GPGPU Programming
Performance and Libraries

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Overview

1. GPU Performance
   - Compute Capability
   - Device Occupancy
   - How Fast Is Very Fast?
2. Tools
   - Development
   - Performance
3. Libraries
   - Thrust
   - CUBLAS
   - CUSPARSE/PSBLAS/SPGPU
   - CUFFT
4. Case studies:
   - Particle algorithms
   - Biomedical imaging

GPU Performance

In this section you will hear about...

- Compute Capability (different generations of GPUs)
- How to keep your GPU busy (device occupancy)
- How to measure GPU codes performance
- How fast the GPUs really are

GPUs Are Evolving

Compute Capability: hardware specifications of the GPU

- Number of cores per Streaming Multiprocessor (SM)
- Global memory transactions handling
- Amount of shared memory and registers
- Number of shared memory banks
- Support for double precision arithmetic
- Concurrent kernel execution and memory transfers
- Significant differences between major revisions
- Incremental differences between minor revisions

Support for CC 1.x was dropped with CUDA 7; support for CC 2.x was dropped in CUDA 9.

Compute Capability Highlights: The Original Series (1.x)

Compute Capability 1.0

- Up to 512 threads per block
- Up to 768 threads (24 warps) resident on an SM
- 16 KB shared memory in 16 banks per SM
- 8K 32-bit (32 KB) registers per SM
- 8 CUDA cores per SM

Further improvements

- Concurrent kernel execution and one-way CPU-GPU data transfer (CC 1.1)
- Significantly loosened coalescing requirements (CC 1.2)
- Up to 1,024 threads (32 warps) resident on an SM (CC 1.2)
- 16K 32-bit (64 KB) registers per SM (CC 1.2)
- (Slow) support for double precision arithmetic (CC 1.3)

Compute Capability Highlights: Fermi

Compute Capability 2.0

- 1,024 threads per block
- Up to 1,536 threads (48 warps) resident on an SM
- Up to 48 KB shared memory in 32 banks per SM
- 32K 32-bit (128 KB) registers per SM
- 32 CUDA cores per SM

Further improvements

- Global memory caching Up to 48 KB L1 cache per SM,
- 768 KB L2 cache per device (CC 2.0)
- Significantly improved double precision performance (CC 2.0)
- Concurrent multiple kernel execution and two-way CPU-GPU data transfers (CC 2.0)
- 48 CUDA cores per SM (CC 2.1)

Compute Capability Highlights: Kepler

Compute Capability 3.0

- Up to 2,048 threads (64 warps) resident on an SM
- 64K 32-bit (256 KB) registers per SM
- 192 CUDA cores per SM

Further improvements

- Support for PCI-Express 3.0 (faster CPU-GPU transfers)
- Warp Shuffle allows for efficient intra-warp communication
- Truly independent concurrent CUDA streams (Hyper-Q)
- Dynamic parallelism Kernels can be called from within another kernel (CC 3.5)

Compute Capability Highlights: Maxwell and Pascal

Maxwell: CC 5.x (per SM, 5.0/5.2):

- 128 CUDA cores
- 4 warp schedulers
- 32 single-precision transcendental functions units
- 64/96 KB shared memory

Pascal: CC 6.x (per SM, 6.0/6.2):

- 64/128 CUDA cores per SM
- 2/4 warp schedulers
- 16/32 single-precision transcendental functions units
- 64/96 KB shared memory

Compute Capability Highlights: Volta

Volta: CC 7.x (per SM):

- 64 FP32 cores for single-precision;
- 32 FP64 cores for double-precision;
- 64 INT32 cores for integer math;
- 8 mixed-precision Tensor Cores for deep learning;
- 16 single-precision transcendental functions units;
- 4 warp schedulers.

Compute Capability Highlights: Turing

Turing: CC 7.5 (per SM):

- 64 FP32 cores for single-precision;
- 2 FP64 cores for double-precision;
- 64 INT32 cores for integer math;
- 8 mixed-precision Tensor Cores for deep learning;
- 4 warp schedulers.

Device Occupancy

How many threads are executed on my GPU?

- Depends on Compute Capability
- Depends on demand on shared memory and registers

Example (Compute Capability 2.0)

- Maximum of 1,536 threads or 48 warps per SM
- 32K / 1,536 $\approx$ 20 32-bit registers per thread
- 48 KB / 1,536 = 32 bytes of shared memory per thread

If more shared resources are required, the number of threads resident on an SM is limited!

$$\text{Device Occupancy} = 100\% \frac{\#\text{ of threads resident on SM}}{\text{Maximum }\#\text{ of threads on SM}}$$

Digression: Computational Problem Types

Memory-bound problems

- Only few operations are performed per input value
- Typically problems with linear time complexity
  - AXPY operation
  - Sparse matrix-vector multiplication
  - Solving tridiagonal systems
  - Stencil operators (finite difference method for PDEs)

Compute-bound problems

- Many operations are performed per input value
- Typically problems with higher than linear time complexity
  - Fast Fourier Transform (FFT)
  - Matrix-matrix multiplication (GEMM)

A problem can be memory- or compute-bound depending on a particular architecture or even the problem instance!

How Does Device Occupancy Affect the Performance?

Memory-bound problems

- ✗ Many requests to global memory, many warps are inactive
- ✗ When there are no active threads, the SM is idle
- ✓ High device occupancy means more resident warps, effectively reducing SM idle time (hiding memory latency)
- ✓ Conclusion: the higher device occupancy, the better

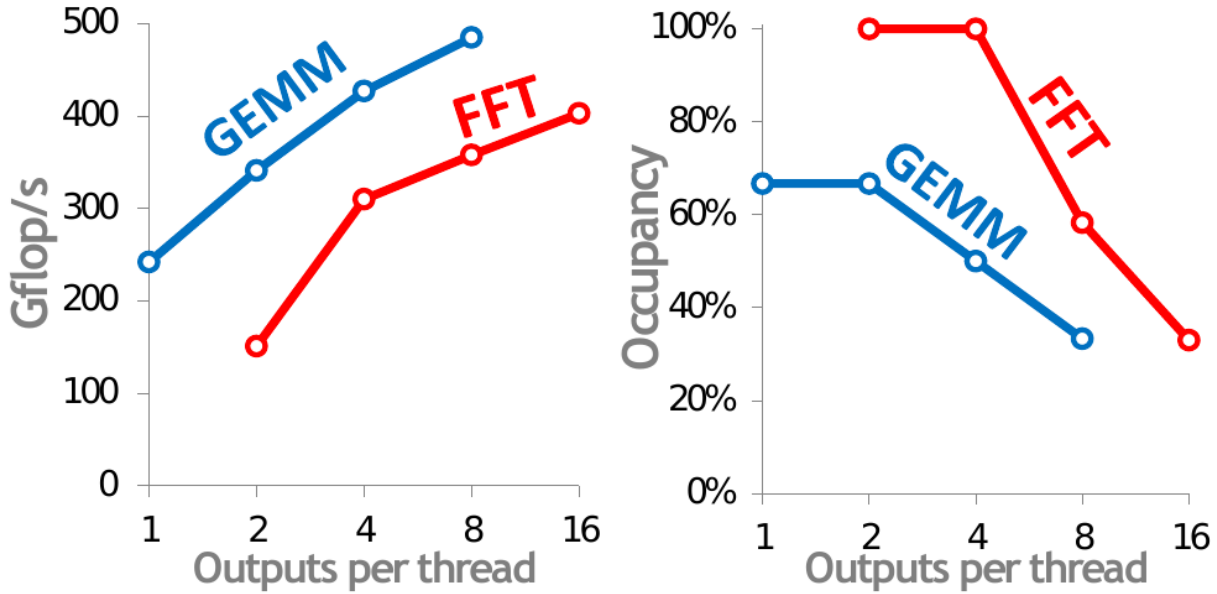
Compute-bound problems

- Fewer requests to global memory, more computation on data in shared memory and registers
- Global memory latency hiding not crucial for performance
- Conclusion: Lower occupancy is acceptable...
- ...but still need to have enough threads to keep the GPU busy

High Occupancy == High Performance?

Do I really need high device occupancy?

- As a rule of thumb: yes, try to keep it above 50
- However, it is possible to get good performance at low occupancy for compute-bound problems



Source: V. Volkov, Better Performance at Lower Occupancy (2010), http://www.cs.berkeley.edu/ volkov/volkov10-GTC.pdf

Performance Limits

Why do we even care about GPUs?

- ✗ They are difficult to program
- ✓ GPUs can be many times faster than CPUs
- ✓ They are cheaper to buy (GFLOPS per £)
- ✓ They are cheaper to run (GFLOPS per Watt)

Intel i7-980x (6 cores)

- 80 GFLOPS (DP)
- 25.6 GB/s (main memory)
- £ 680
- 130 Watts

Kepler K40 (2880 cores)

- 1680 GFLOPS (DP)
- 288 GB/s (global memory)
- £ 2000
- 235 Watts

How NOT to measure performance

$$\text{Speed.up} = \frac{T_{CPU}}{T_{GPU}}$$

- Your GPU code is $100\times$ faster than your CPU code...
- ...but how good is your CPU code?!
- Should the CPU-GPU data transfer be included in T_{GPU} ?
- Metric often (incorrectly) used in the literature!
- Can be meaningful if measured against a well-established CPU implementation (Atlas, Intel MKL, FFTW3)

How to measure performance

- Use the correct metric:

  Operation throughput for compute-bound problems
  Memory throughput for memory-bound problems
- Compare against the device theoretical peak performance

Operation Throughput (1/3): Definition and measure

$$\text{Throughput} = \frac{\#\text{ of floating -point instructions}}{\text{Time}}[GFLOPS]$$

Counting instructions (performance model)

- Only operations in the algorithm definition should be counted
- Indexing, loop iteration etc. should be omitted
- Can use well-established models for known problems, e.g.
  - Thomas Algorithm (tridiagonal solver): $8n$
  - Fast Fourier Transform: $5n \log n$
- Alternative models can be used:
  - Number of frames per second (rendering)
  - Number of comparisons per second (sorting)
  - Number of passwords checked per second (cryptography)

Operation Throughput (2/3): Peak Performance

Single precision

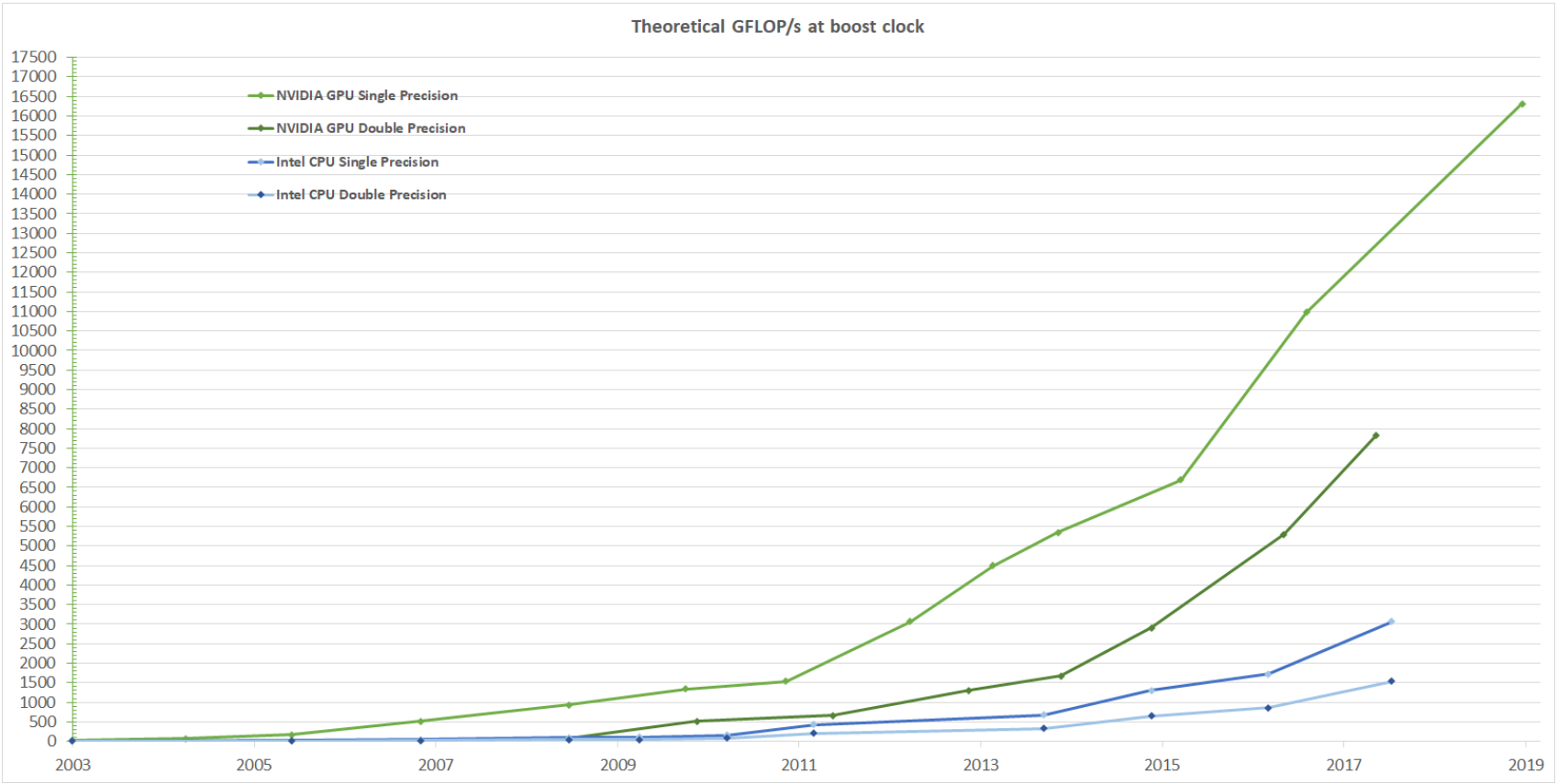
$$\#\text{ of CUDA cores} \times \text{clock rate} \times \text{operations per cycle}$$

- Typically two FLOPS per cycle (Fused Multiply-Add)
- K80: 4992 cores, 560/875 MHz, 5591/8736 GFLOPs

Double precision

- On HPC equipment (e.g. Tesla C2050) typically half the single precision performance (K80 1864/2912 GFLOPS)
- *May be significantly lower on gaming cards!*
- GTX 480: 1345 GFLOPS SP, 168 GFLOPS DP

Operations Throughput (3/3): CPU versus GPU



Source: CUDA Programming Guide

Memory Throughput (1/2): Definition and measure

$$\text{Throughput} = \frac{\text{Amount of data transferred}}{\text{Time}} [GB/s]$$

Counting data (performance model)

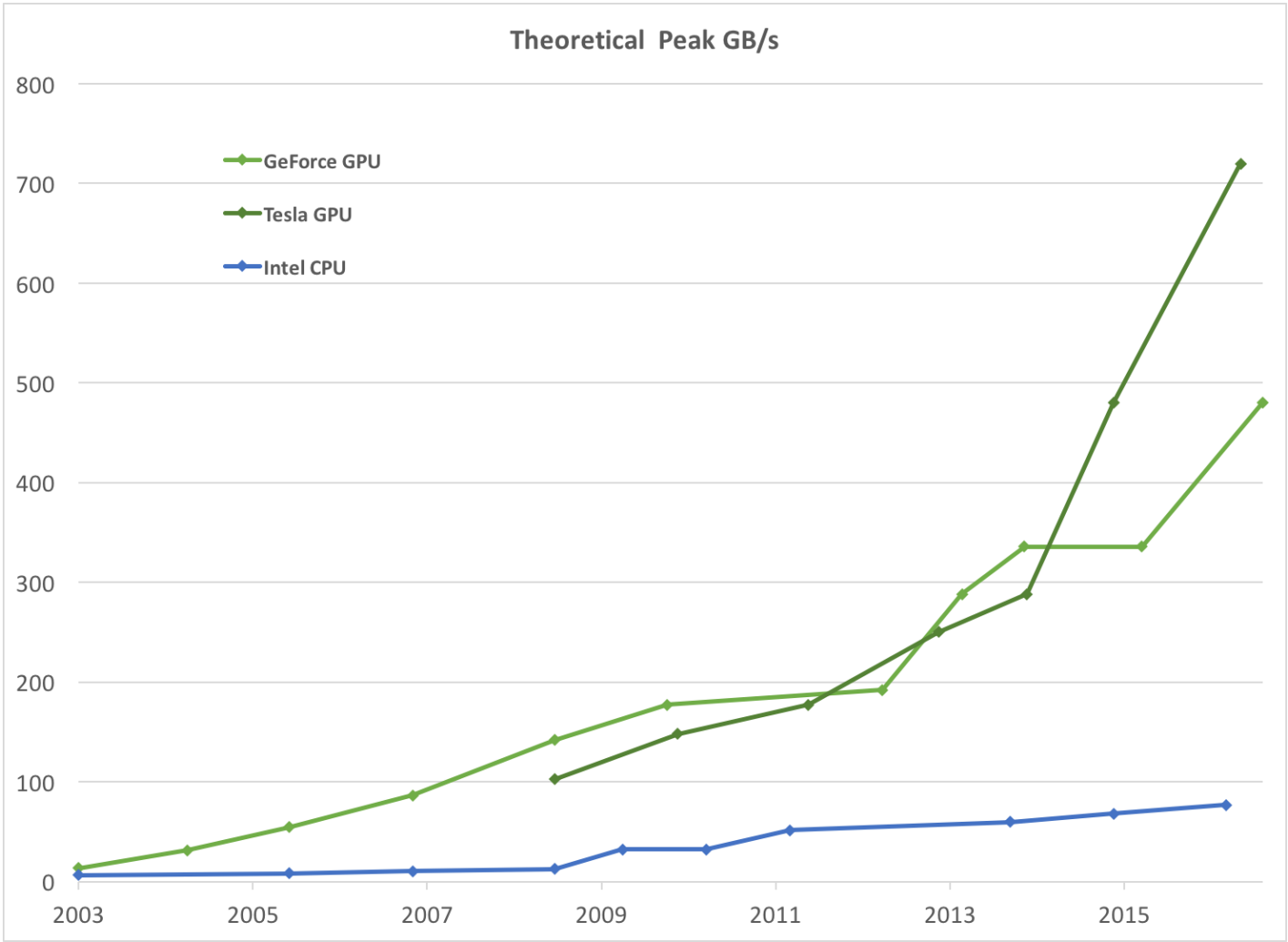
- Only the input and the output data should be counted
- Intermediate reads/writes should be omitted

Theoretical maximum bandwidth

$$2 \times \text{memory bus width} \times \text{ memory clock rate}$$

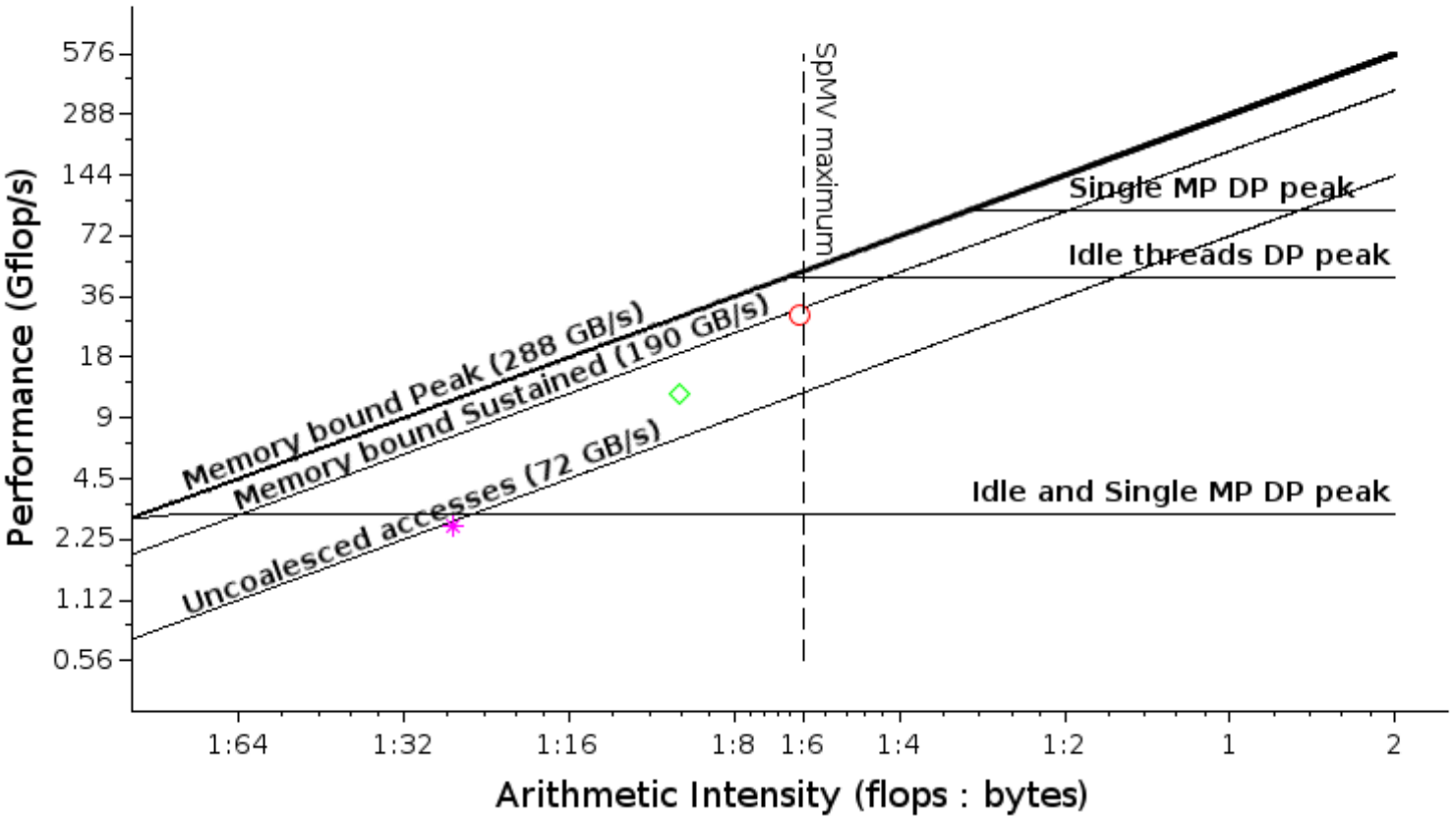
- Tesla C2050: 384-bit bus, 1,500 MHz = 144 GB/s
- GTX 480: 384-bit bus, 1,848 MHz = 177.4 GB/s
- Kepler K40: 384-bit bus, 3,000 MHz = 288 GB/s

Memory Throughput (2/2): CPU versus GPU



Source: CUDA Programming Guide

An Example: Roofline for SPMV K40



Summary

- Compute Capability determines hardware specs of the GPU
  - CC 1.2: loosened coalescing requirements
  - CC 1.3: support for double precision arithmetic
  - CC 2.0: global memory caching, fast DP arithmetic
  - CC 3.5: dynamic parallelism
- The number of threads resident on SM (device occupancy) is determined by demand for registers and shared memory
  - High occupancy is crucial for memory-bound algorithms
  - Compute-bound problem can perform well at lower occupancy
- Modern GPUs are several times faster than modern CPUs
- The GPUs are more cost-efficient (initial price, running costs)
- Using the correct metric is essential for meaningful performance measurement
  - Instruction throughput for compute-bound problems
  - Memory throughput for memory-bound problems
  - Speed-up over CPU is only meaningful in case of the well-established CPU codes

Tools

In this section you will hear about...

- Development tools for writing GPU codes
- Using Visual Profiler to optimise your code

Nsight

Development platform supported by NVIDIA

- Available on Windows (Visual Studio Edition)
- Available on Linux (Eclipse Edition)
- Includes GPU code debugger
- Includes Visual Profiler
- Provides syntax highlighting and code completion

Additional Tools (Linux only)

cuda-memcheck

This is your new best friend!

- Memory checker, similar to `valgrind`
- Detects memory accesses outside the allocated memory
- Detects CUDA API calls errors
- Detects global memory leaks[a]
- Possibly detects race conditions

[a] Your code should call `cudaResetDevice()` before exiting!

cuda-gdb

gdb-like debugger for GPU codes

CUDA Occupancy Calculator (1/2)

Part of CUDA Toolkit (`tools/CUDA_Occupancy_Calculator.xls`)

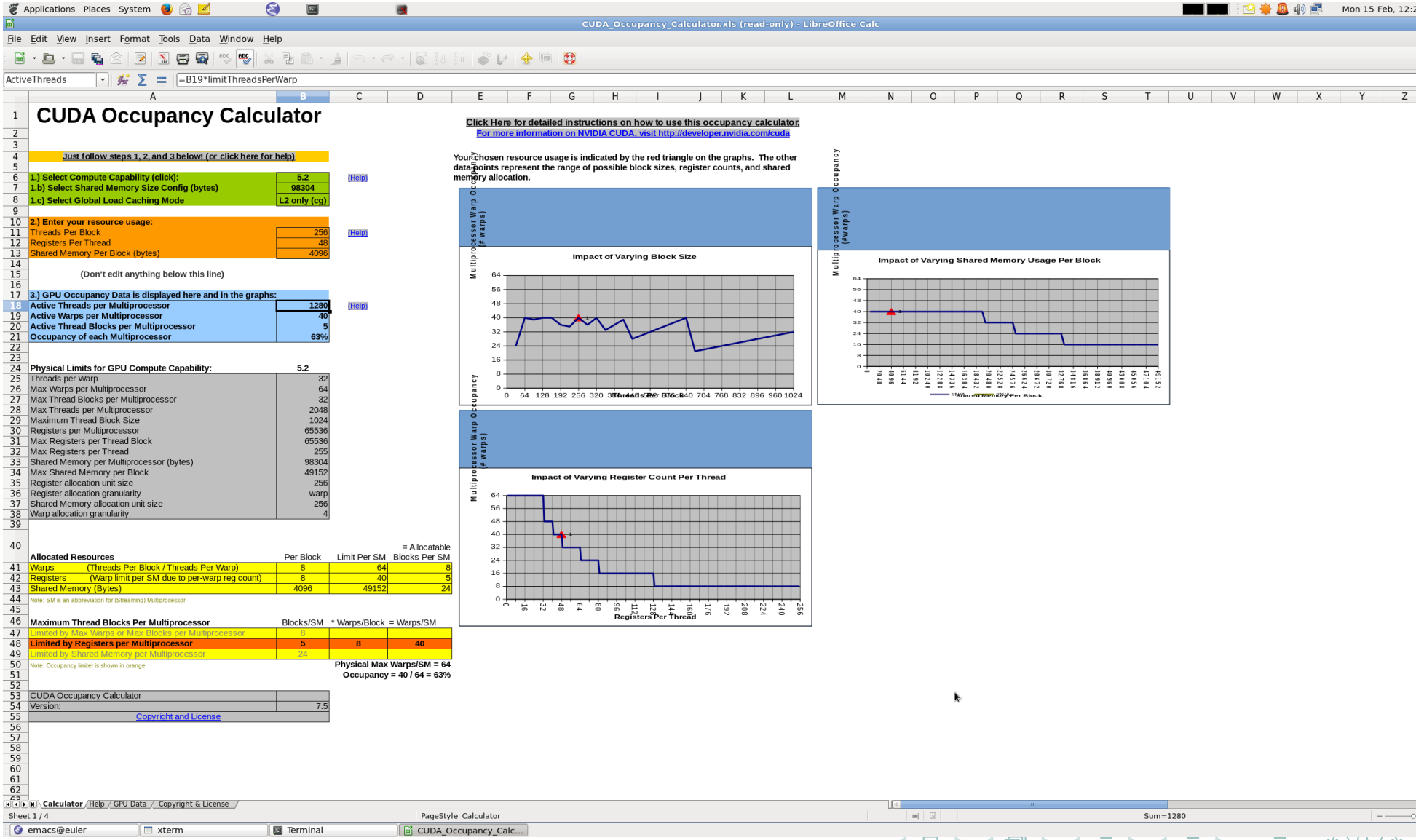
Input (per kernel)

- Device compute capability
- Number of threads per block
- Number of registers per thread
- Amount of shared memory per block

Output

- Maximum number of resident threads per SM (maximum device occupancy)
- Identifies bottlenecks (occupancy limiting factors)

CUDA Occupancy Calculator (1/2)



Visual Profiler

- Part of CUDA Toolkit (bin/nvvp)
- Presents detailed kernel execution breakdown
- Identifies possible performance limiting factors

Comprehensible execution timeline

- host-side API calls and kernel execution
- host-device and device-device memory transfers
- kernel execution

Per-kernel performance statistics

- Execution time
- Dispatch parameters (grid/block size, registers)
- Global memory utilisation (coalescing, effective bandwidth)
- Instruction overheads (divergent warps, cache misses)
- Device occupancy (theoretical, achieved)

Visual Profiler

You can also split the analysis

Collect data (on remote machine)

```
nvprof -f --export-profile myapp.nvprof  ./myapp
```

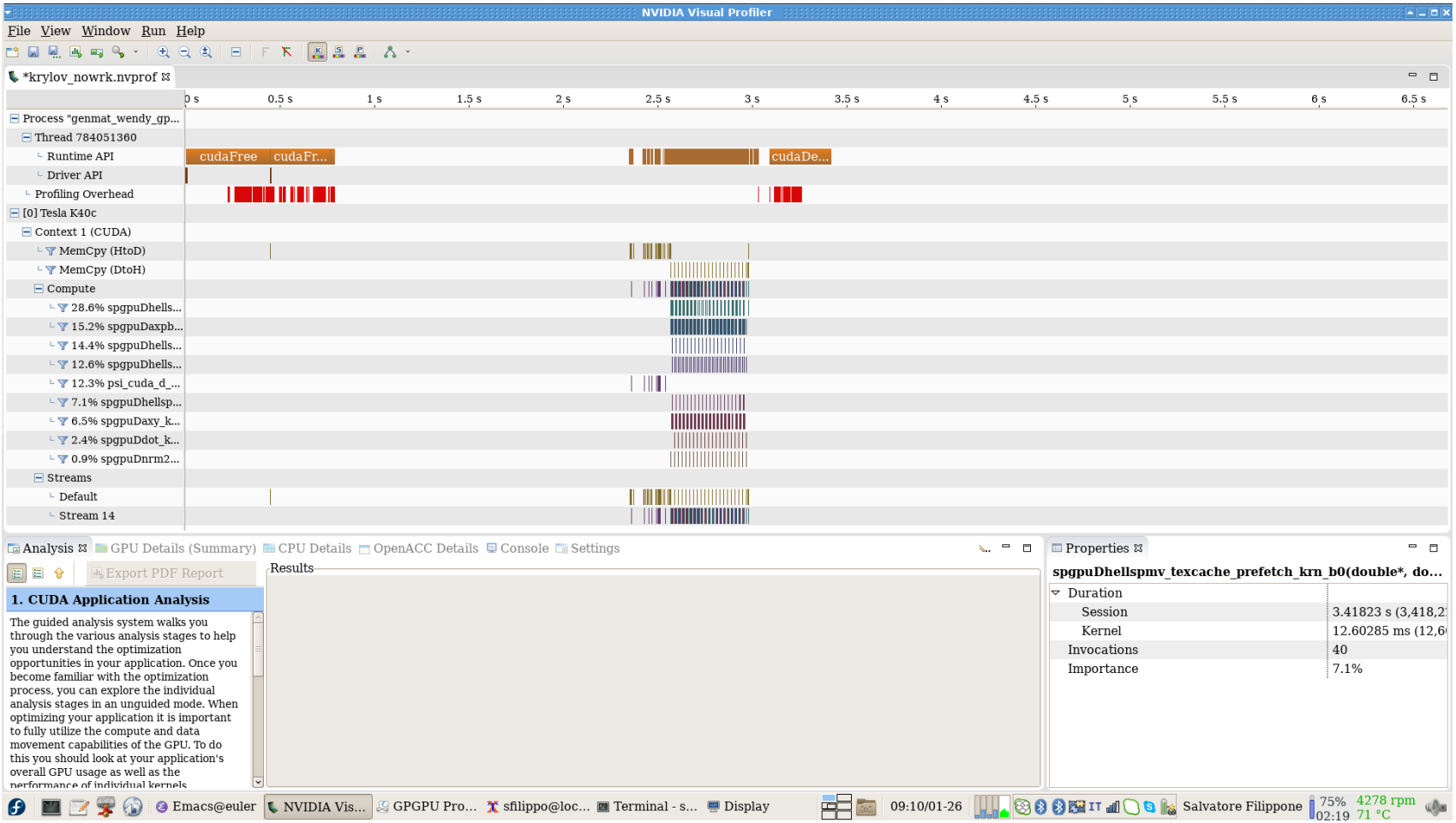
Visualize profile (on local machine)

```
nvvp  myapp.nvprof
```

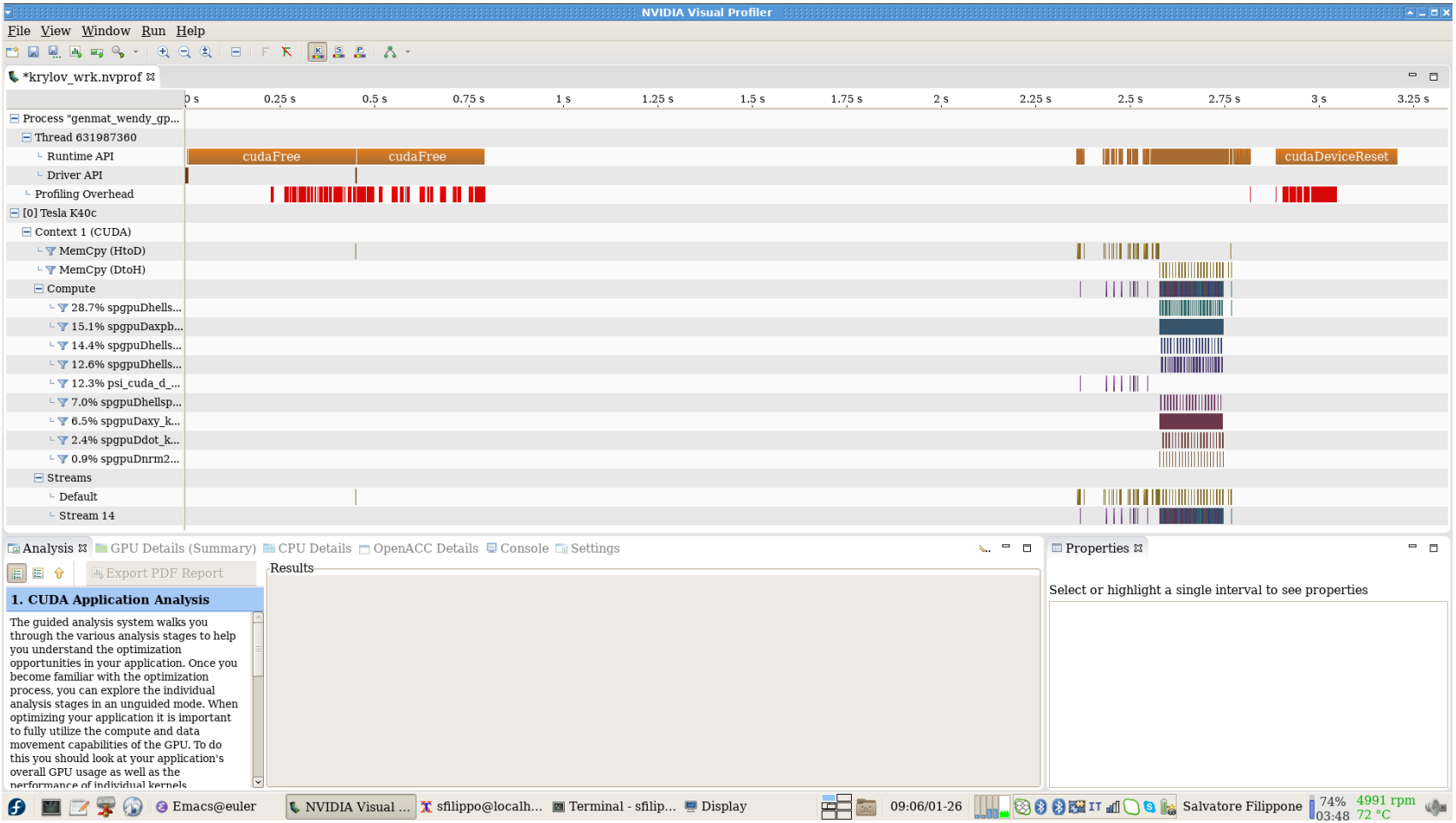
Solving a linear system with CG preconditioned with AMG on a GPU.

	CPU	GPU v1
Iterations	55	55
Error on exit	8.41503E-07	8.41503E-07
Levels	3	3
Prec build time	1.13570E+00	1.25210E+00
Time to soution	4.81588E+00	9.67357E-01
Time per iteration	8.75614E-02	1.75883E-02

Visual Profiler



Visual Profiler



Solving a linear system with CG preconditioned with AMG on a GPU.

	CPU	GPU v1	GPU v2
Iterations	55	55	55
Error on exit	8.41503E-07	8.41503E-07	8.41503E-07
Levels	3	3	3
Prec build time	1.13570E+00	1.25210E+00	1.26119E+00
Time to soution	4.81588E+00	9.67357E-01	4.73947E-01
Time per iteration	8.75614E-02	1.75883E-02	8.61722E-03

Libraries

In this section you will hear about...

- How not to reinvent the wheel...
- Thrust (high-level memory interface, basic parallel programming routines)
- CUBLAS (dense linear algebra on the GPU)
- CUSPARSE (sparse linear algebra on the GPU)
- CUFFT (Fast Fourier Transform on the GPU)
- PSBLAS-EXT/SPGPU (sparse linear algebra on hybrid MPI/CUDA)

Thrust

"Code at the speed of light"

- C++ template library based on
- Standard Template Library (STL)
- High-level interface fully interoperable with CUDA C

- ✓ Great for rapid prototyping of relatively complex algorithms
- ✓ Provides decent performance at minimal programming effort
- ✓ No need to link any external libraries (good portability)

- ✗ Slow compilation (templates!)
- ✗ Everything runs in the default stream — no overlapping computation and memory transfers

Thrust Features (1/5): Memory management

```
std::list<int> h_list;
... // Initialise h_list on the CPU
// Allocate and initialise vector d_vec on the GPU
thrust::device_vector<int> d_vec(h_list.begin(),
                                 h_list.end());
// Transfer data from the GPU to the CPU
 std::vector<int> h_vec(d_vec.size());
 thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
 // Host and device memory is deallocated automatically
```

- STL-like host and device vectors (arrays)
- Easy CPU-GPU memory transfers
- Memory is deallocated automatically

Thrust Features (2/5): Transformations

```
// Allocate three device vectors of length 10
thrust::device_vector<int> X(10), Y(10), Z(10);
// Initialise X to 0, 1, ..., 9
thrust::sequence(X.begin(), X.end());
// Compute Y = -X
thrust::transform(X.begin(), X.end(), Y.begin(),
                                  thrust::negate<int>());
// Fill Z with twos
thrust::fill(Z.begin(), Z.end(), 2);
// Compute Y = X mod 2
thrust::transform(X.begin(), X.end(), Z.begin(),
                        Y.begin(), thrust::modulus<int>());
```

- Fast coefficient-wise vector transformations
- Easy to define your own unary and binary operators

Thrust Features (3/5): Reductions

```
thrust::device_vector<int> X(10);
thrust::sequence(X.begin(), X.end());
// Sum all the elements in vector
int sum = thrust::reduce(X.begin(), X.end(), 0,
                                                thrust::plus<int>()
// Sum is so common that it is the default reduction
sum = thrust::reduce(X.begin(), X.end());
// Same result as above
// Count the ones in a vector
int count = thrust::count(X.begin(), X.end(), 1);
```

- Fast reductions *(these are not easy to implement efficiently!)*
- Easy to define your own binary reduction operators

Thrust Features (4/5): Prefix-sums (Scan)

```
int data[6] = {1, 0, 2, 2, 1, 3};
int res[6];
thrust::exclusive_scan(data, data + 6, res);
// Out-of-place scan; res is now {0, 1, 1, 3, 5, 6}
thrust::inclusive_scan(data, data + 6, data);
// In-place scan; data is now {1, 1, 3, 5, 6, 9}
```

- Scan is a very useful building-block for many algorithms
- Thrust supports *in-place* and *out-of-place* scans
- Easy to define you own binary reduction operators
- Can perform transformation before scan efficiently

Thrust Features (5/5): Data reordering

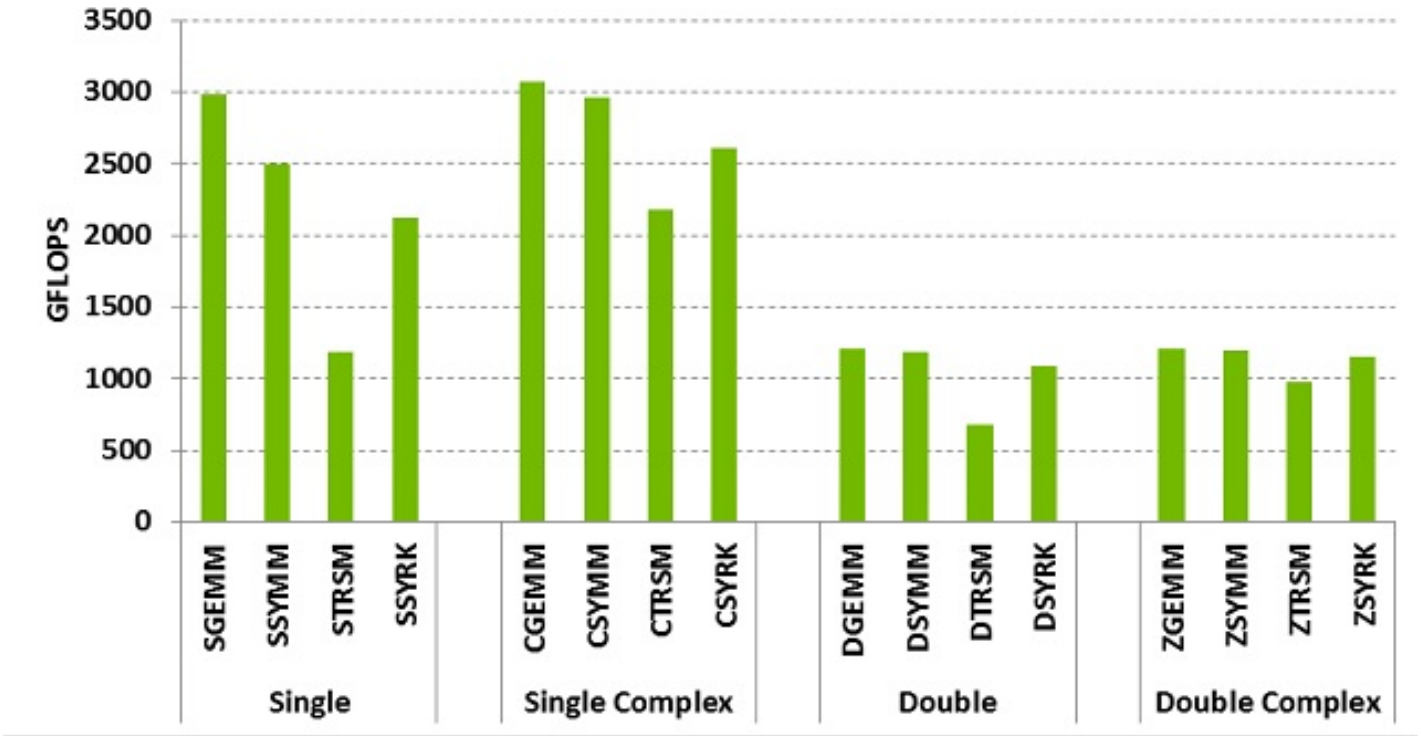
```
// Generate 32M random numbers on the host
thrust::host_vector<int> h_vec(32 * 1024 * 1024);
thrust::generate(h_vec.begin(), h_vec.end(), rand);
// Transfer data to the device
thrust::device_vector<int> d_vec(h_vec);
// Sort data on the device
thrust::sort(d_vec.begin(), d_vec.end());
```

- Very fast sorting (846M keys per second on GTX 480)
- Supports sorting key-value pairs and stable sort
- Other reorderings and stream compaction available:

partition	reorder according to a predicate (e.g. Quick-sort)
copy_if	copy only elements that satisfy a condition
remove_if	remove elements that fail a predicate
unique	remove consecutive duplicates within a sequence

Basic Linear Algebra Subprograms

- ✓ Highly optimised; runs fast on a wide range of GPU cards
- ✓ Supports single and double precision, and complex numbers
- ✗ Low-level data management must be handled manually



Performance with K40, m=n=k=4096, CUDA 7.0

What are the BLAS?

Dense linear algebra software: major success story!

- BLAS:
  - Level 1 BLAS $y \leftarrow y + \alpha x$ $O(n)$ operations, $O(n)$ data; Lawson et al., 1979
  - Level 2 BLAS $y \leftarrow y + Ax$ $O(n^2)$ operations, $O(n^2)$ data; Dongarra et al., 1988
  - Level 3 BLAS $C \leftarrow C + AB$ $O(n^3)$ operations, $O(n^2)$ data; Dongarra et al., 1990
- LAPACK
- BLACS
- ScaLAPACK

History: J. Dongarra & D. Walker in SIAM Review, June 1995.

BLAS: Basic Linear Algebra Subroutines

Nomenclature scheme (compatible with Fortran 77 naming rules):

- First letter: data type
  - S: Single precision real
  - D: Double precision real
  - C: Single precision complex
  - Z: Double precisione complex
- Second/third letter: matrix kind (level 2 & 3)
  - GE: GEneral rectangular;
  - GB: General Banded;
  - TR: TRiangular;
  - TB: Triangular Banded;
  - SY: SYmmetric;
  - HE: HErmitian;

BLAS: Basic Linear Algebra Subroutines

Nomenclature scheme (compatible with Fortran 77 naming rules):

- From fourth letter: operator
  - MV: Matrix-Vector product $y \leftarrow \alpha Ax + \beta y$
  - SV: Triangular System Solve (single Vector) $x \leftarrow A^{-1}x$
  - MM: Matrix-Matrix Product $C \leftarrow \alpha AB + \beta C$
  - SM: Triangular System Solve (Multiple Vectors) $B \leftarrow \alpha A^{-1}B$
  - RK: Rank K update $C \leftarrow \alpha AA^T + \beta C$
  - R2K: Rank 2K update $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$

BLAS: Basic Linear Algebra Subroutines

Example 1: `_GEMV`

$$y \leftarrow \alpha A x + \beta y \qquad y \leftarrow \alpha A^T x + \beta y$$

```
_GEMV(  TRANS,  M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
```

TRANS: Choose Ax or $A^T x$;

M,N: Dimensions of (the part of) matrix A;

ALPHA, BETA: scalars α, β;

A: The data structutre (array) containing A;

LDA: Leading dimension

X,Y: Arrays containing x, y;

INCX, INCY: Stride between elements of x e y

Note: many of the arguments are needed to avoid copies of variables (major performance problem with uncautious use of Matlab).

Example 2: `_GEMM`

$$C \leftarrow \alpha op(A) op(B) + \beta C \qquad op(X) = X, X^T, X^H$$

```
_GEMM(  TRANSA, TRANSB,  M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)
```

TRANSA, TRANSB: Choose $op(X)$;

M,N: Dimensions of (part of) matrix result C;

K: Third dimension of the product;

ALPHA, BETA: scalars α, β;

A,B,C: Fortran Array containing matrices A, B, C;

LDA,LDB,LDC: Leading dimensions

Note: the transpose (where applied) is NOT performed explicitly; the result is computed "as if" the matrix was transposed.

CUBLAS

```
cublasHandle_t cublas;
float alpha = 2.0f;
float *d_x, *d_y;
... // Initialise CUBLAS context and GPU vectors
cublasSaxpy(cublas, N, &alpha, d_x, 1, d_y, 1);
cublasSdot(cublas, N, d_x, 1, d_y, 1, &alpha);
cublasSnrm2(cublas, N, d_x, 1, &alpha);
cublasSscal(cublas, N, &alpha, d_x, 1);
// Matrix-vector multiplication (symmetric, double)
cublasDsymv(...);
// Triangular system solve (general, complex single)
cublasCtrsv(...);
// Matrix-matrix multiplication (symmetric, single)
cublasSsymm(...);
// Triangular system solve (general, multiple rhs)
cublasDtrsm(...);
```

BLAS-like extensions

```
// Matrix-matrix addition/transposition
cublasDgeam(...);
// Matrix-matrix multiplication (multiple matrices)
cublasDgemmBatched(...);
// LU factorisation with partial pivoting
cublasDgetrfBatched(...);
// Multiple small matrices inversion
cublasDgetriBatched(...);
```

GPU implementation of sparse matrix algebra

- ✓ Runs fast on a wide range of GPU cards
- ✓ Supports single and double precision, and complex numbers
- ✓ Supports different sparse formats: Coordinate, Compressed (CSR, CSC), Ellpack, Hybrid (ELL+COO), Block Compressed
- ✓ Supports sparse and dense vectors
- ✓ Compatible with CUBLAS
- ✗ Low-level data management must be handled manually
- ✗ Low performance in some specialised routines! (e.g. symmetric sparse matrix-vector multiplication, sparse triangular system solve)

Sparse matrices

A matrix is sparse when there are so many zeros that it pays off to take advantage of them in the computer representation

I.e. nonzero coefficients are $O(n)$ instead of n^2.

- Ubiquitous in scientific computing;
- Problematic implementation;
- No native support in programming languages;
- Internal representation efficiency vs. usability;

Major software development effort.
How do we solve

$$Ax = b$$

when A is sparse?

Classical (stationary) Iterative Solvers

Search for a solution to $Ax = b$ with:

$$x_{k+1} = Bx_k + f$$

Jacobi version

$$A = L + D + U \Rightarrow B = -D^{-1}(L + U)$$

needs vector-vector and sparse matrix-vector multiplication

Gauss-Seidel version:

$$A = L + D + U \Rightarrow B = -(L + D)^{-1}U$$

needs sparse matrix-vector multiplication and sparse triangular system solve

Krylov subspace methods

Search for a solution to $Ax = b$ by projection:

$$\begin{aligned} x_m &\in \{x_0 + \mathcal{K}_m\} \\ b - Ax_m &\perp \mathcal{L}_m \end{aligned}$$

central role: Kyrlov subspace

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}$$

The ancestor of them all is the Conjugate Gradient (Hestenes and Stiefel 1952).
Thus: need to implement the product of a sparse matrix by a dense vector.

$$y \leftarrow \alpha Ax + \beta y$$

Why matrix-vector product performs poorly

- Indirect addressing
- Low ratio between floating point operations and memory accesses
- High consumption of the CPU-memory bandwidth (indices have to be explicitly read)
- Low spatial or temporal locality:
  - The elements of the matrix are accessed sequentially and are not reused
  - The elements of the destination vector are accessed sequentially and are reused for each element in the corresponding row of the matrix
  - The elements of the source vector are not accessed sequentially and are not necessarily reused

Can we reach peak FLOPS performance with SpMV?

Forget about it. But you can aim for (close to peak) bandwidth!

Why matrix-vector product performs poorly

Memory bound kernel

More reasonable "peak performace": execute $2 \cdot NNZ$ operations when reading NNZ doubles+indices, or

$$P = \frac{2NNZ}{\frac{NNZ \cdot 12}{BW}} = \frac{BW}{6}$$

For the K40 the absolute "roofline" is $288/6 = 48$ GFLOPS.
Even better, use *measured sustained bandwidth* and get $190/6 = 32$ GFLOPS.

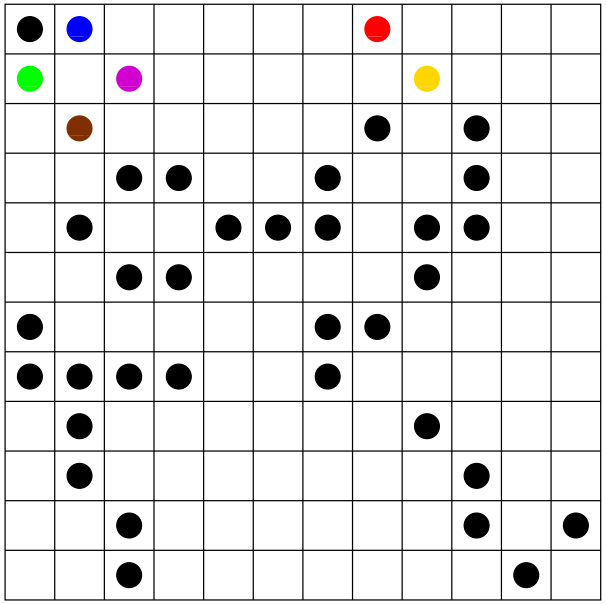
Storage schemes for sparse matrices

COOrdinate storage:

M	Rows;
N	Columns;
NZ	Non zeroes;
IA(1:NZ)	Row indices;
JA(1:NZ)	Column indices;
AS(1:NZ)	Coefficients;

Note: by definition of number of rows we have $1 \leq IA(i) \leq M$, likewise for the columns.

Storage schemes: COO



Elements Array	●	●	●	●	●	●	●	●	...
Col idx array	1	2	8	1	3	9	2	8	...
Row idx array	1	1	1	2	2	2	3	3	...

```
for i=1:nz
   ir = ia(i)
   jc = ja(i)
   y(ir) = y(ir) + as(i)*x(jc)
end
```

Cost: 5 memory reads, 1 write and 2 flops per iteration.

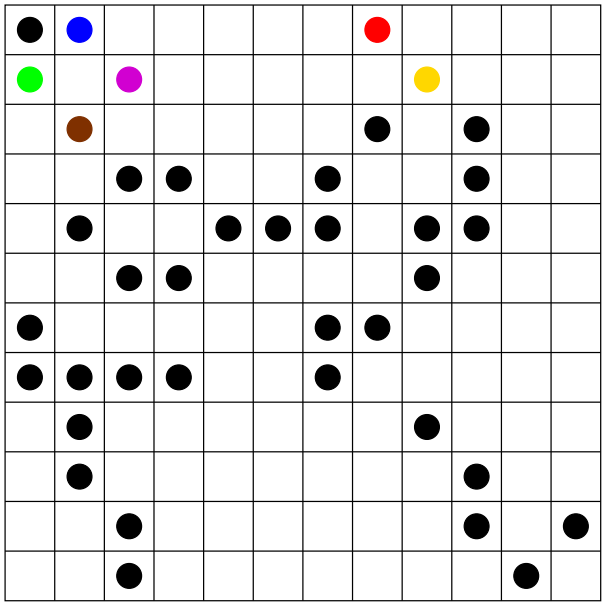
Storage schemes for sparse matrices

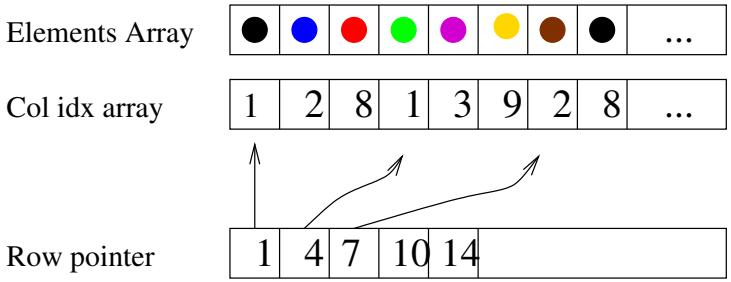
Compressed Storage by Rows:

M	Rows;
N	Columns;
IA(1:M+1)	Pointers to row start;
JA(1:NZ)	Column indices;
AS(1:NZ)	Coefficients;

Storage schemes

CSR






```
for  i=1:m
   for  j=ia(i):ia(i+1)-1
        y(i) = y(i) + as(j)*x(ja(j))
   end
end
```

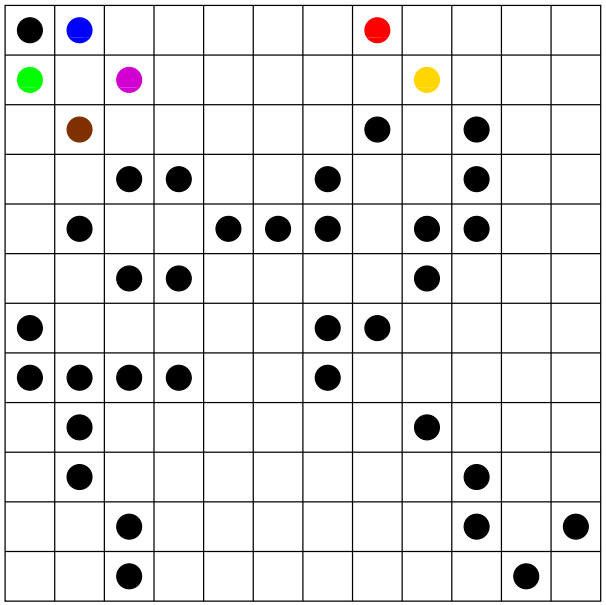
Cost: 3 memory read and 1 write per outer iteration, 3 memory read and 2 flops per inner iteration.

Storage schemes for sparse matrices

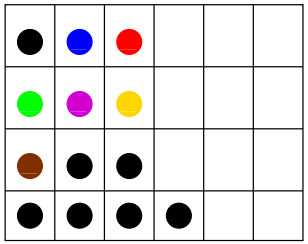
ELLPACK

M	Rows;
N	Columns;
MAXNZ	Max nonzeros per row
JA(1:M,1:MAXNZ)	Column indices
AS(1:M,1:MAXNZ)	Coefficients;

The ELLPACK (ELL) storage format



Elements Array



Col idx array

1	2	8			
1	3	9			
2	8	10			

```
for  i=1:m
   for  j=1:maxnz
      y(i) = y(i) +  as(i,j)*x(ja(i,j))
   end
end
```

Cost: 1 memory read and 1 write per outer iteration, 3 memory read and 2 flops per inner iteration (also, regular access pattern).

Storage Schemes for sparse matrices

And many more

Diagonals, Jagged diagonals, Block CSR, Variable Block Rows, Compressed Storage by Columns, ...

To make sense of them, see Filippone et al., ACM TOMS, 2017, where 67 different formats are reviewed: that's just for NVIDIA cards, just in the last 5 years.

Facts

- Different computer architectures are best used by different formats;
- Different formats are best for different operations (we need them all);

Requirements (put your library developer's hat on):

- We want to be able to change in response to machine changes (might possibly be done at compile time, but annoying);
- We want to be able to change in response to usage requirements (need to change at run time)
- We need to switch among formats, some of them unkonwn at compile time;

We want maximum freedom/flexibility (i.e. we like to have our cake and eat it too)

Use Design Patterns

- STATE;
- BUILDER;
- MEDIATOR;
- PROTOTYPE.

See: A. Buttari, S. Filippone, ACM TOMS, 2012,
V. Cardellini, S. Filippone and D. Rouson, Scientific Programming, 2014

GPU Storage Format: ELL-G

```
Dspmvm_gpu_krn (double *y, double alpha, double* cM, int* rP,int* rS,
                int n, int pitch, double *x, double beta, int firstIndex)
{
  int i = threadIdx.x + blockIdx.x * (THREAD_BLOCK);
  if (i >= n)
    return;
  double y_prod = 0.0;
  int row_size = rS[i];

  rP += i;
  cM += i;
  for (int j = 0; j < row_size; j++)    {
    int pointer = rP[0] - firstIndex;
    double value = cM[0];
    rP += pitch;
    cM += pitch;
    y_prod += __dmul_rn (value, x[pointer]);
  }
  if (beta == 0.0)
    y[i] = (alpha * y_prod);
  else
    y[i] = __dmul_rn (beta, y[i]) + __dmul_rn (alpha, y_prod);
}
```

Performance issues

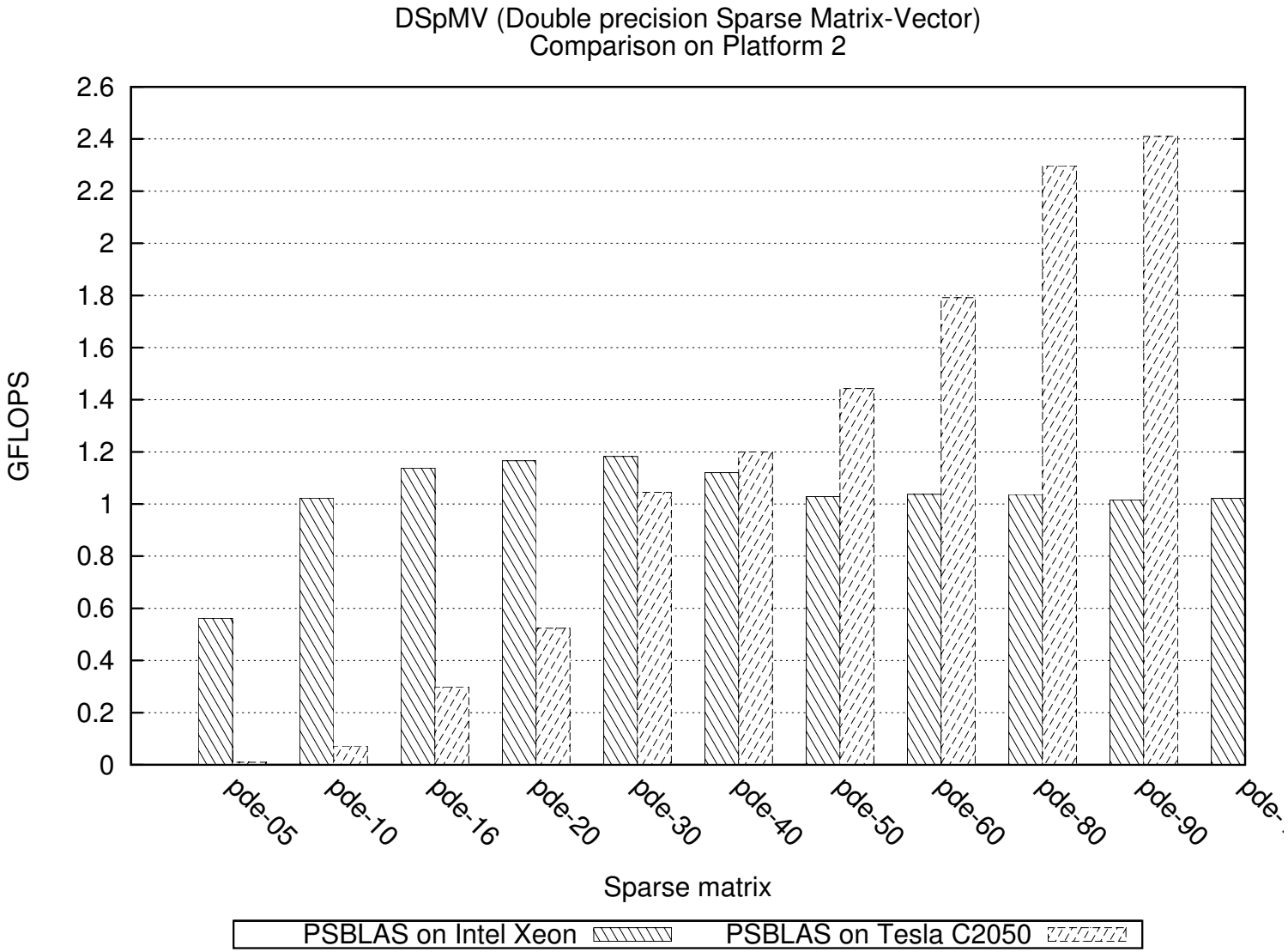
- The matrix is populated once, used many times
  $\Rightarrow$ if your matrix changes often you should update it on the *device* side;
- The vectors are updated many times
  $\Rightarrow$ Keep the updated values and the computations inside the GPU and pull them out only when needed;
- GPU memory should work as a "black hole"

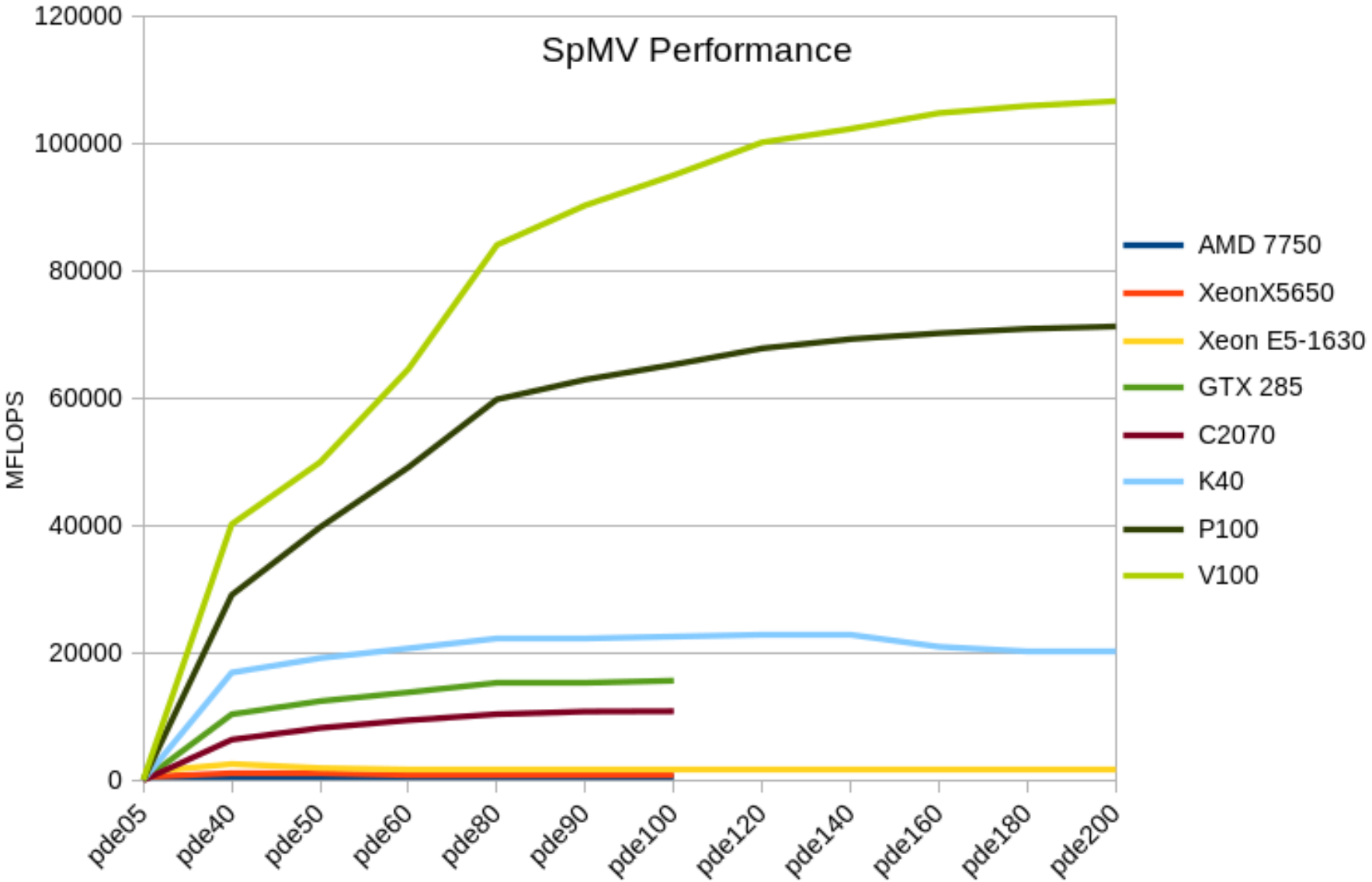
Performance data

Well, we first begin with *human* performance data, i.e. development time:

- Had to develop the CUDA kernels (easy: grad students are cheap. Hmm: *good* grad student may be cheap but not easy to find ...)
- No need to change the main framework;
- Test programs only needed a new MOLD variable to be declared;
- Needed to write the F2003 wrapper around CUDA code, adapting the existing ELL code: about a day;
- Glueing the CUDA code and running the initial tests took about half a day;
- Repeating the wrapping process to interface the NVIDIA CUSPARSE library took all of an additional day.

What happens if you move the vector data?

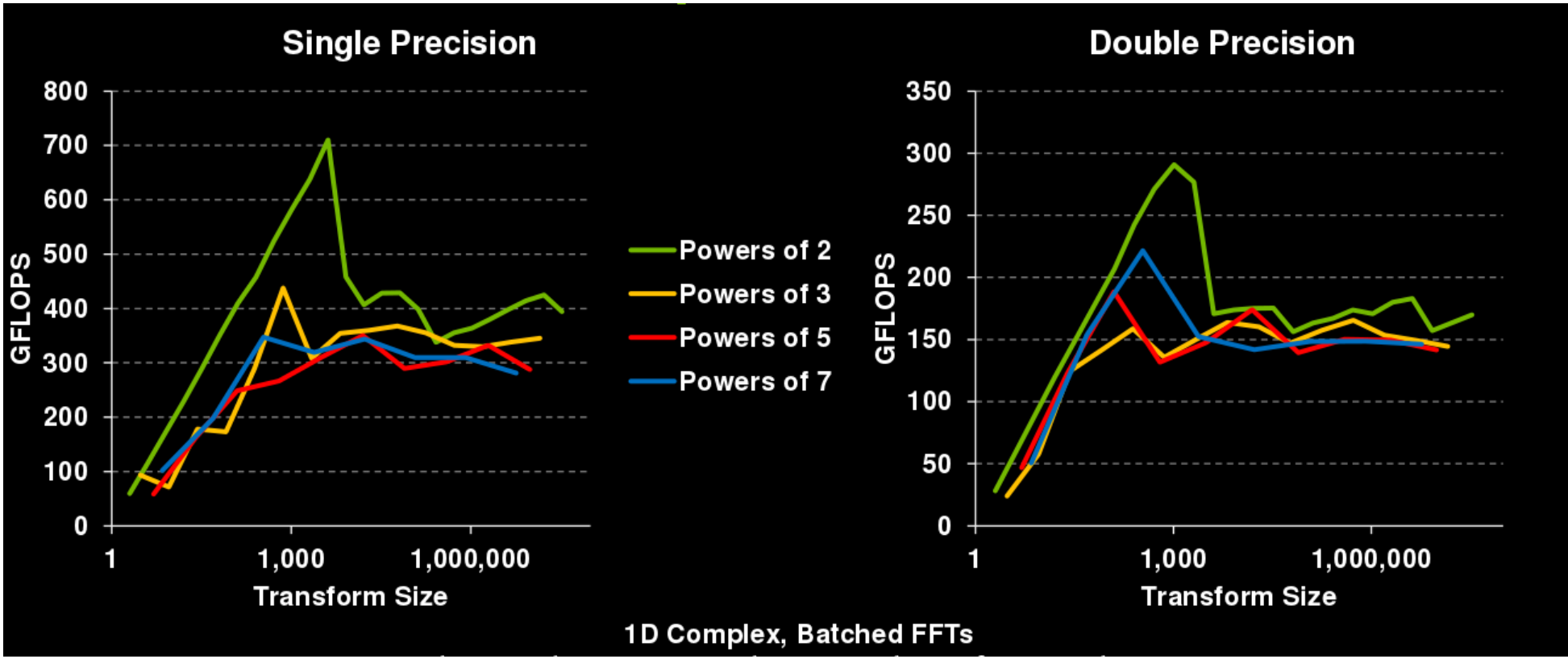



SpMV Performance
MFLOPS
120000
100000
80000
60000
40000
20000
0
pde05
pde40
pde50
pde60
pde80
pde90
pde100
pde120
pde140
pde160
pde180
pde200
AMD 7750
XeonX5650
Xeon E5-1630
GTX 285
C2070
K40
P100
V100

GPU implementation of Fast Fourier Transform

- ✓ Highly optimised; runs fast on a wide range of GPU cards
- ✓ Interface based on FFTW3 (fast CPU library for FFT)
- ✓ Supports single and double precision
- ✓ Supports one-, two- and three-dimensional transforms
- ✓ Supports R2C (real-to-complex), C2C, C2R transforms
- ✓ Can perform multiple transforms at the same time
- ✗ Low-level data management must be handled manually
- ✗ No support for real-to-real transforms (DST, DCT)
- ✗ Performance highly dependant on the problem size

CUFFT


Single Precision
GFLOPS
800
700
600
500
400
300
200
100
0
1
1,000
1,000,000
Transform Size
Powers of 2
Powers of 3
Powers of 5
Powers of 7
Double Precision
GFLOPS
350
300
250
200
150
100
50
0
1
1,000
1,000,000
Transform Size
1D Complex, Batched FFTs

Other GPU Libraries

CURAND High-quality random numbers generation on the GPU Useful in stochastic algorithms, e.g. Monte-Carlo simulations

NPP (Nvidia Performance Primitives) Image, video and signal processing functions

MAGMA Highly optimised dense algebra library Interface based on BLAS/LAPACK routines

CUSP High-level sparse matrix and graph computation library Provides variety of iterative methods: Conjugate Gradients (CG), BiCG, BiCGStab, GMRES etc. Provides variety of preconditioners: Diagonal, Algebraic Multigrid (AMG), Approximate Inverse

Case study:Particle Algorithms

Spatial Partitioning Algorithms

Many physical phenomena display spatial locality properties.

- Collisions among N discrete objects: need $N(N-1)$ comparisons to keep track of all possible interactions;
- Propagation of waves/wavefronts in a domain with obstacles.

Spatial Partitioning approach:

- Decompose spatial domain, often recursively/hierarchically;
- Interaction limited to nearest neighbours;

Implementations:

- Uniform grids
- Octrees: dense and sparse

Uniform grids

Space is divided into fixed subdomains (cubes in 3D);

- AABB: Axis Aligned Bounding Box (of the objects under consideration) determines the grid size;
- Tight grid variant: associate an element to all the cells it touches;
- Loose grid variant: associate an element with only one of the cells, then keep track of the others.

On the GPU building the grid is done at each time step, since it is very cheap

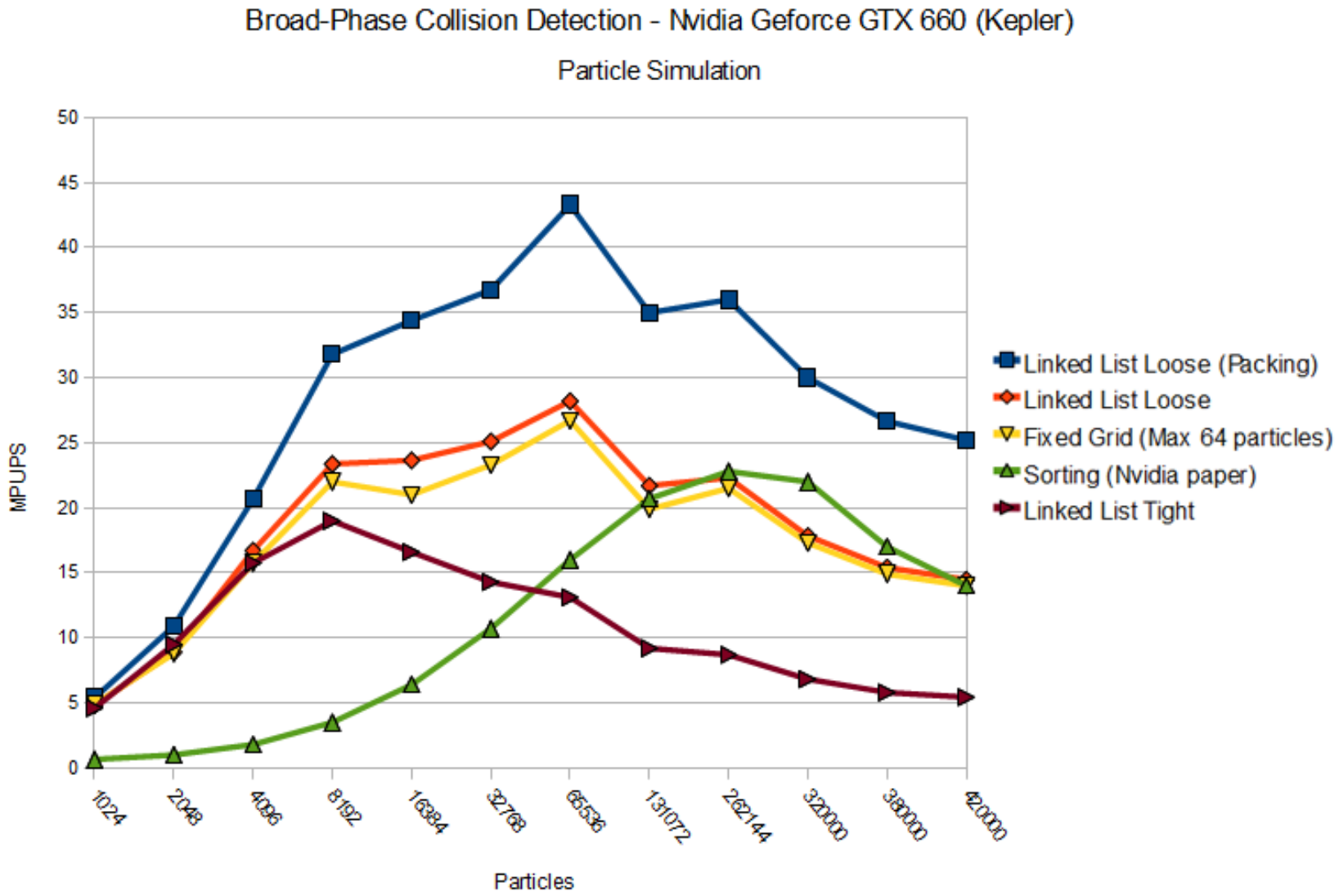
Occupancy lists

Atomics: Static allocation of elements; Vector of counters of elements per cell, updated with `atomicAdd()` and used as displacements. Same algorithmic pattern as conversion among sparse formats (COO to ELL): *Parallel Prefix Sum*;

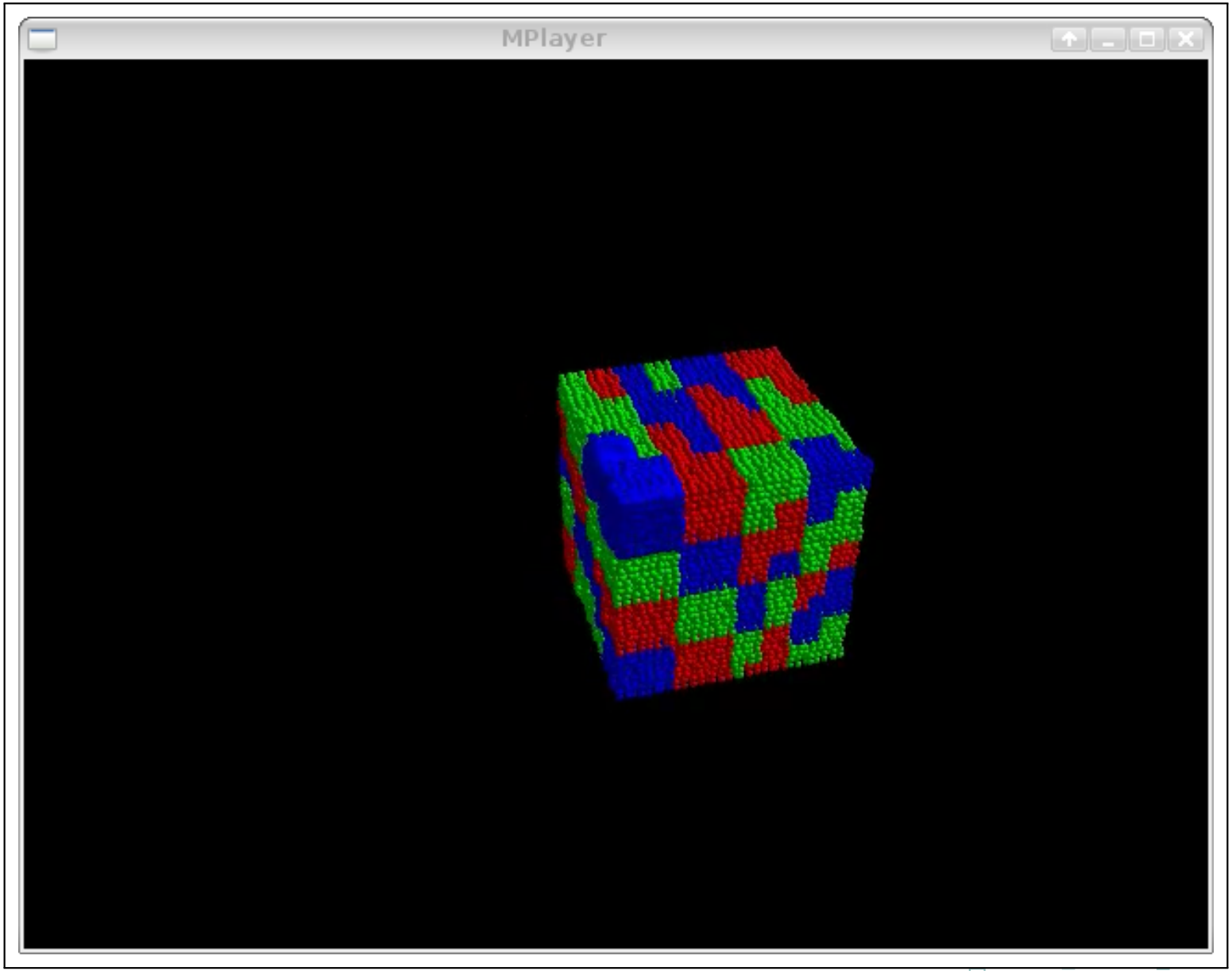
Sorting Build a list of pairs (cell,item), sort them with Radix Sort; elements in the same cell are close, cost $N \times (\log(MaxCellID))$

Linked Lists Features: Two arrays: `CellHead` and `CellNext` with indices into the list of cells, updates with `atomicExch`, just single pass, Perfectly coalesced memory accesses.

The algorithm uses the lists to exclude very far elements.



Collision Detection Performance



Krylov subspace methods

Search for a solution to $Ax = b$ by projection:

$$\begin{aligned} x_m &\in \{x_0 + \mathcal{K}_m\} \\ b - Ax_m &\perp \mathcal{L}_m \end{aligned}$$

central role: Kyrlov subspace

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}$$

The ancestor of them all is the Conjugate Gradients Method (Hestenes and Stiefel 1952); to implement it, we need the product of a sparse matrix by a (dense) vector:

$$y \leftarrow \alpha Ax + \beta y$$

Preconditioning for Krylov Methods

Choose a transformation M

$$Ax = b \Leftrightarrow M^{-1}Ax = M^{-1}b,$$

Total iteration time:

$$T_{\text{tot}} = T_{\text{setup}} + N_{\text{it}} \times T_{\text{it}}$$

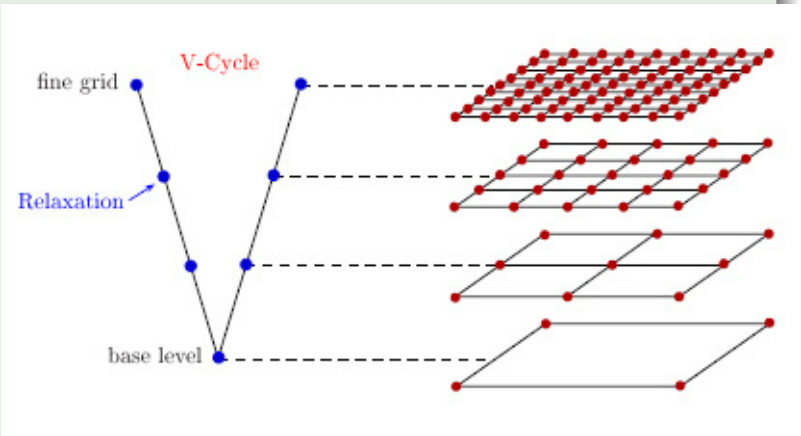
We would like:

- M easy (fast) to determine $\Rightarrow$ T_{setup} small;
- M^{-1} easily applied $\Rightarrow$ T_{it} small;
- $||M^{-1}A - I||$, $||M^{-1}||$ and $||M - A||$ small in some norm $\Rightarrow$ N_{it} small.

It is impossible to satisfy all of them perfectly!
So, we need to find good tradeoffs.

Example: symmetric V-cycle

procedure V-cycle$\left(k, nlev, A^k, b^k, x^k\right)$
  if $(k \neq nlev)$ then
    $x^k = x^k + (M^k)^{-1}\left(b^k - A^k x^k\right)$
    $b^{k+1} = \left(P^{k+1}\right)^T \left(b^k - A^k x^k\right)$
    $x^{k+1} = \text{V-cycle}\left(k+1, A^{k+1}, b^{k+1}, 0\right)$
    $x^k = x^k + P^{k+1} x^{k+1}$
    $x^k = x^k + (M^k)^{-T}\left(b^k - A^k x^k\right)$
  else
    $x^k = \left(A^k\right)^{-1} b^k$
  endif
  return x^k
end



AMG methods do not explicitly use the problem geometry and rely only on matrix entries to generate coarse grids (setup phase)

D. Bertaccini, S. Filippone: CAMWA 2016.
D'Ambra, Filippone and Vassilevski, ACM TOMS, 2018.

Putting it all together: PARFLOW

Code "Parflow" from Juelich Supercomputing Centre: modelling the flow of water through anisotropic porous media (Horizon 2020 EoCoE Project, Grant 676629)

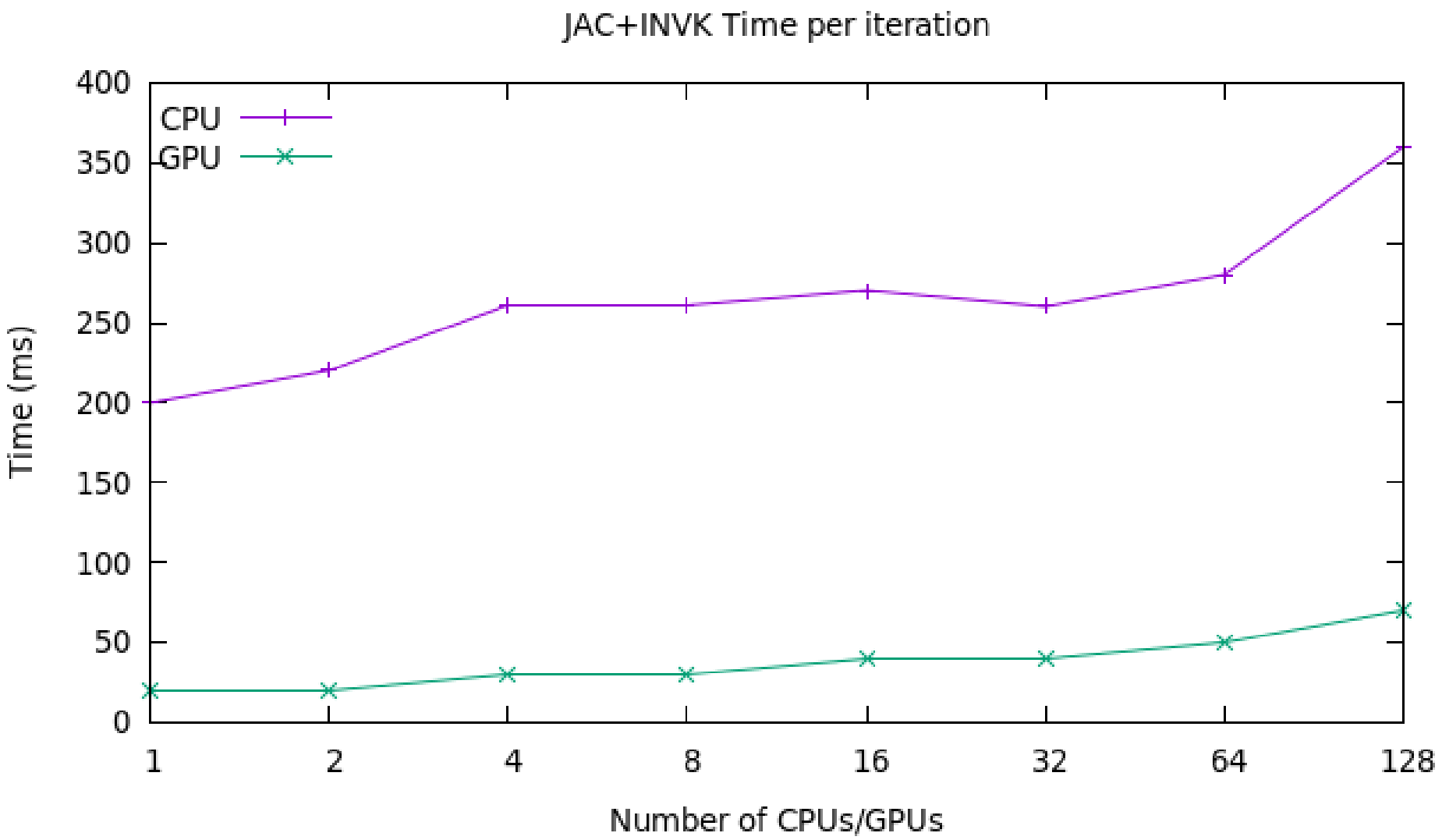
Simplified steady-state model

$$-\nabla \cdot \mathbf{K} \nabla p = f$$

on unit cube, with no-flow boundary conditions

Cell-centered finite volumes, random anisotropic conductivity tensor with lognormal distribution, cartesian grid with uniform refinement.
Solver: CG with V-cycle, Jacobi smoother at lower level(s), coarse solver block Jacobi with INVK(0,1) or INVK(1,2).

Experiments performed on JURECA at Jülich Supercomputing Center, 128 NVIDIA Tesla K80 GPUs.
Abdullahi-Hassan et al, ParProcLetters, 2019



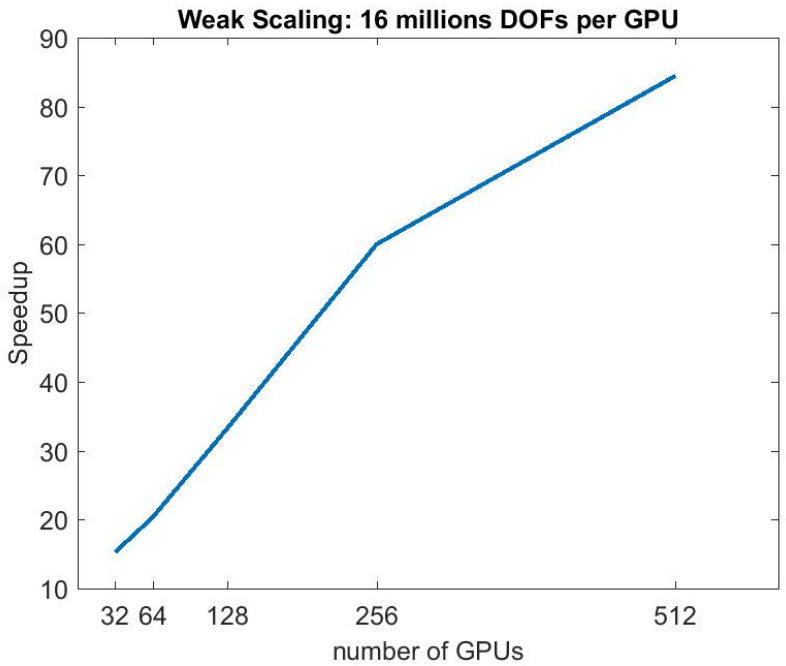
Weak scalability results, from 2 to 256 millions equations; time per

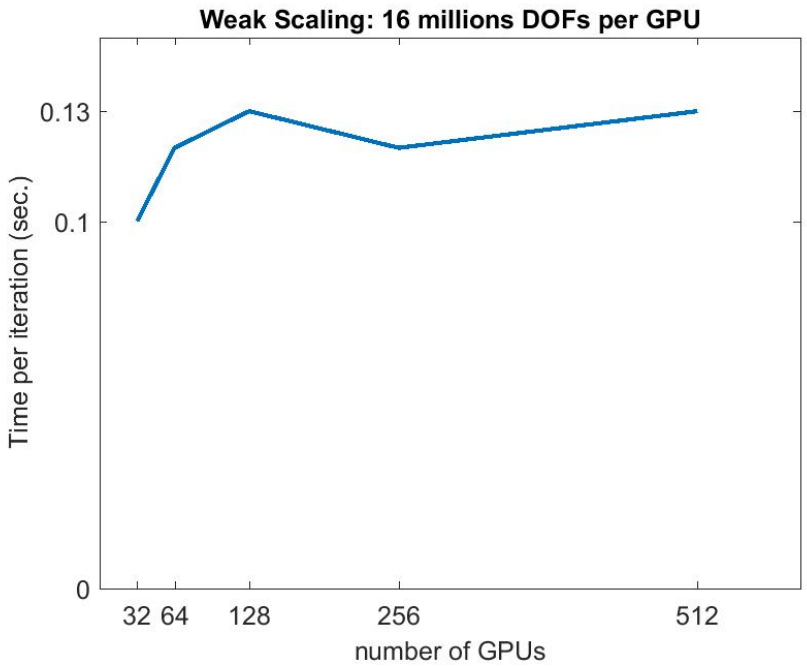
JACOBI+INVK, from 1 to 128 GPUs, weak scaling

				CPU			GPU	
	N	Nlv	Its	Tsetup (s)	Tsolve (s)	T/iter (ms)	Tsolve (s)	T/iter (ms)
1	2M	4	19	3.13	3.76	200	0.40	20
2	4M	4	21	4.06	4.57	220	0.52	20
4	8M	4	20	4.65	5.13	260	0.63	30
8	16M	4	25	5.20	6.58	260	0.85	30
16	32M	4	27	5.39	7.21	270	1.02	40
32	64M	4	31	5.35	8.21	260	1.24	40
64	128M	4	37	6.25	10.20	280	1.71	50
128	256M	5	31	7.19	11.16	360	2.23	70
128	256M	4	46	7.00	14.05	305	2.04	44

- Time per iteration proportional to number of levels (for sufficiently large parallelsm);
- Number of iterations remains within a factor 2 when size grows by a factor of 128;

JACOBI+INVK, from 32 to 512 Pascal P100 GPUs on Piz-Daint (CSCS Swiss Supercomputing Centre, Lugano), weak scaling 16M Eqn/GPU, up to 8×10^9 total eqn.





Case study: Biomedical Image Segmentation

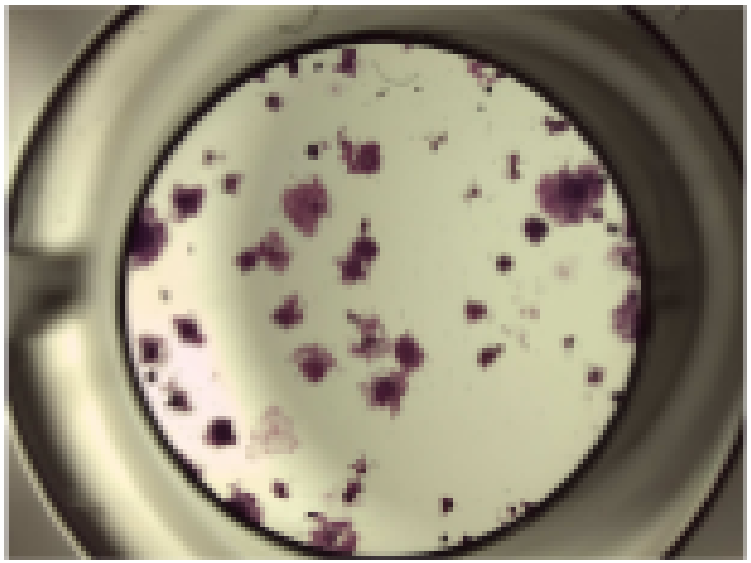
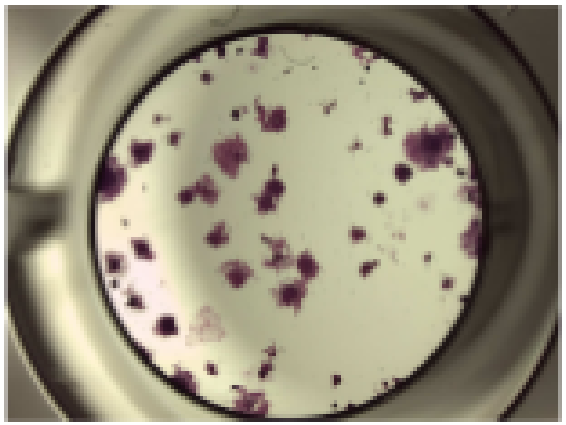


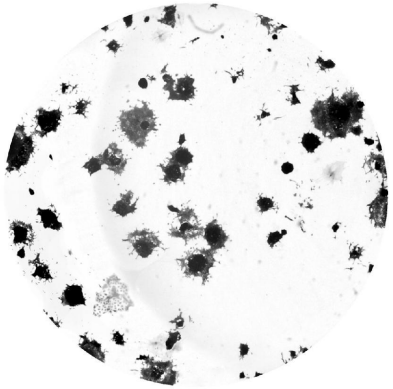
Image Analysis: A Biomedical Application

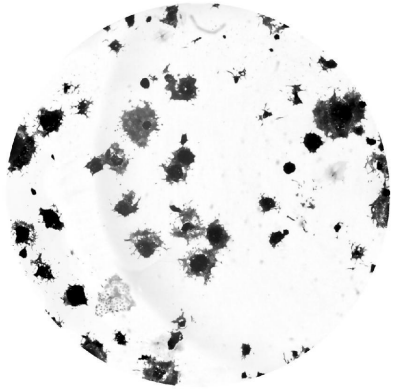
Problem: automatically discriminating domed and flat colonies in microscopy images.

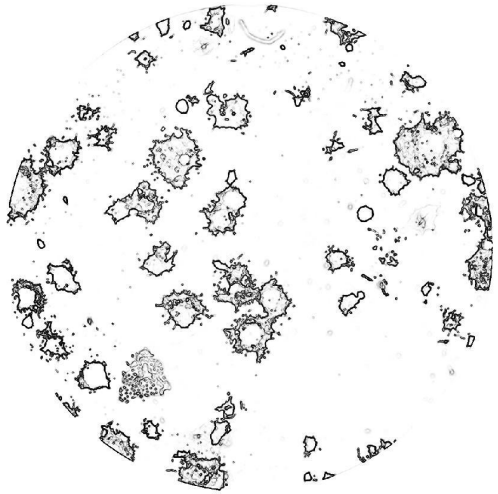
1. Pre-processing, (cleanup imperfections in the image acquisition process);
2. Segmentation, to separate the cell colonies in each well image;
3. Feature computation, to provide numerical descriptors of each segmented colony;
4. Classification, to provide the discrimination and quantification of various colonies

Image Analysis: A Biomedical Application









P. D'Ambra
and S. Filippone: A Parallel Generalized Relaxation Method for High-Performance Image Segmentation on GPUs JCAM, 2016

Image Segmentation: the problem

Mathematical formulation (Mumford–Shah):

Given an image f defined on a Lipschitz bounded domain Ω, find a piecewise smooth approximation u, a boundary set K (the edge set) and a complete partition of Ω, each connected component Ω_i corresponds to a single physical object

$$\Omega = \Omega_1 \cup \Omega_2 \cup \cdots \cup \Omega_i \cup K, \quad K = \Omega \cap (\partial\Omega_1 \cap \partial\Omega_2 \cap \cdots \cap \partial\Omega_i)$$

by minimizing the functional

$$E(u, K) = \mu^2 \int_\Omega (u - f)^2 dx + \int_{\Omega \setminus K} |\nabla u|^2 dx + \nu |K| \qquad (1)$$

Dropping any term creates trivial solutions; however $|K|$ makes it difficult to prove that a solution exists.

Ambrosio–Tortorelli approximation:

Represent $K \subset \Omega$ by a function z.

Let $\Omega \subset \mathbb{R}^2$ a Lipschitz bounded open set and $f \in L^\infty(\Omega)$ the image; then we minimize the functional:

$$E_\epsilon(u,z) = \int_\Omega (u-f)^2 dxdy + \beta \int_\Omega z^2 |\nabla u|^2 dxdy + \alpha \int_\Omega \left(\epsilon |\nabla z|^2 + \frac{(z-1)^2}{4\epsilon} \right) dxdy, \quad (2)$$

where $u \in C^1(\Omega \setminus K)$, and z, $0 \leq z \leq 1$, is a function which controls $|\nabla u|$ such that

$$z(x,y) \approx 0 \text{ if } (x,y) \in K$$

and $z(x,y) \approx 1$ if (x,y) in a smooth region.

Image Segmentation: the problem

Parameters:

- β: controls the smoothness of u outside the edge set K (large $\beta \Rightarrow$ small gradients outside K);
- α: controls the length of the edge set, large $\alpha \Rightarrow$ less jumps in the recovered image function.
- ϵ: obtain a sequence of elliptic functionals that for $\epsilon \to 0$, are $\Gamma-$convergent to the original MS functional

A good choice for ϵ is critical (together with non–convexity); usually good choice $h/\epsilon < 1$,

Image Segmentation: the solution strategy

Second–order finite–difference discretization of Euler–Lagrange (natural boundary conditions):

$$\begin{cases} 2(u-f) - 2\beta\nabla\cdot(z^2\nabla u) = 0 \\ 2\beta z|\nabla u|^2 - 2\alpha\epsilon\nabla^2 z + \frac{\alpha}{2\epsilon}(z-1) = 0 \end{cases} \quad (x,y)\in\Omega, \tag{3}$$

System of coupled elliptic equations; the discretized form can be rewritten as:

$$\begin{bmatrix} A(\mathbf{z}) & \mathbf{0} \\ \mathbf{0} & B(\mathbf{u}) \end{bmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix}. \tag{4}$$

Both $A(\mathbf{z})$ and $B(\mathbf{u})$ are well conditioned M–matrices.

Image Segmentation: the solution strategy

Generalized Gauss–Seidel relaxation with inner linear iterations:

1: $k \leftarrow 0$, $\mathbf{z}^0 \leftarrow \mathbf{1}$ (all–ones vector) and $\mathbf{u}^0 \leftarrow f_h$ (original image)
2: Build R.H.S. $\mathbf{f}_1$ and $\mathbf{f}_2$
3: **repeat**
4: Build matrix $A(\mathbf{z}^k)$
5: Solve $A(\mathbf{z}^k)\mathbf{u} = \mathbf{f}_1$ for $\mathbf{u}^{k+1}$ starting from $\mathbf{u}^k$
6: Build matrix $B(\mathbf{u}^{k+1})$
7: Solve $B(\mathbf{u}^{k+1})\mathbf{z} = \mathbf{f}_2$ for $\mathbf{z}^{k+1}$ starting from $\mathbf{z}^k$
8: $k = k + 1$
9: **until** convergence

Ref: P. D'Ambra, G. Tartaglione, Solution of Ambrosio-Tortorelli Model for Image Segmentation by Generalized Relaxation Method, Communications in Nonlinear Science and Numerical Simulation , vol. 20, 2015.

Image Segmentation: the Jacobi Solver

```
itx = 0
do
  call psb_geaxpby(done,bv,dzero,xit,desc,info)
  call psb_spmm(-done,a_nd,xv,done,xit,desc,info)
  call xit%mlt(ainvd,info)
  nrmden = psb_genrmi(xit,desc,info)
  call psb_geaxpby(done,xit,-done,xv,desc,info)
  nrmdiff    = psb_genrmi(xv,desc,info)
  itx = itx + 1
  if ((nrmdiff <= tol*nrmden).or.(itx > maxit)) then
    call psb_geaxpby(done,xit,dzero,xv,desc,info)
    exit
  end if

  call psb_geaxpby(done,bv,dzero,xv,desc,info)
  call psb_spmm(-done,a_nd,xit,done,xv,desc,info)
  call xv%mlt(ainvd,info)
  nrmden = psb_genrmi(xv,desc,info)
  call psb_geaxpby(done,xv,-done,xit,desc,info)
  nrmdiff    = psb_genrmi(xit,desc,info)
  itx = itx + 1
  if ((nrmdiff <= tol*nrmden).or.(itx > maxit))  exit
end do
```

Image Segmentation: Building the Matrices

1. "Walk" through the discretization mesh and build the coefficients for $A(\mathbf{z})$ and $B(\mathbf{u})$;
2. Pass these coefficients to the library to store them into the matrix data structure.
3. After initialization, only a small subset of the matrix coefficients change at each iteration: do updates, not complete regeneration.

Very heavy part of the application, should be implemented on the GPU, avoid data traffic between host and device.

- The generation of the updates must be performed in the GPU;
- The updates to the sparse matrix structure must work with both the matrix and the coefficient updates already in the GPU device memory;

Also: need to generate the updates for *blocks* of rows, to reduce call overhead and have good device utilization.

Image Segmentation: Building the Matrices

How to size the thread blocks?

- Each thread responsible for the updates of one row.
- Number of rows in a single invocation must be enough to keep all the multiprocessors busy;
- Each multiprocessor should have a number of thread blocks available,
- Block size of 256 is adequate;
- Total number of rows in an update (=threads):

$$5 \text{ blocks/SM} \times 13 \text{ SM on K20} \times 256 = 16640$$

Image Segmentation Performance — CPU

Computing platform: Intel Sandy Bridge E5–2670 / NVIDIA K20
Image sizes $nx = ny = 256, 512, 1024, 2000, 2500, 3000$.
Data Partition — CPU

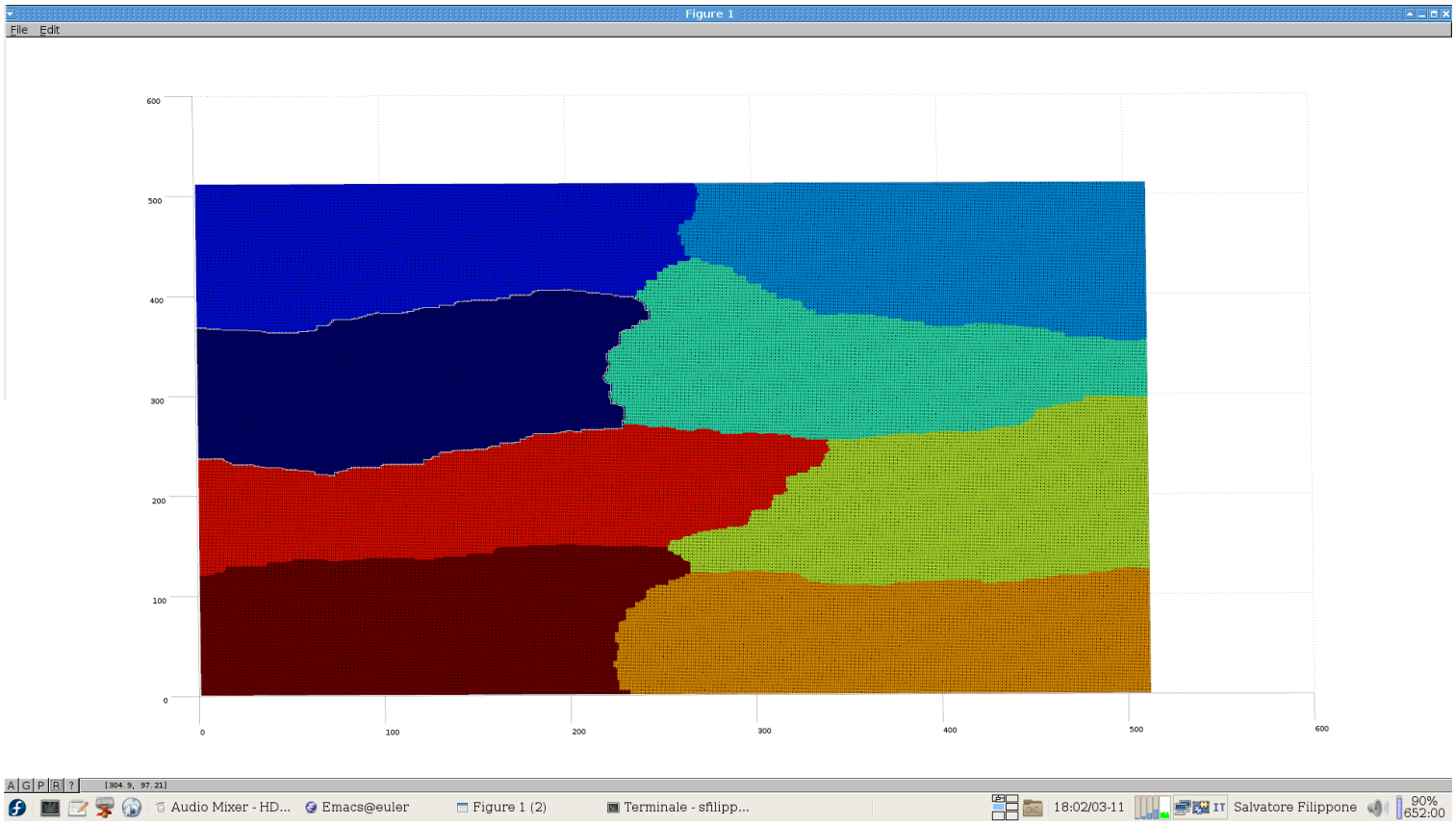


Image Segmentation Performance — GPU

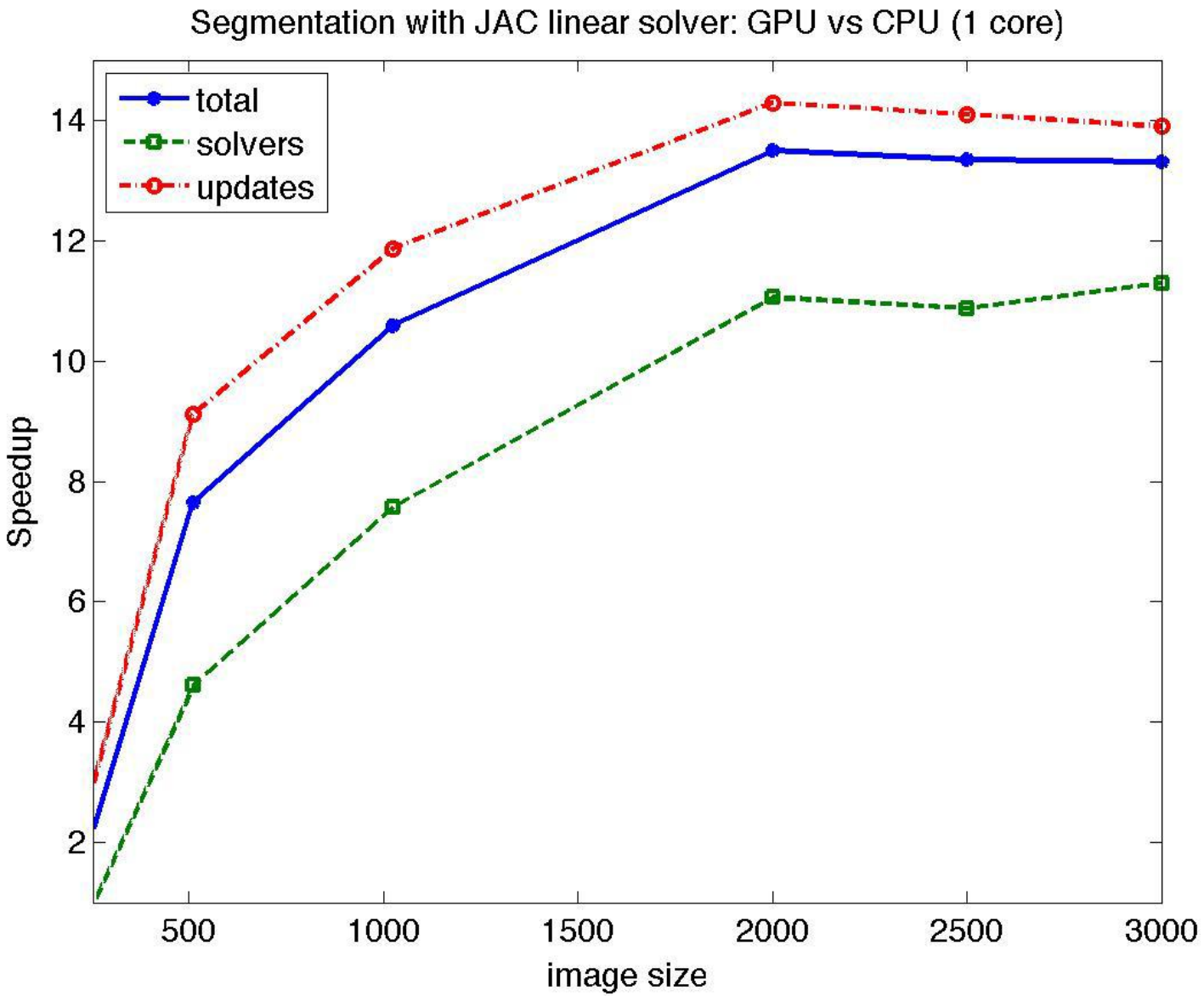


Image Segmentation Performance — CPU

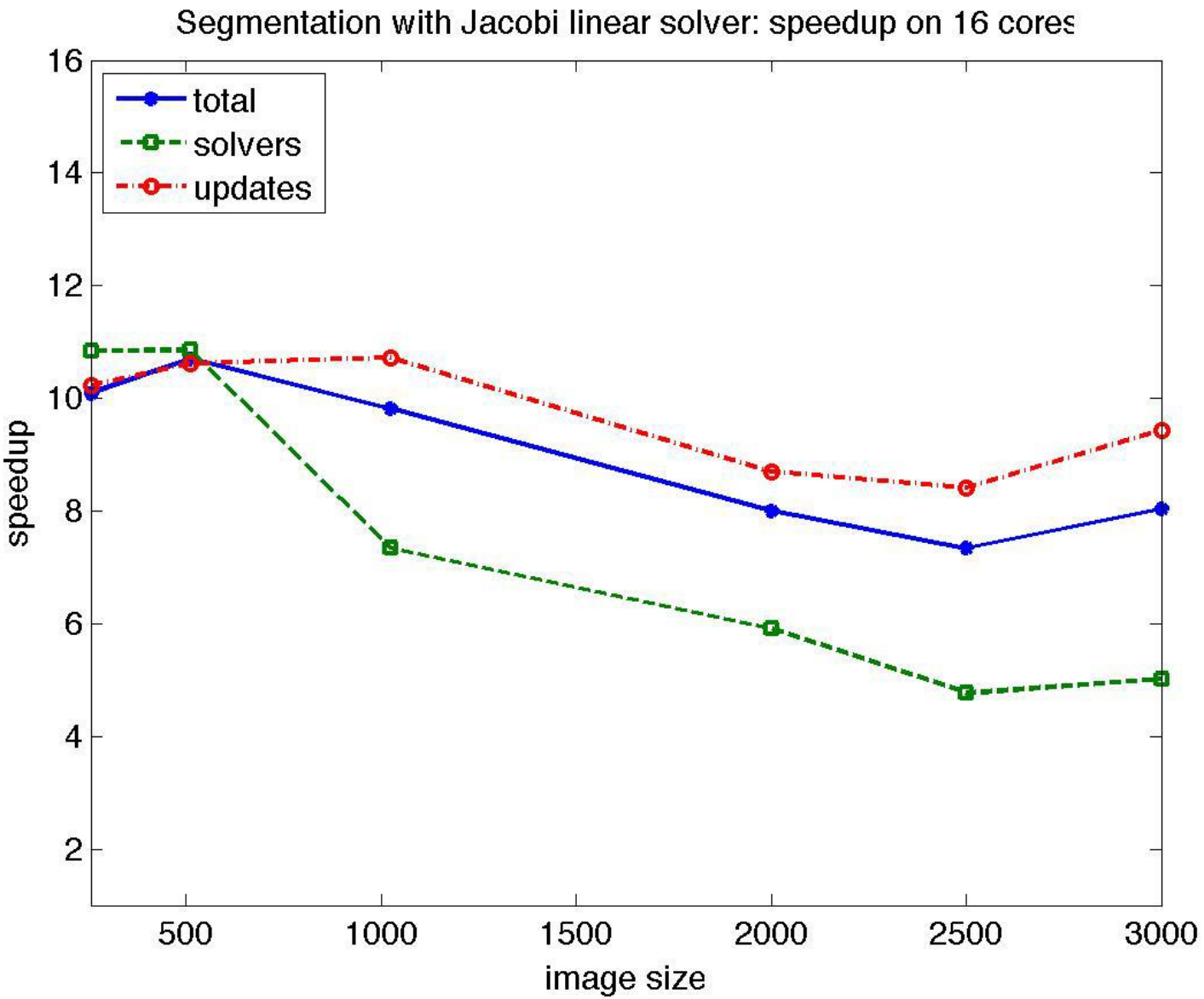


Image Segmentation Performance — 2 GPU