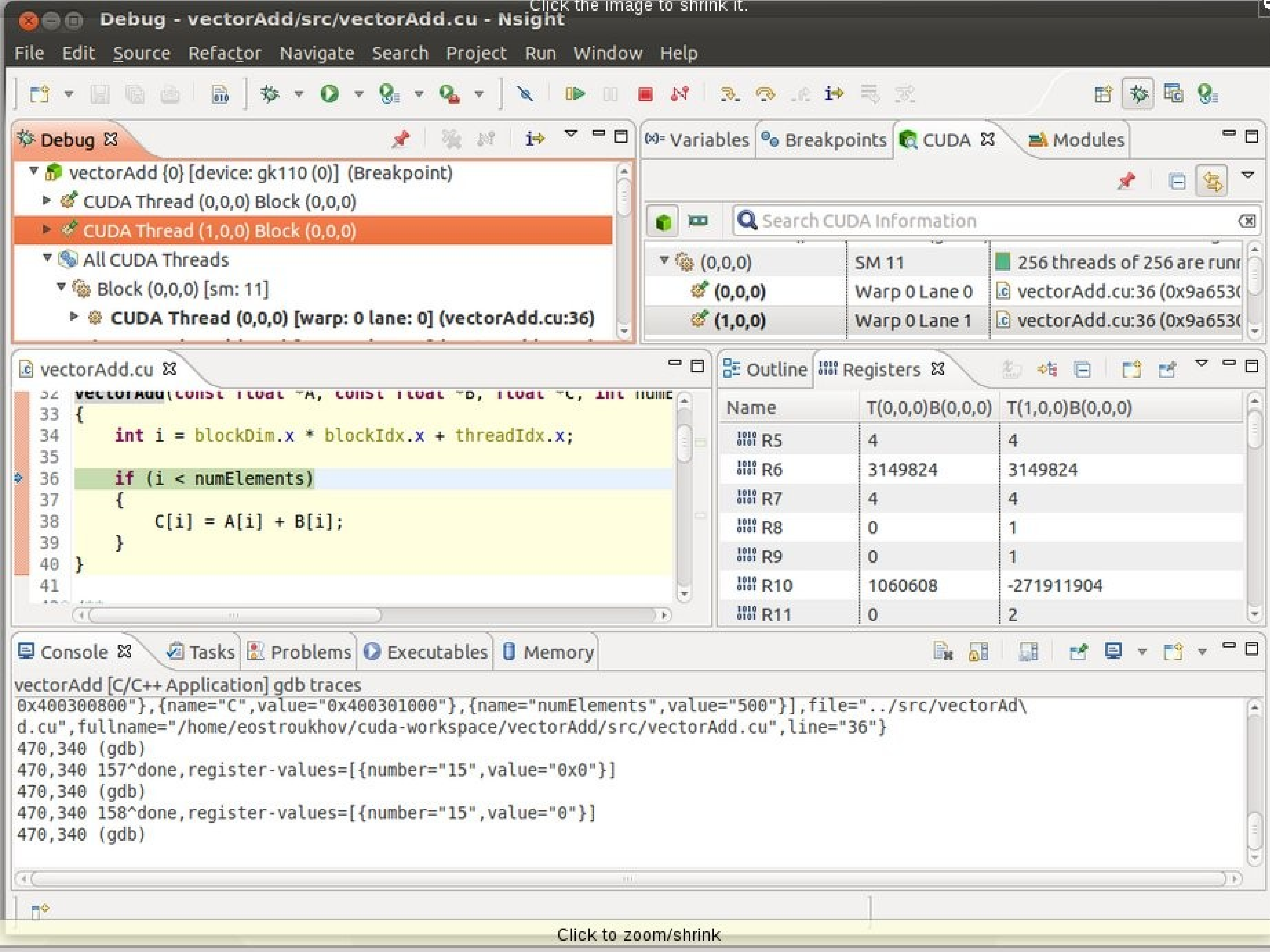
CUDA-GDB runs on Linux and Mac OS X, 32-bit and 64-bit. CUDA-GDB is based on GDB 7.6 on both Linux and Mac OS X.

1.2. Supported Features

CUDA-GDB is designed to present the user with a seamless debugging environment that allows simultaneous debugging of both GPU and CPU code within the same application. Just as programming in CUDA C is an extension to C programming, debugging with CUDA-GDB is a natural extension to debugging with GDB. The existing GDB debugging features are inherently present for debugging the host code, and additional features have been provided to support debugging CUDA device code.

CUDA-GDB supports debugging C/C++ and Fortran CUDA applications. (Fortran debugging support is limited to 64-bit Linux operating system) All the C++ features supported by the NVCC compiler can be debugged by CUDA-GDB.

CUDA-GDB allows the user to set breakpoints, to single-step CUDA applications, and also to inspect and modify the memory and variables of any given thread running on the hardware.



Click the image to shrink it.
Debug - vectorAdd/src/vectorAdd.cu - Nsight
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
vectorAdd {0} [device: gk110 (0)] (Breakpoint)
CUDA Thread (0,0,0) Block (0,0,0)
CUDA Thread (1,0,0) Block (0,0,0)
All CUDA Threads
Block (0,0,0) [sm: 11]
CUDA Thread (0,0,0) [warp: 0 lane: 0] (vectorAdd.cu:36)
Variables Breakpoints CUDA Modules
Search CUDA Information
(0,0,0) SM 11 256 threads of 256 are runr
(0,0,0) Warp 0 Lane 0 vectorAdd.cu:36 (0x9a653(
(1,0,0) Warp 0 Lane 1 vectorAdd.cu:36 (0x9a653(
vectorAdd.cu
{
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < numElements)
{
C[i] = A[i] + B[i];
}
}
Outline Registers
Name T(0,0,0)B(0,0,0) T(1,0,0)B(0,0,0)
R5 4 4
R6 3149824 3149824
R7 4 4
R8 0 1
R9 0 1
R10 1060608 -271911904
R11 0 2
Console Tasks Problems Executables Memory
vectorAdd [C/C++ Application] gdb traces
0x400300800"},{name="C",value="0x400301000"},{name="numElements",value="500"}],file="../src/vectorAd\
d.cu",fullname="/home/eostroukhov/cuda-workspace/vectorAdd/src/vectorAdd.cu",line="36"}
470,340 (gdb)
470,340 157^done,register-values=[{number="15",value="0x0"}]
470,340 (gdb)
470,340 158^done,register-values=[{number="15",value="0"}]
470,340 (gdb)
Click to zoom/shrink

Wrapping Up 1

Exercises (CUDA Lectures in afternoon)

0. hello, device- first kernel call, hello world, GPU properties
1. add - vector addition using one thread in one block only
2. add-index - vector addition using blocks in parallel, one thread per block only.
3. add-parallel - vector addition using all blocks and threads in parallel
4. dot - scalar product using shared memory of one block only for reduction
5. dot-full - scalar product using shared memory and atomic add across blocks
6. dot-perfect - scalar product; fat threads and final reduction on host.

8. histo - histogram using fat threads and atomic add on shared and global memory, timing

7. matmul - matrix multiplication with tiled access shared memory (expect Friday)

Wrapping Up 2

Elements of CUDA C learnt:

threadId.x , blockId.x, blockDim.x, gridDim.x	Threads, Blocks
(threadId.y, blockId.y, blockdim.y, gridDim.y	(matmul coming with 2D grids)
kernel<<<n,m>>> (...)	kernel calls
kernel<<<dimBlock,dimGrid>>>(…)	dim3 variable type (matmul)
__global__	device code
__shared__	shared memory on GPU
cudaMalloc / cudaFree	manage global memory of GPU
cudaMemcpy / cudaMemset	copy/set to or from memory
cudaGetDeviceProperties	get device properties in program
cudaEventCreate, cudaEventRecord, cudaEventSynchronize, cudaEventElapsedTime, cudaEventDestroy	CUDA profiling
AtomicAdd	atomic functions

Wrapping Up 3

What we have not yet learnt...

__constant__	constant memory on GPU
__device__	functions device to device
Intrinsic Functions (__device__ type)	
https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__SINGLE.html#group__CUDA__MATH__SINGLE	
__host__	functions host to host
More atomic functions	
cudaBindTexture	using texture memory
fat threads for 2D and 3D stencils	thread coalescence opt.
cudaStreamCreate, cudaStreamDestroy	working with CUDA streams
using Tensor Cores	
...	

Histogram

Chapter in Book of Jason Sanders

https://wwwstaff.ari.uni-heidelberg.de/spurzem/lehre/WS20/cuda/files/cuda-histograms.pdf

Link on our webpage

On kepler: 8_histo

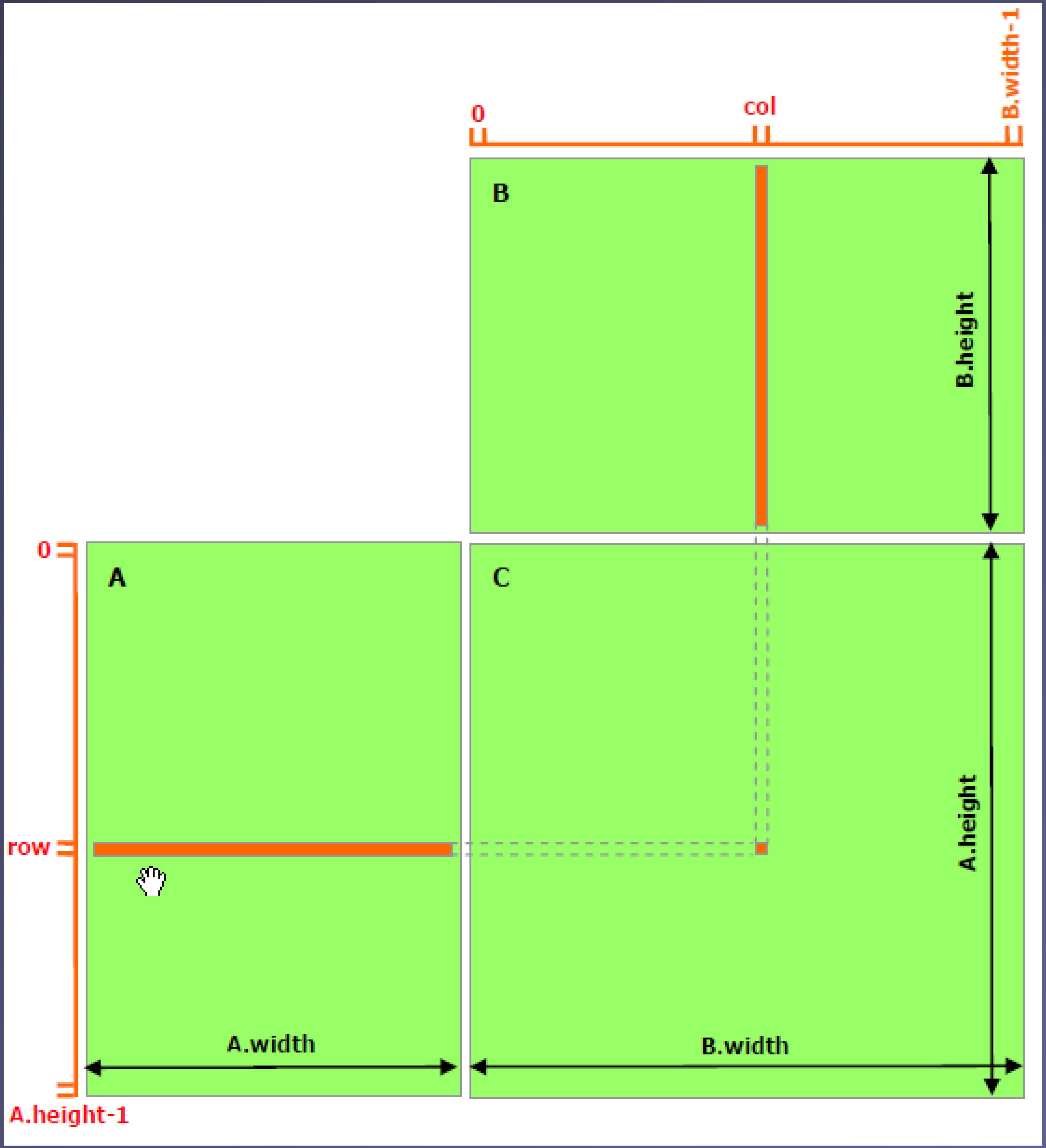
histo.cu

histo-no-atomic.cu

Both use atomic on shared memory!

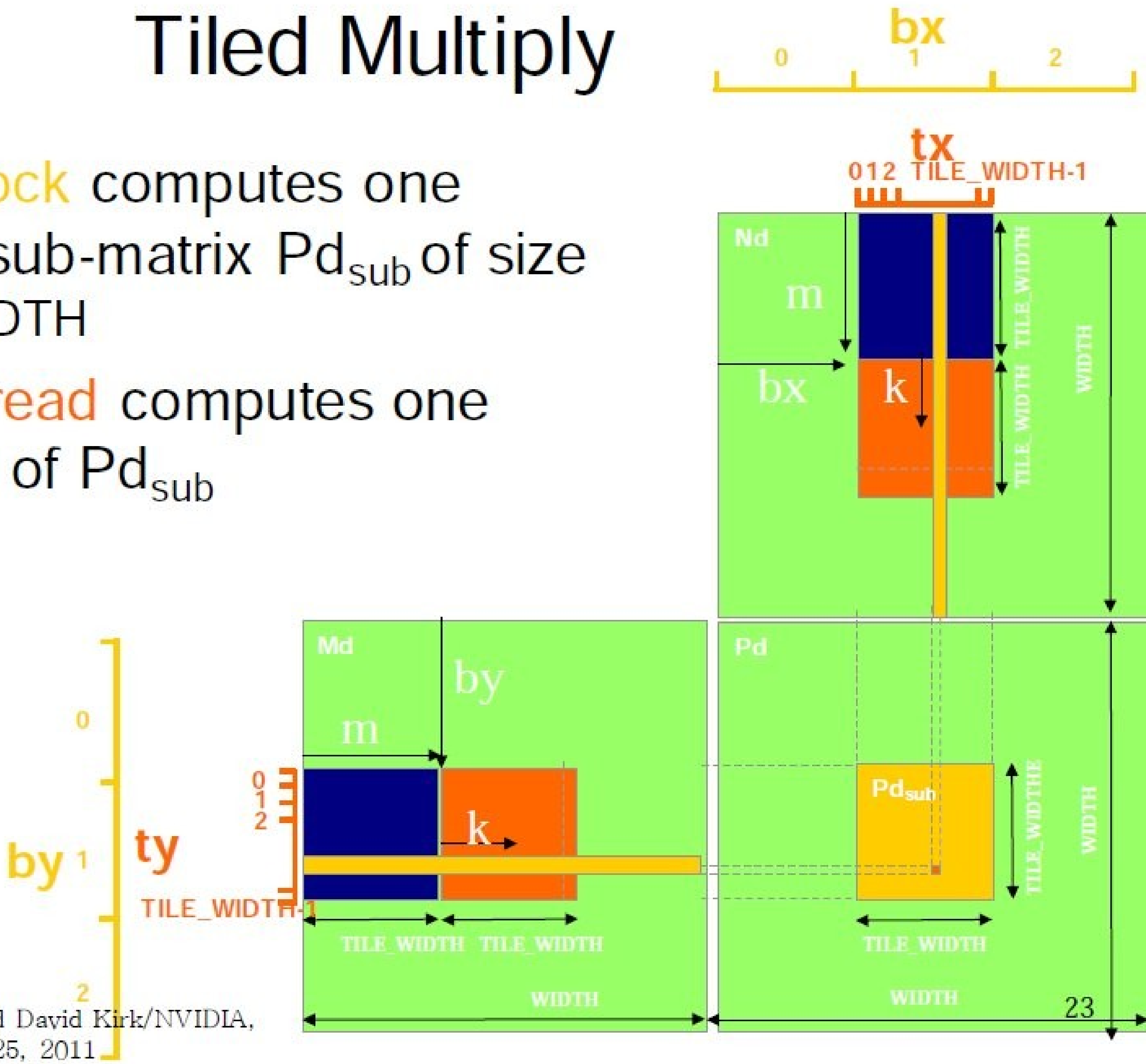
But only first one uses also atomic on global memory!

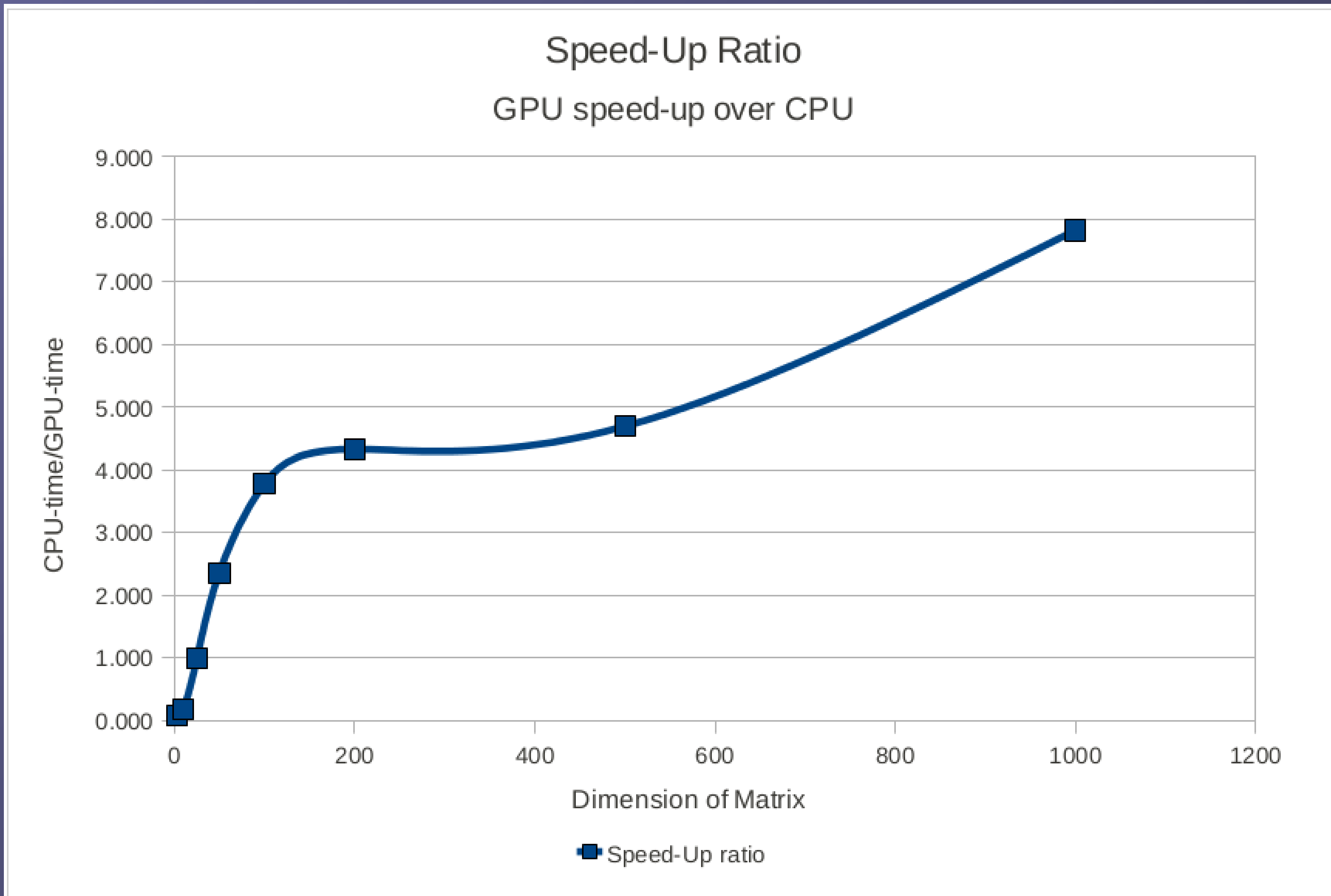
Intuitive multiply



Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}




Speed-Up Ratio
GPU speed-up over CPU
9.000
8.000
7.000
6.000
5.000
4.000
3.000
2.000
1.000
0.000
CPU-time/GPU-time
0
200
400
600
800
1000
1200
Dimension of Matrix
Speed-Up ratio