Optimization of Dense Linear Systems on Platforms with Multiple Hardware Accelerators

Enrique S. Quintana-Ortí

Universitat Jaume I
HPC&A
High Performance Computing and Architectures

BCW

July, 2010

Universitat Jaume I

Disclaimer

- Not a course on how to program dense linear algebra kernels on GPUs
  - Where have you been Monday-Wednesday?

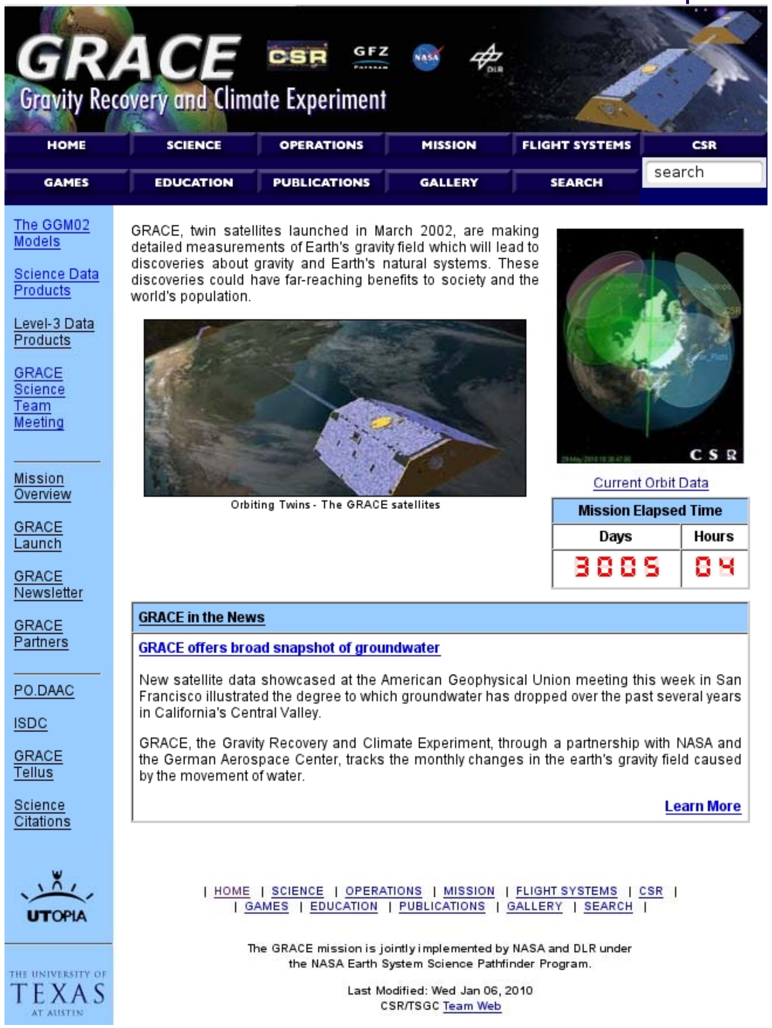
  - V. Volkov et al. “Benchmarking GPUs to tune dense linear algebra”, SC08
  - L-S. Chien, “Hand-tuned SGEMM on GT200 GPU”, TR Tsing Hua Univ., Taiwan

- Sorry if the performance numbers of some “products” do not look so good…

Large-scale linear systems: Estimation of Earth's gravity field

- GRACE project
  www.csr.utexas.edu/grace

- Solve $y = H\, x_0 + \epsilon$,
  dense $H \rightarrow m \times n$
  m = 66.000 observations
  n = 26.000 parameters for a model of resolution 250km



Large-scale linear systems

- Dense linear algebra is at the bottom of the “food chain” for many scientific and engineering apps.

- Molecular dynamics simulations
- BEM for electromagnetism and fast acoustic scattering problems
- Analysis of dielectric polarization of nanostructures
- Model reduction of VLSI circuits

Large-scale linear systems

- Dense matrix computations feature a high computational cost
  - Solving $Ax = b$, with dense $A \rightarrow n \times n$ requires $O(n^3)$ flops

- ...but GPUs love large, costly problems with regular pattern accesses

Outline

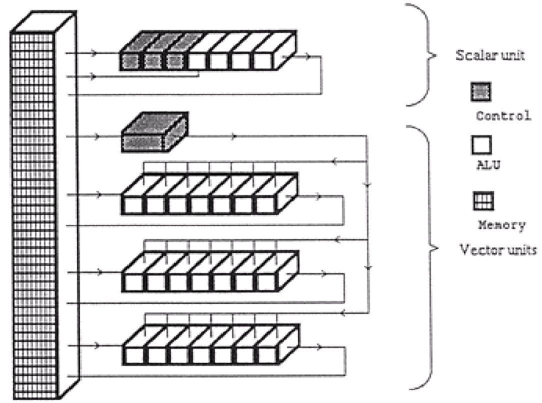
- Dense linear algebra libraries
- Optimizations for single-GPU platforms
- Programming multi-GPU platforms:
    - Shared memory
    - Clusters equipped with GPUs

Outline

- Dense linear algebra libraries
- Optimizations for single-GPU platforms
- Programming multi-GPU platforms:
  - Shared memory
  - Clusters equipped with GPUs

Dense linear algebra libraries: BLAS-1 & BLAS-2

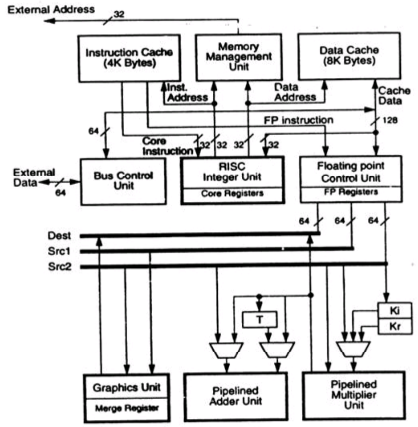
- Once upon a time, vector processors were mainstream…



- ...cast most computations as vector operations
  - BLAS-1: axpy ($y := y + \alpha\, x$), dot ($y := x^T y$)
  - BLAS-2: gemv ($y := \alpha\, y + \beta\, A\, x$), trsv ($x := T^{-1} b$)

Dense linear algebra libraries: BLAS-3

- “The attack of the killer micros”, Brooks, 1989…

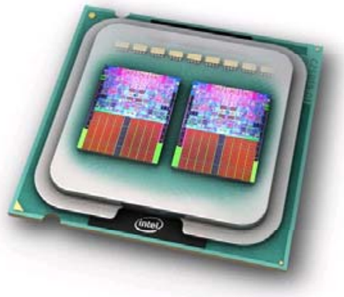


- …cast most computations in terms of operations with high data reuse
  - BLAS-3: gemm ($C := \alpha\, C + \beta\, A\, B$), trsm ($X := T^{-1} B$)

Dense linear algebra libraries: Importance of BLAS

- Provide portable performance
- Recognized by hardware vendors
  - Intel MKL
  - ACM ACML
  - IBM ESSL
  - GotoBLAS (K. Goto now with Microsoft)

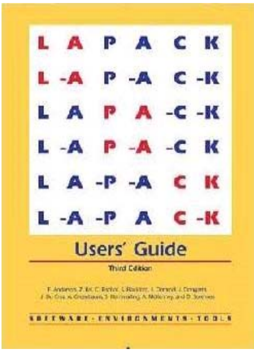
  - NVIDIA CUBLAS



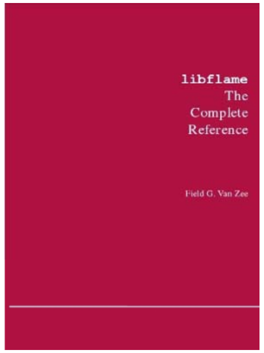
Dense linear algebra libraries

- More complex linear algebra operations:
  - Linear systems
  - Linear least-squares problems
  - Eigenvalues
  - Singular values and numerical rank

LAPACK
www.netlib.org/lapack



libflame
http://www.cs.utexas.edu/users/flame



Outline

- Dense linear algebra libraries
- Optimizations for single-GPU platforms
  - Basic performance
  - Padding
  - Matrix multiply as a building block
  - Hybrid computations for linear systems
  - Iterative refinement
  - Data transfer
- Programming multi-GPU platforms

Initial considerations

- Compute XYZ in the CPU or in the GPU?
  - Problem size
  - “Nature” of XYZ

  - Overheads:
    - Allocate/free memory in GPU
    - Data transfer between CPU and GPU
    - Invoke CUDA/CUBLAS

Experimental setup

- General-purpose
  - Intel Dual Xeon Quad-Core E5410
    - 8 cores@2.33 GHz
    - SP/DP peak 149/75 GFLOPS
    - 8 GB FB-DIMM
    - GotoBLAS2 1.11
  - AMD Phenom Quad-Core
    - 4 cores@2.2 GHz
    - 4 GB DDR2
    - GotoBLAS 1.26

- NVIDIA
  - Tesla C1060 (x4 = S1070)
    - 240 SP cores@1.3 GHz
    - SP/DP peak 933/78 GFLOPS
    - 4 GB DDR3
    - CUBLAS 2.3
  - Fermi GTX480
    - 480 SP cores@1.4 GHz
    - 1.5 GB GDDR5
    - CUBLAS 2.3

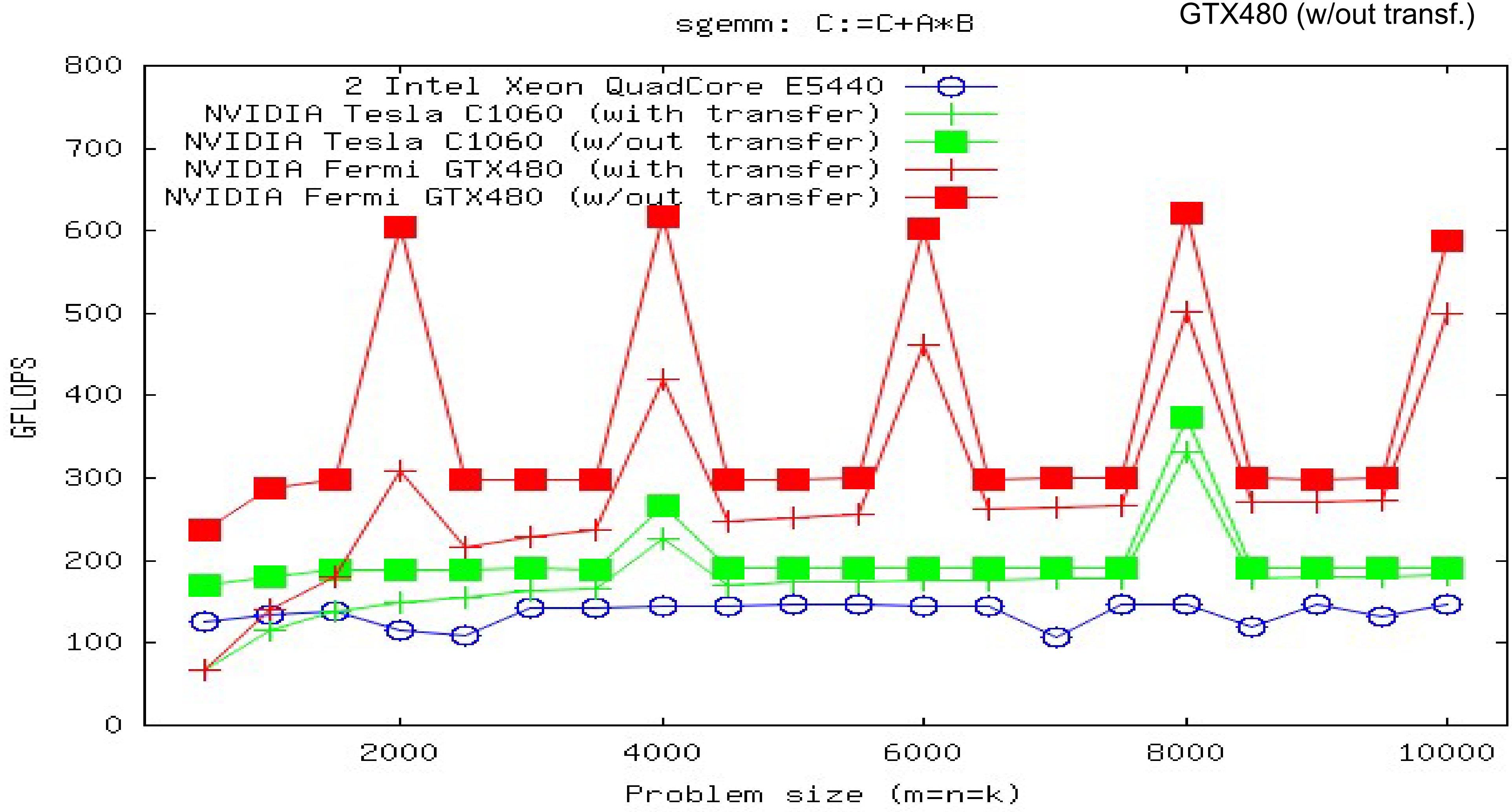
Basic performance of single GPU: gemm

$$C := C + A * B$$

- High data reuse $2n^3$ flops vs. 3 n x n data
- Variants:
  - 3 matrix dimensions: m, n, k
  - A or B can be transposed

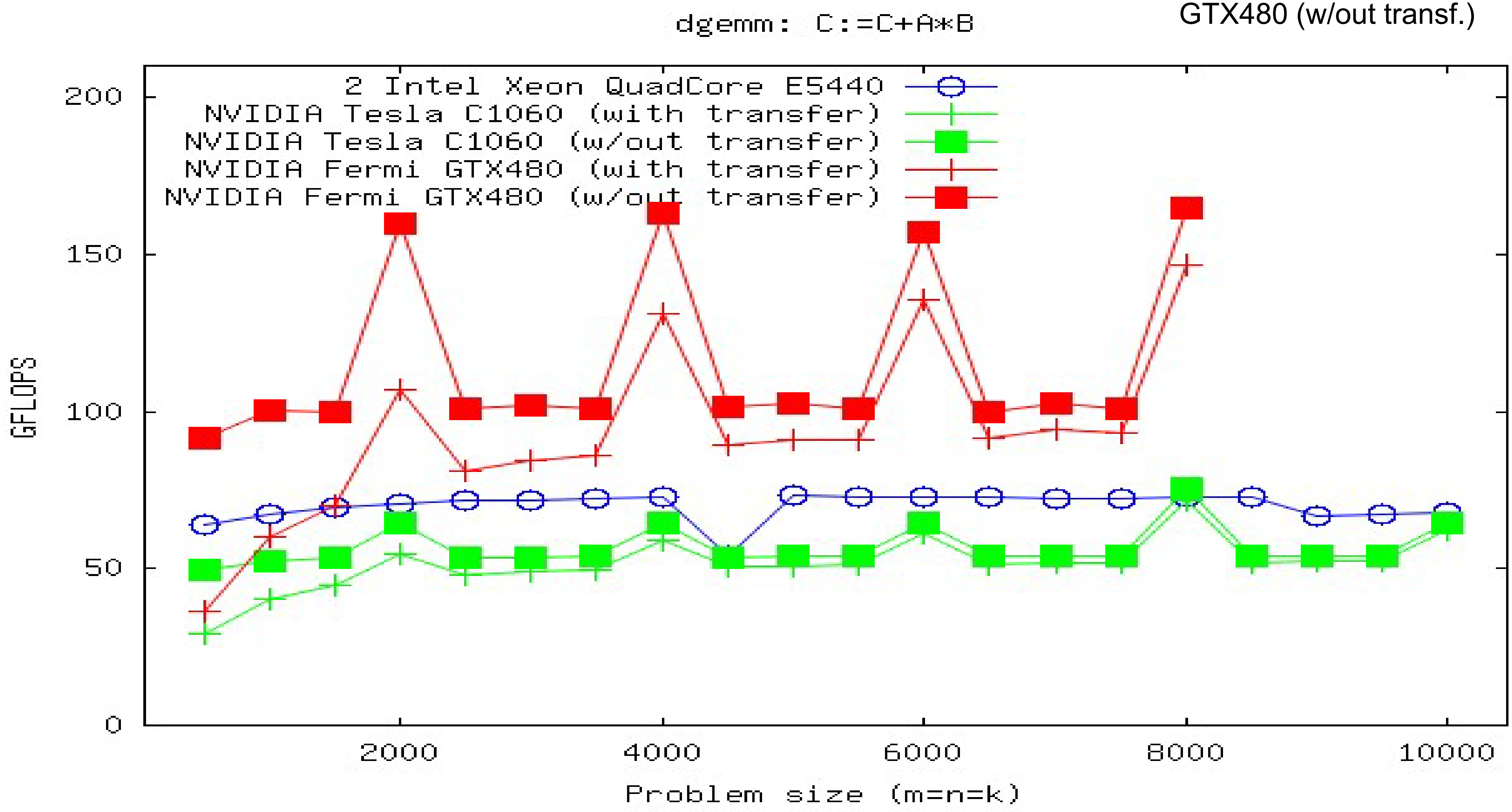
Basic performance of single GPU: gemm

Version	S_p
C1060 (with transfer)	2.26
C1060 (w/out transfer)	2.55
GTX480 (with transfer)	3.43
GTX480 (w/out transf.)	4.25



Basic performance of single GPU: gemm

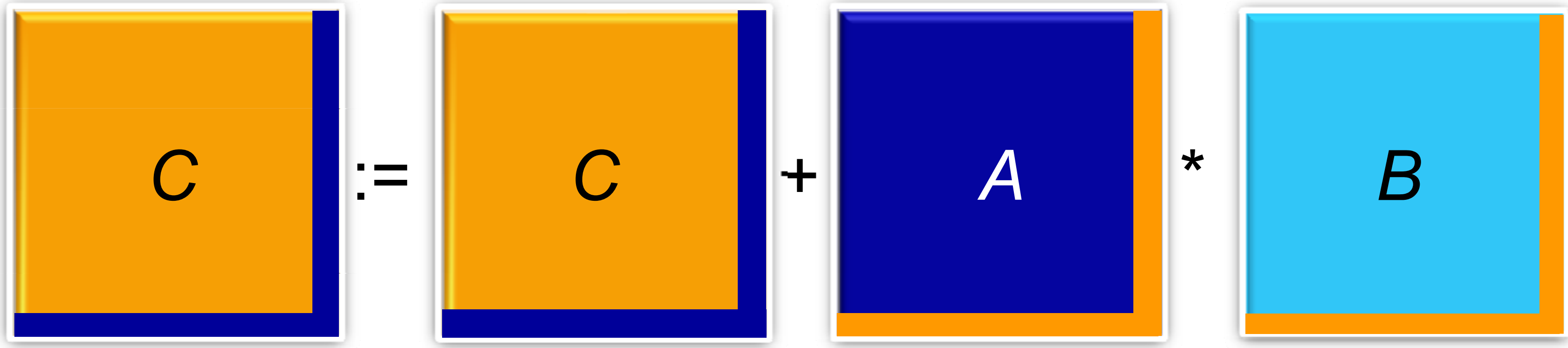
Version	S_p
C1060 (with transfer)	0.92
C1060 (w/out transfer)	0.95
GTX480 (with transfer)	2.17
GTX480 (w/out transf.)	2.43



Basic performance of single GPU: gemm

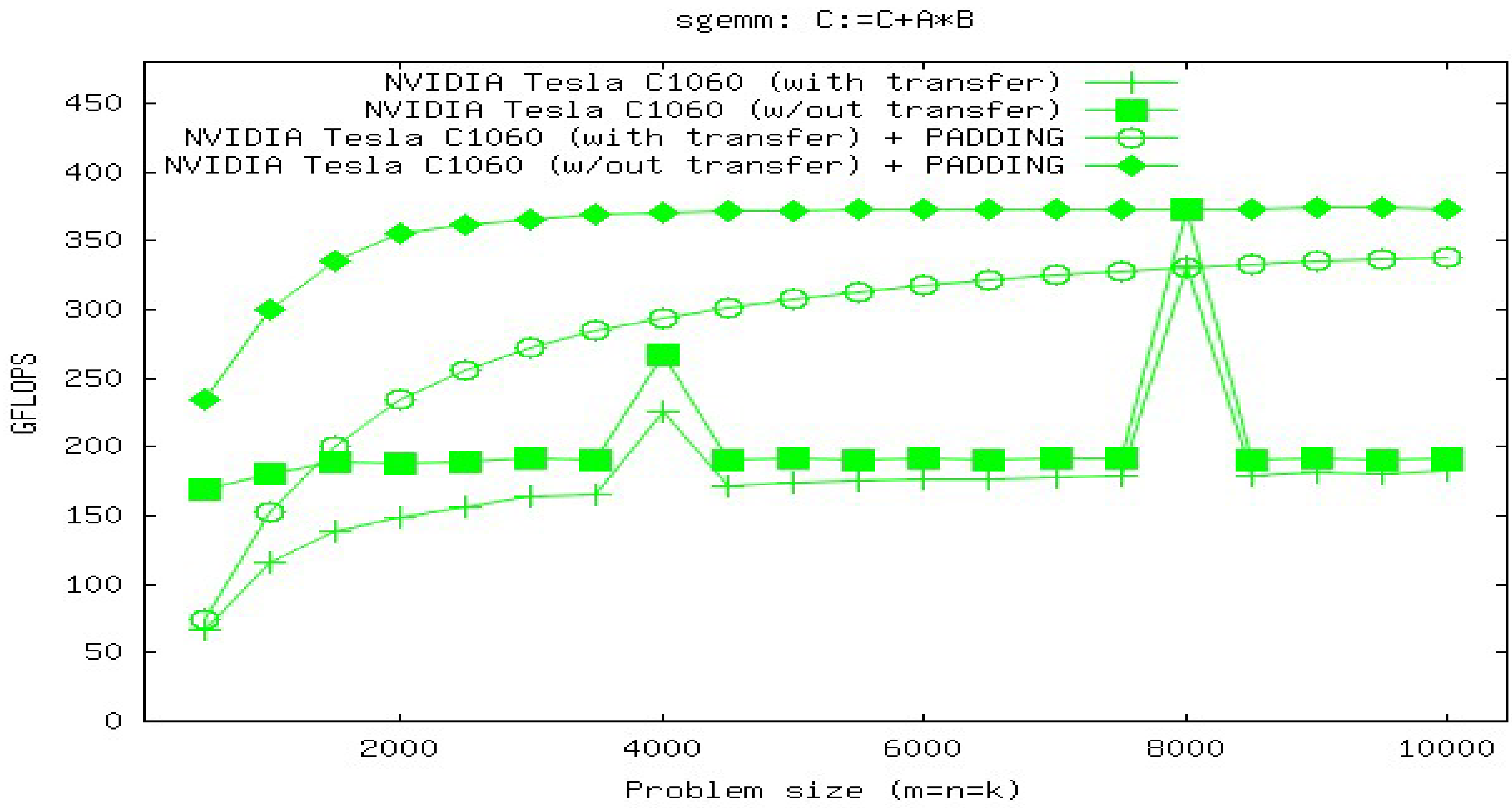
- Conclusions
  - CUBLAS irregularly optimized (use of V. Volkov gemm for problem of dimension 32*k*)
  - Data transfer amortized for large problems

Padding



- Adds negligible cost: $2n^3 \rightarrow 2n^3 + \epsilon$ flops
- Applicable to many other operations
- Can be made transparent to the user

Padding



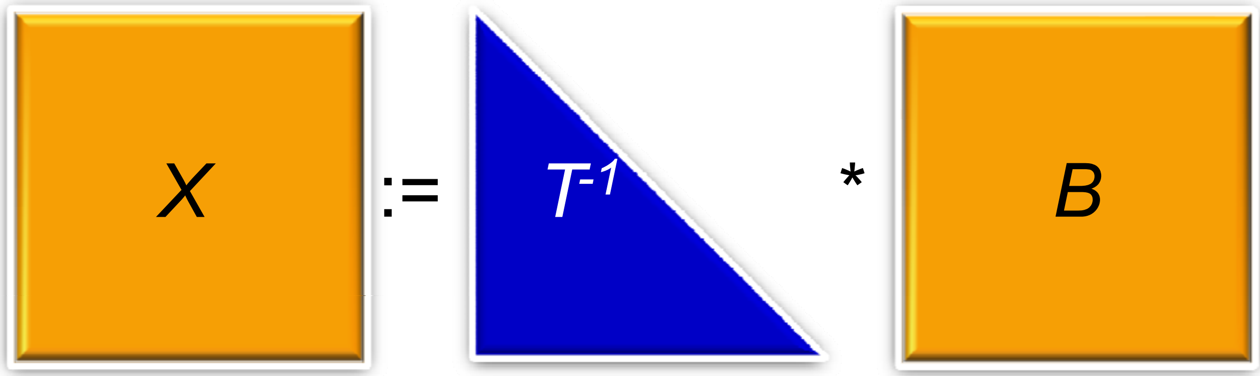
sgemm: C:=C+A*B
NVIDIA Tesla C1060 (with transfer)
NVIDIA Tesla C1060 (w/out transfer)
NVIDIA Tesla C1060 (with transfer) + PADDING
NVIDIA Tesla C1060 (w/out transfer) + PADDING
GFLOPS
450
400
350
300
250
200
150
100
50
0
2000
4000
6000
8000
10000
Problem size (m=n=k)

Matrix multiply as a building block

- Other BLAS-3
  - Variants of matrix-matrix product: trmm, symm
  - Triangular system solve: trsm
  - Symmetric rank-k update: syrk
  - Symmetric rank-2k update: syr2k

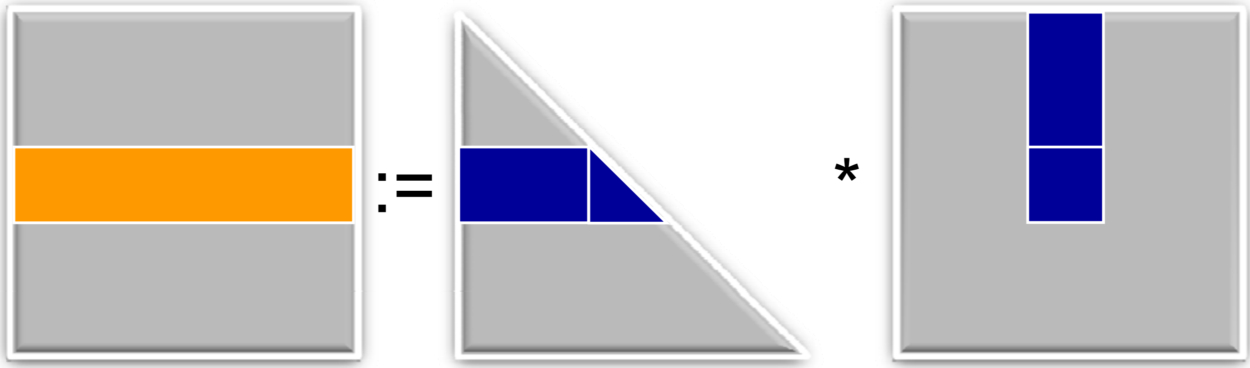
- Same ratio computation/communication (data) as gemm

Matrix multiply as a building block: trsm



- High data reuse O(n^3) flops vs. O(n^2) data
- Variants:
  - 2 matrix dimensions: *m*, *n*
  - *T* can be transposed, appear on the right/left of *B*, be upper or lower triangular

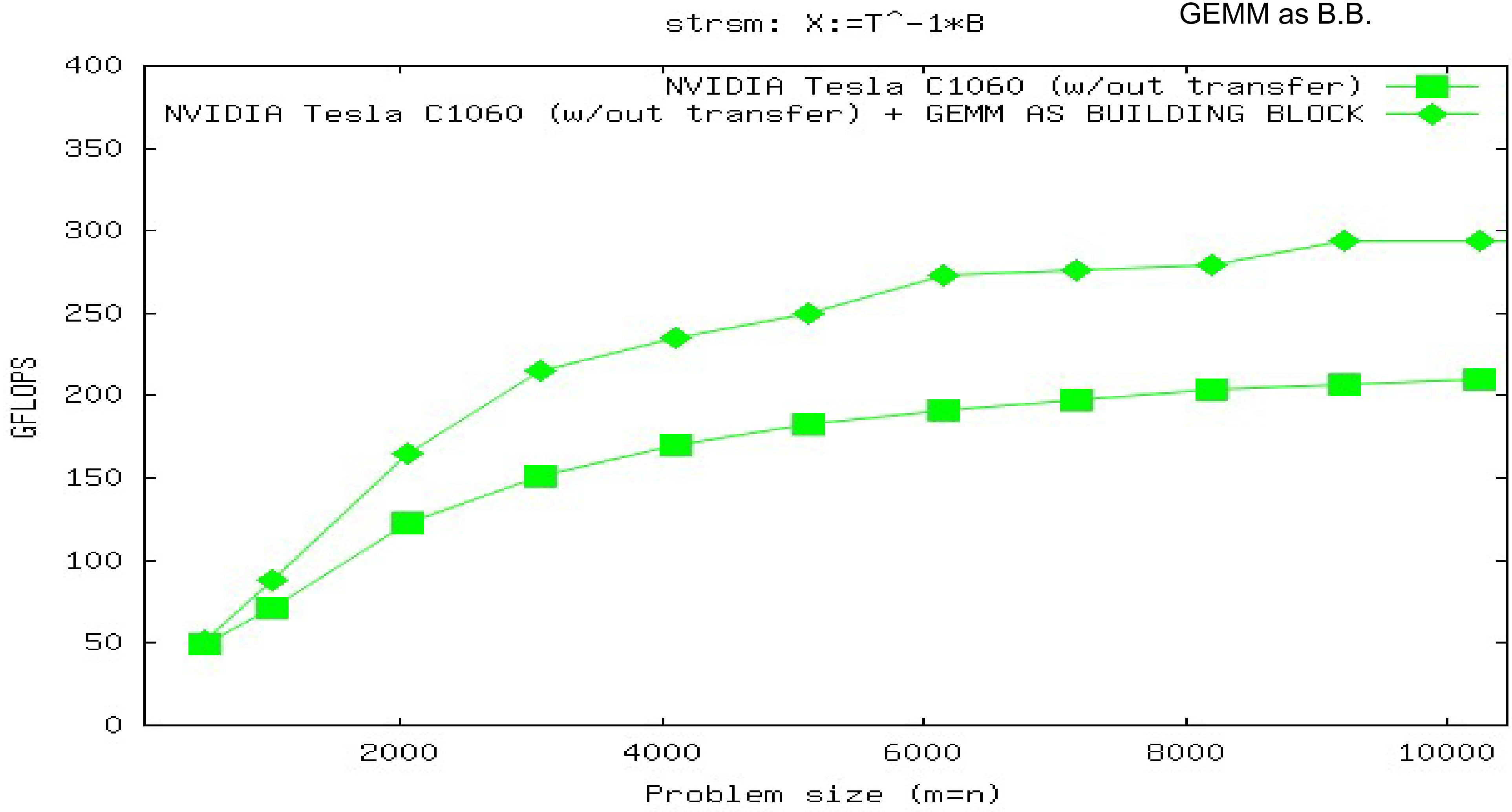
Matrix multiply as a building block: trsm



- Build trsm as a series of gemm plus small trsm
- “Poormen” BLAS: cast operations as gemm

Matrix multiply as a building block: trsm

Version	S_p
GEMM as B.B.	1.50



Hybrid computations for linear systems

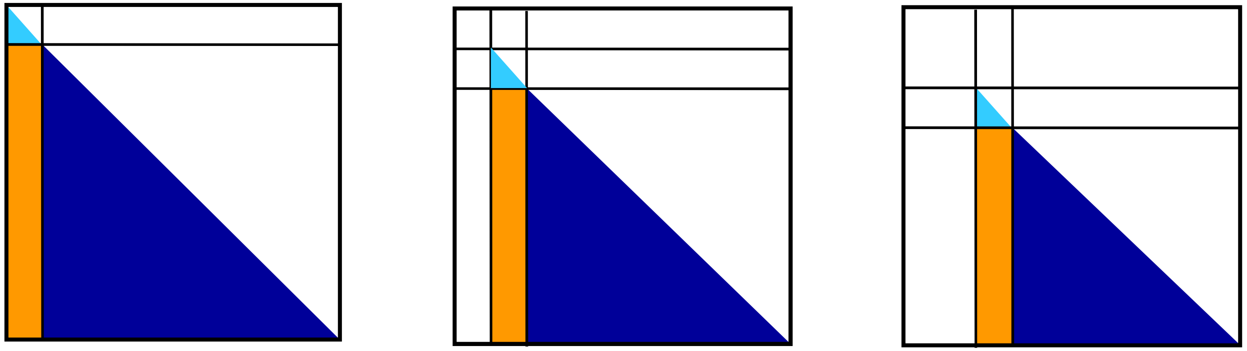
$$x := A^{-1} * b$$

- For dense *A*, decompose it into simpler factors
- Several factorization methods:
  - LU factorization (Gaussian elimination) for general *A*
  - Cholesky factorization for s.p.d. *A*
  - QR factorization for overdetermined *A* (linear least squares)

Hybrid computations: Cholesky factorization

- At each iteration, compute one more block of columns (panel) of *L*
- Overwrite (lower triangle of) *A* with *L*

Hybrid computations: Cholesky factorization



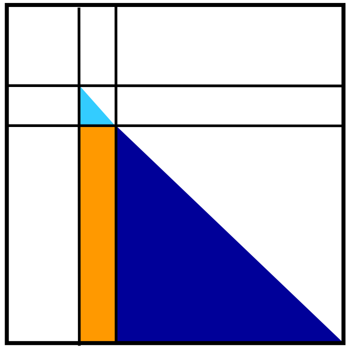
1st iter. 2nd iter. 3rd iter. ...

$A_{11} = L_{11} L_{11}^T$

$A_{21} = L_{21} := A_{21} L_{11}^{-T}$

$A_{22} := A_{22} - L_{21} L_{21}^T$

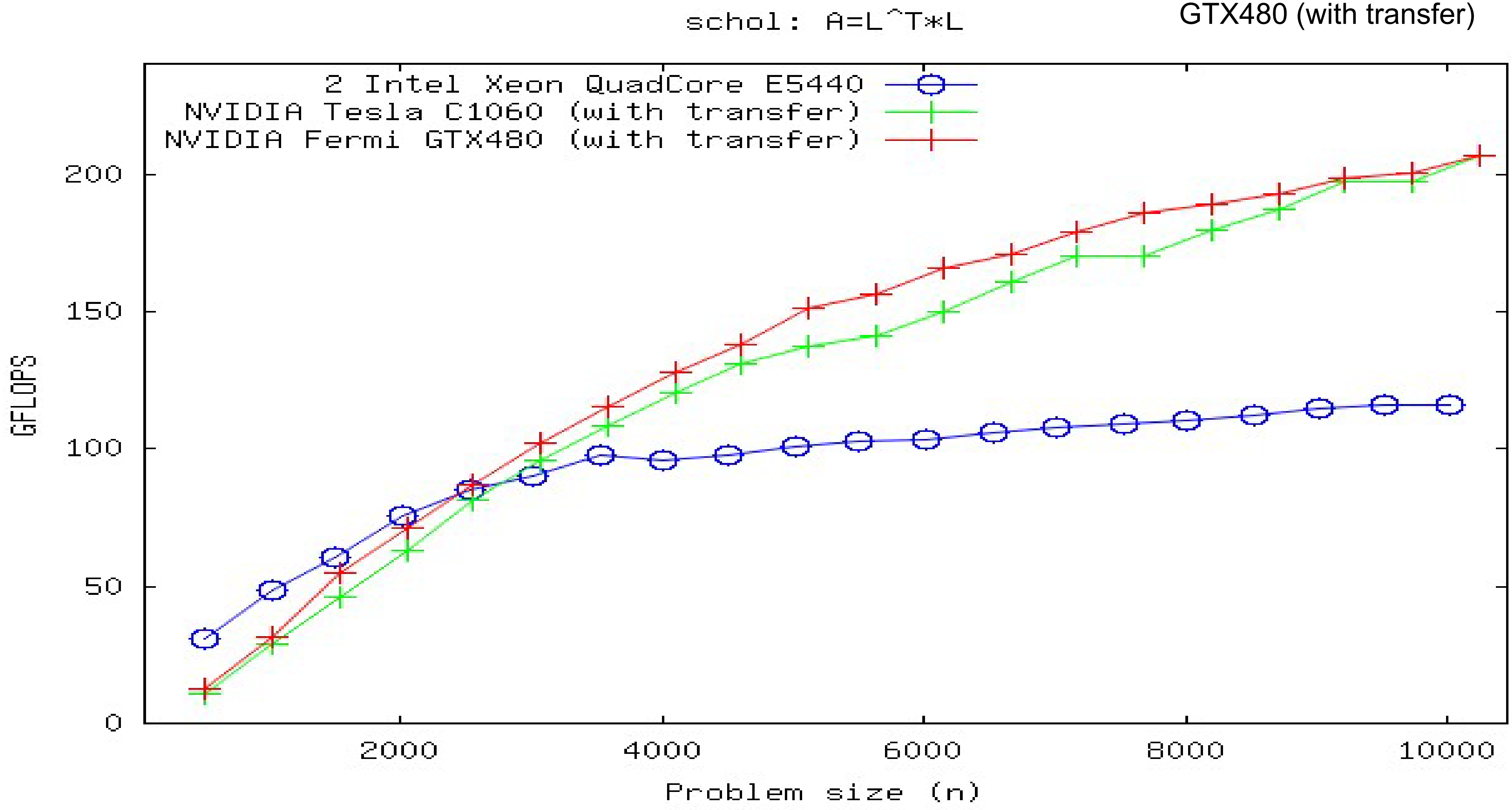
Hybrid computations: Cholesky factorization



- Insight:
  - Off-load non-computationally intensive operations to CPU
- Initially, move all *A* to GPU
- At each iteration:
  - Move A_{11} to CPU, factor block there, and send results back to GPU
  - Update A_{21} and A_{22} on the GPU

Hybrid computations: Cholesky factorization

Version	S_p
C1060 (with transfer)	1.77
GTX480 (with transfer)	1.77



Iterative refinement

- For most apps., double precision is the norm
  - …but GPUs (before Fermi) are significantly faster with single precision arithmetic
- For linear systems, iterative refinement is a cheap method to recover double precision from a single precision approximation!

Iterative refinement

$A_s = L_s\, L_s^T$ *Single precision; $O(n^3)$ flops*

$x_s := L_s^{-T}\,(L_s^{-1}\, b_s)$ *Single precision; $O(n^2)$ flops*

$x := x_s$

`repeat`

$r := b - A\,x$ *Double precision; $O(n^2)$ flops*

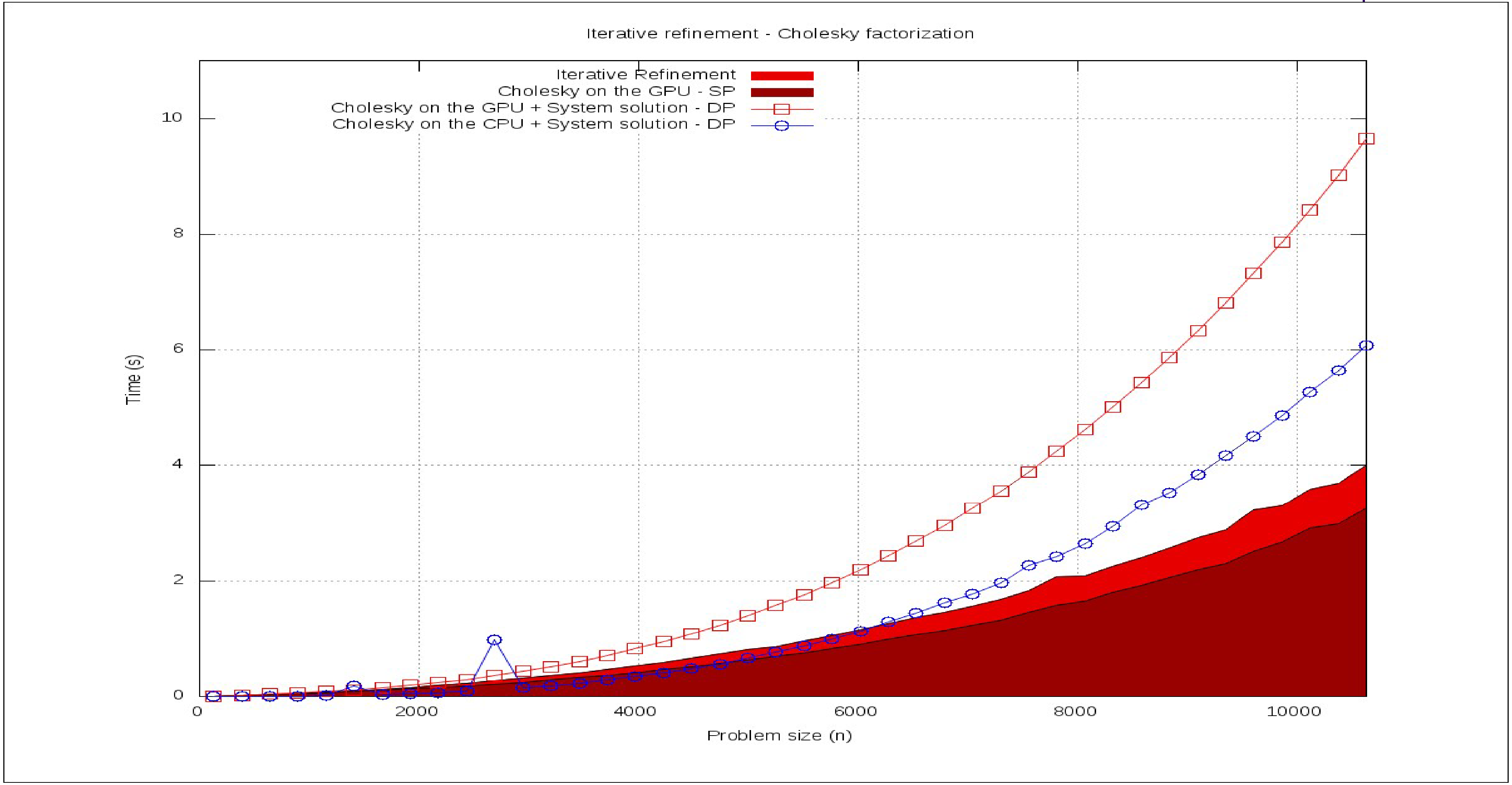
$r_s := r$

$z_s := L_s^{-T}\,(L_s^{-1}\, r_s)$ *Single precision; $O(n^2)$ flops*

$z := z_s$

$x := x + z$ *Double precision; $O(n)$ flops*

Iterative refinement



Data transfer

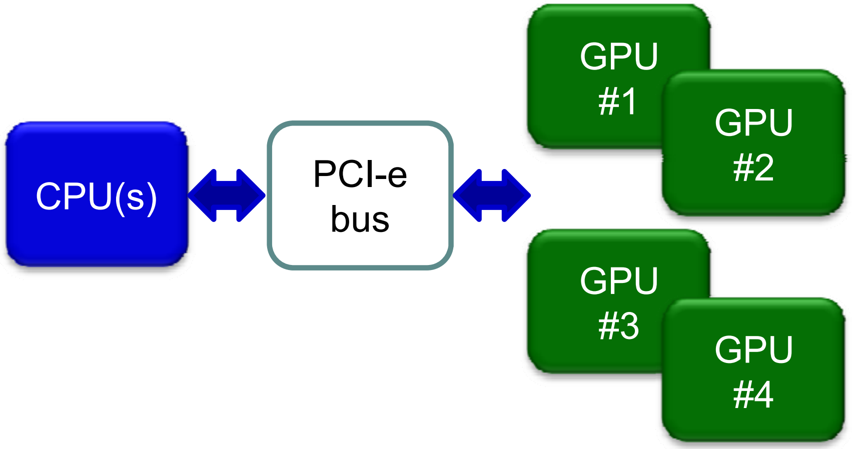
- Who is in control of data transfers between GPU and CPU?
  - User (programmer) via CUDA API
  - System: part of a runtime
    - GMAC
    - SuperMatrix/`libflame`
    - GPUSs

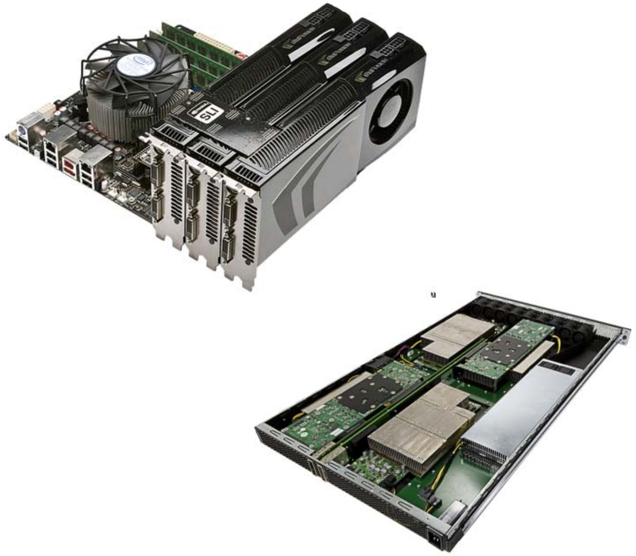
Outline

- Dense linear algebra libraries
- Optimizations for single-GPU platforms
- Programming multi-GPU platforms:
  - Multi-GPU platforms
  - Clusters equipped with GPUs

Programming multi-GPU platforms

- How do we program these?



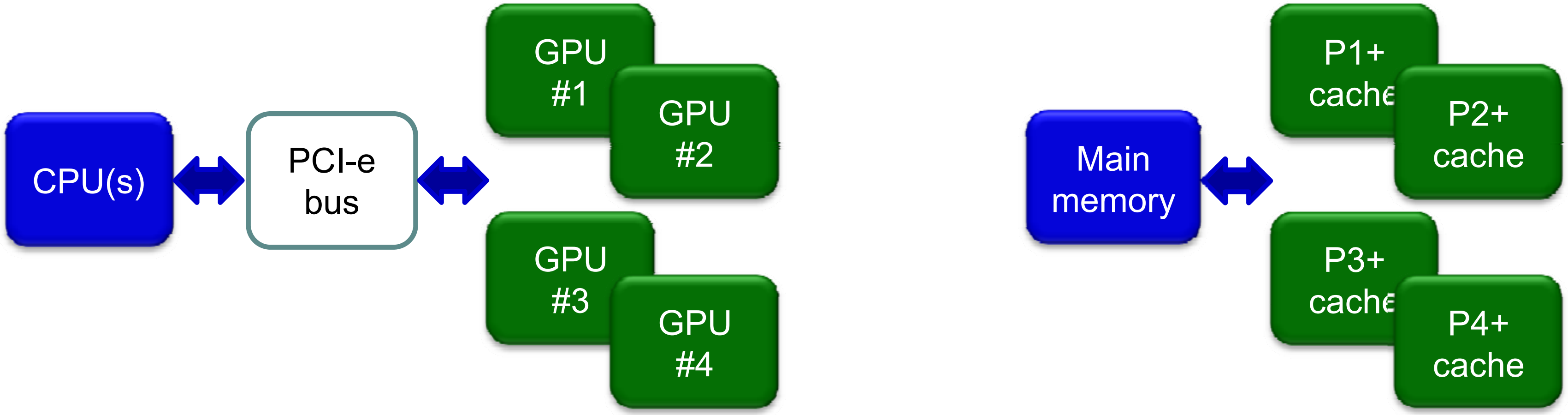


View as a…

- Shared-memory multiprocessor
- Cluster (distributed-memory)

Programming multi-GPU platforms

- Shared memory multiprocessor view

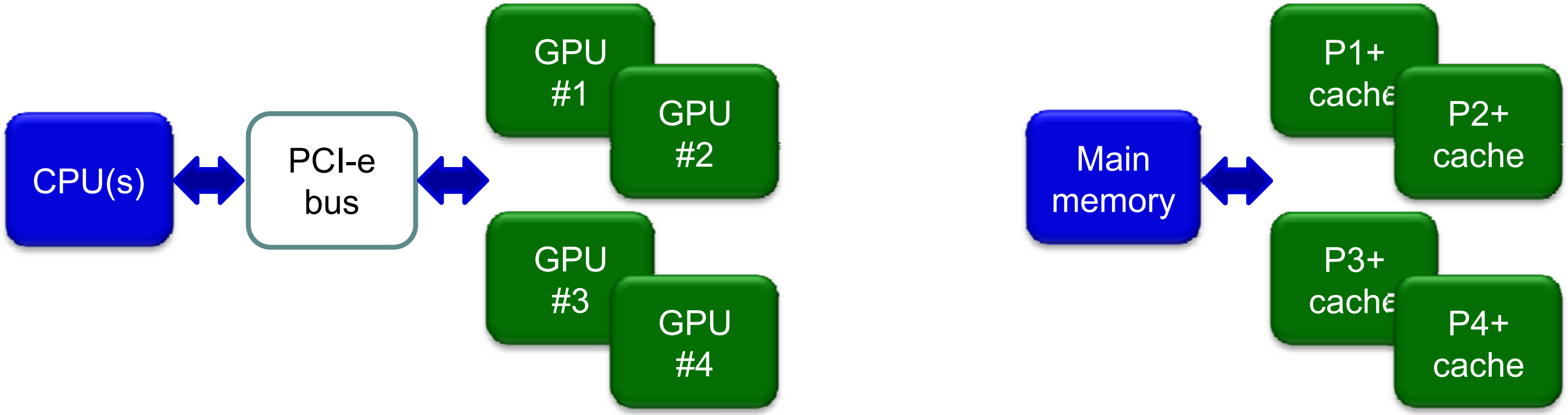


Not straight-forward:

- Heterogeneous system: *n* CPUs + *m* GPUs
- Multiple address spaces: 1 + *m*

Programming multi-GPU platforms

- Shared memory multiprocessor view

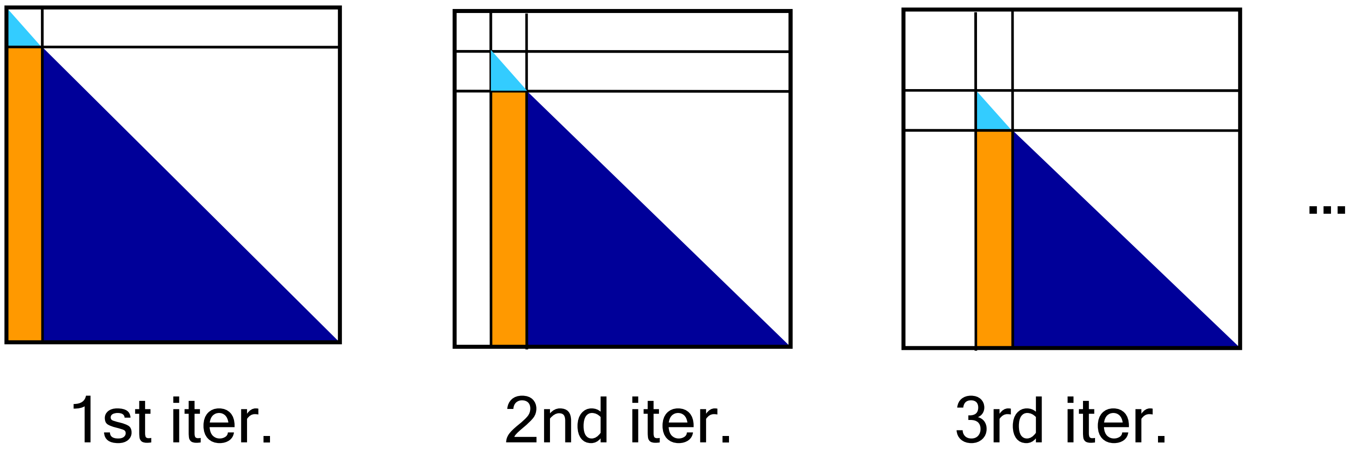


Not straight-forward → Run-time system!

- Heterogeneous system: Task scheduling (temporal+spatial)
- Multiple address spaces: Data movement

Task scheduling in multi-GPU platforms

- Cholesky factorization



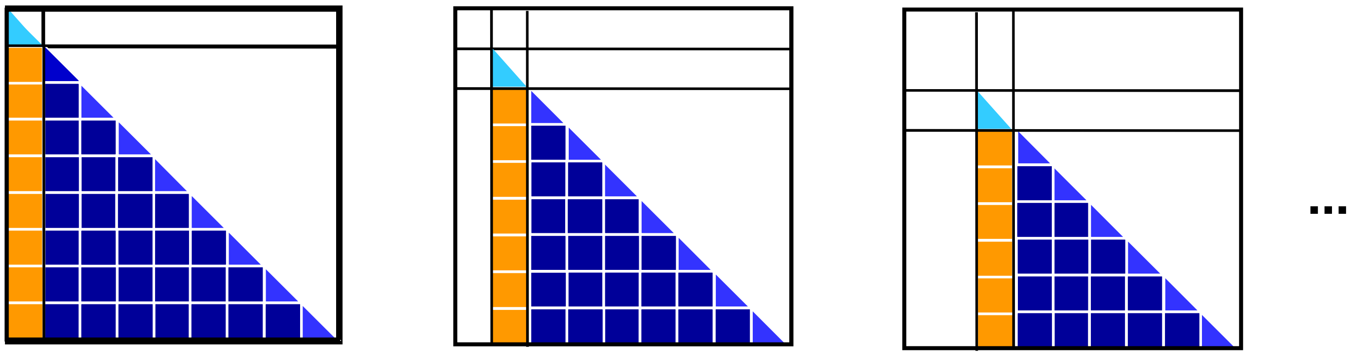
$$A_{11} = L_{11}\,L_{11}^{T}$$

$$A_{21} = L_{21} := A_{21}\,L_{11}^{-T}$$

$$A_{22} := A_{22} - L_{21}\,L_{21}^{T}$$

Task scheduling in multi-GPU platforms

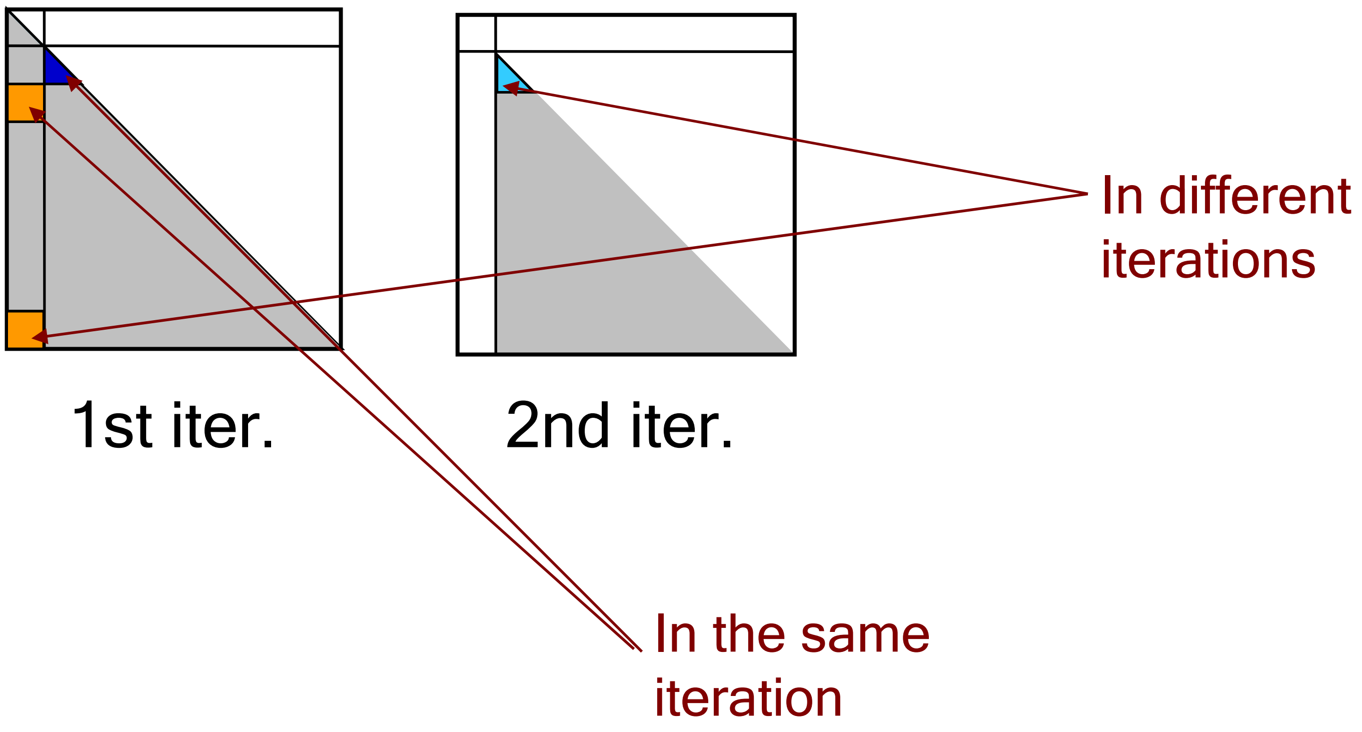
- Plenty of tasks…



1st iter. 2nd iter. 3rd iter.

Task scheduling in multi-GPU platforms

- Plenty of task parallelism!

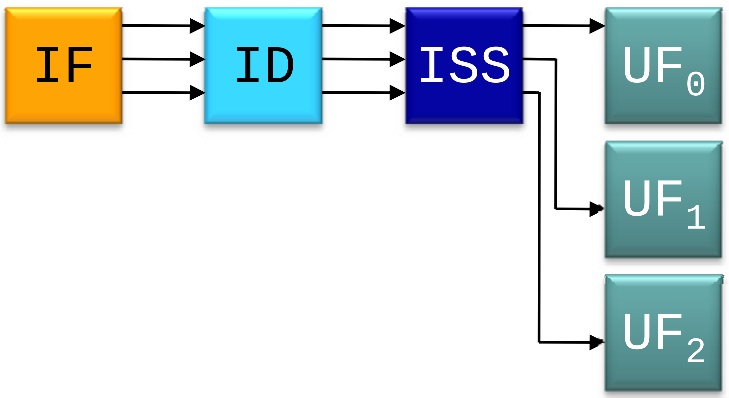


Task scheduling in multi-GPU platforms

Scalar code

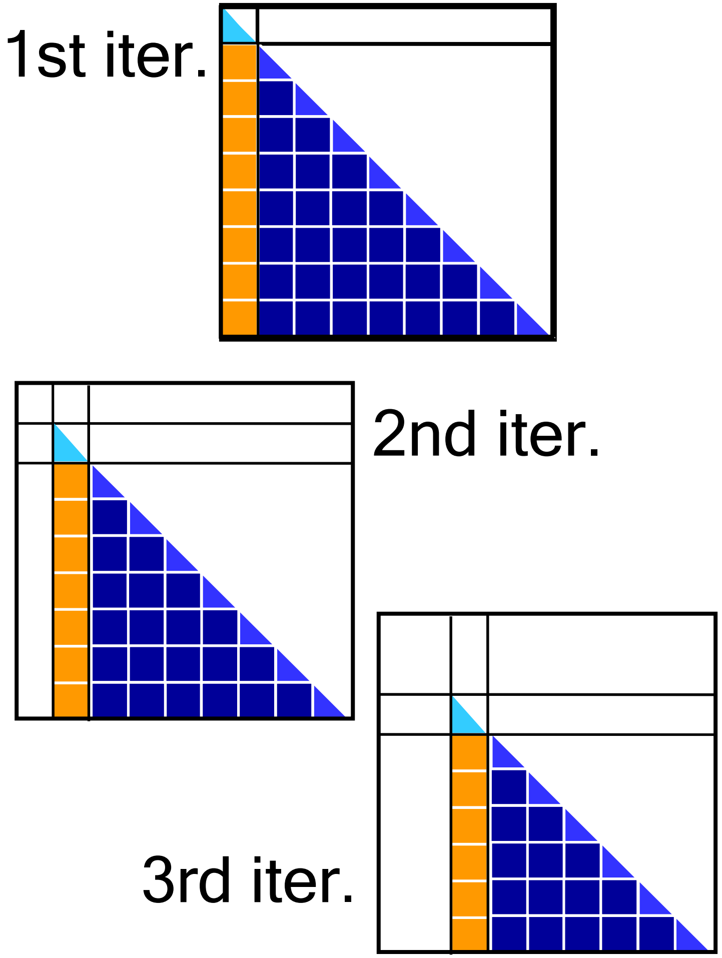
```
loop: ld   f0, 0(r1)
      addd f4, f0, f2
      sd   f4, 0(r1)
      addi r1, r1, #8
      subi r2, r2, #1
      bnez r2, loop
```

(Super)scalar processor



Task scheduling in multi-GPU platforms

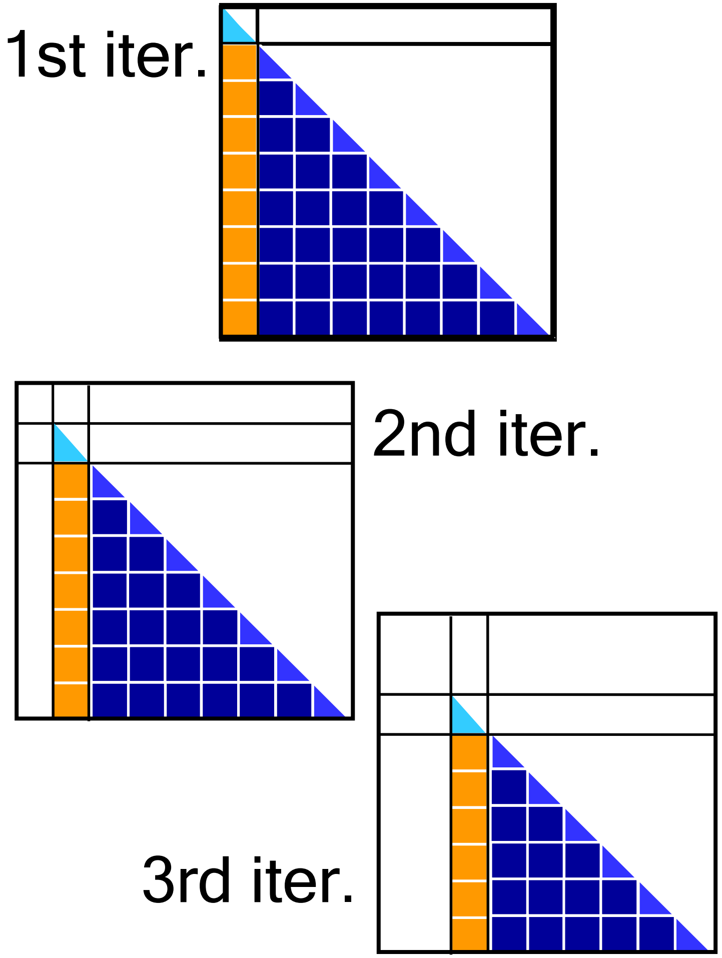
- Something similar for (dense) linear algebra?




```
for (k=0; k<nb; k++){
F:  Chol(A[k,k]);
    for (i=k+1; i<nb; i++)
T:  Trsm(A[k,k], A[i,k]);
    for (i=k+1; i<nb; i++){
P:  Syrk(A[i,k],A[i,i]);
    for (j=k+1; j<i; j++)
P:    Gemm(A[i,k], A[j,k], A[i,j]);
    }
}
```

Task scheduling in multi-GPU platforms

- Something similar for (dense) linear algebra?



- Apply “scalar” techniques at the block level
- Software implementation
- Thread/Task-level parallelism
- Target the cores/GPUs of the platform

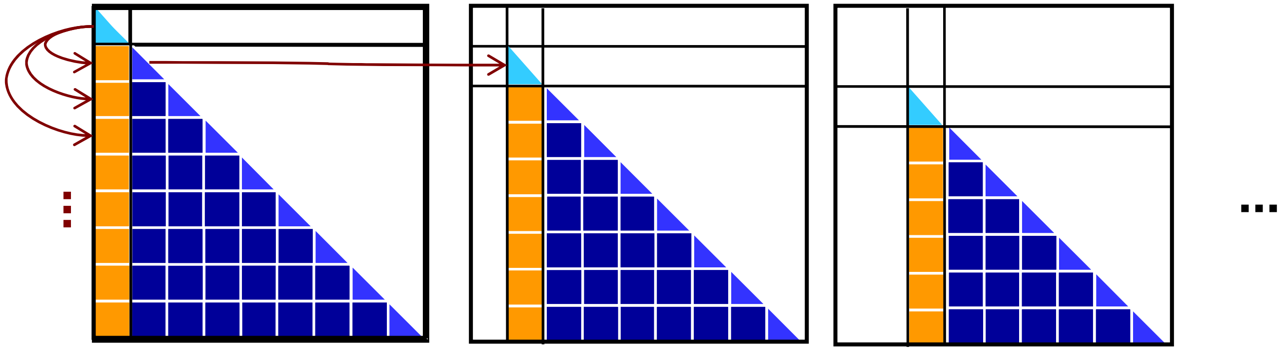
Task scheduling in multi-GPU platforms

- Read/written blocks determine dependencies, as in scalar case

```
loop:  ld   f0, 0(r1)
       addd f4, f0, f2
       sd   f4, 0(r1)
       addi r1, r1, #8 …
```

```
for (k=0; k<nb; k++){
  Chol(A[k,k]);
  for (i=k+1; i<nb; i++)
    Trsm(A[k,k], A[i,k]);
```

Dependencies form a task tree



Task scheduling in multi-GPU platforms

- Blocked code:

```
for (k=0; k<nb; k++){
  Chol(A[k,k]);
  for (i=k+1; i<nb; i++)
    Trsm(A[k,k], A[i,k]); …
```

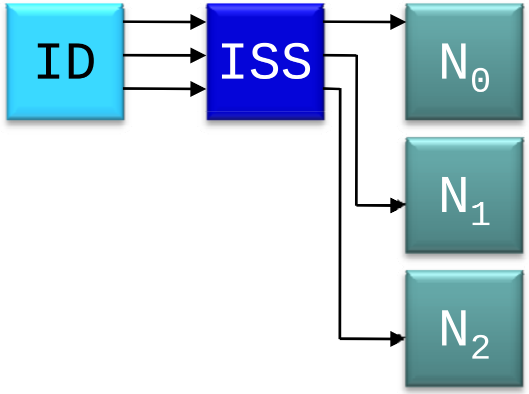
Multi-core/multi-GPU

- How do we generate the task tree?
- What needs to be taken into account to execute the tasks in the tree?

Task scheduling in multi-GPU platforms

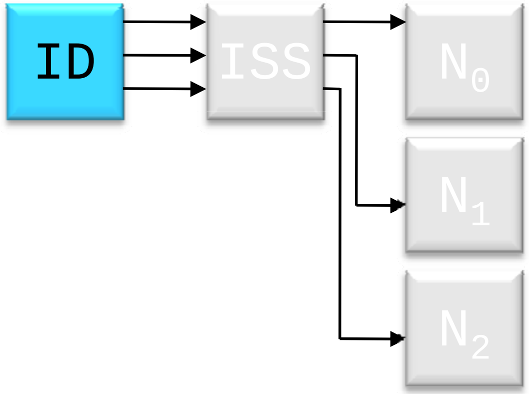
- Use of a *runtime*:
  - Decode (ID): Generate the task tree with a "symbolic analysis" of the code at execution time
  - Issue (ISS): Architecture-aware execution of the tasks in the tree

→



Task scheduling in multi-GPU platforms

- Decode stage:
  - “Symbolic analysis” of the code

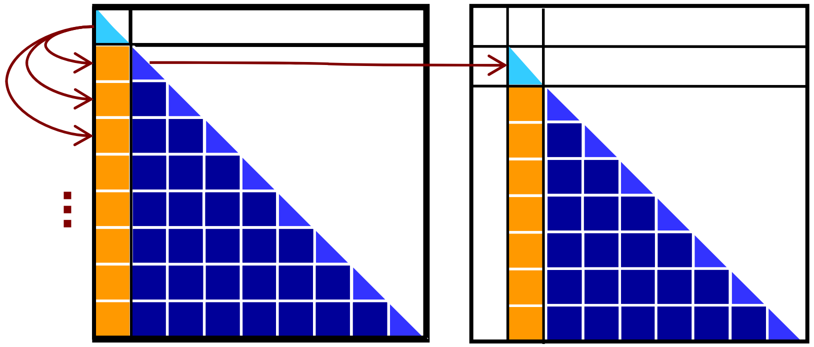


Blocked code:

```
for (k=0; k<nb; k++){
  Chol(A[k,k]);
  for (i=k+1; i<nb; i++)
    Trsm(A[k,k], A[i,k]); …
```

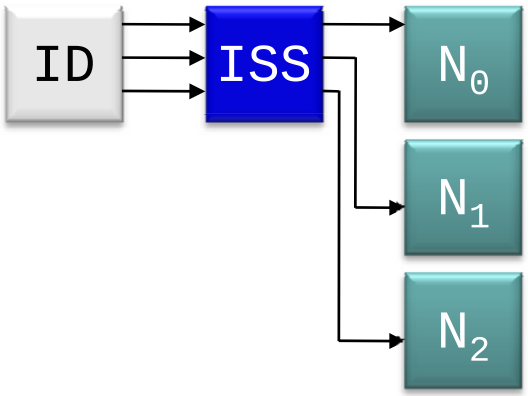
$\rightarrow$

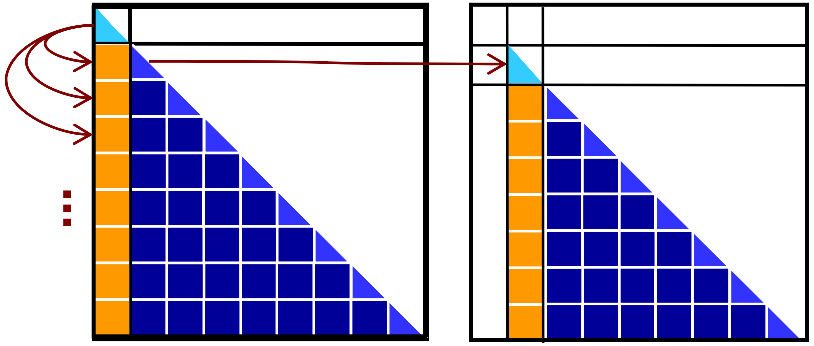
Task tree:



Task scheduling in multi-GPU platforms

- Issue stage:
  - Temporal scheduling of tasks, attending to dependencies
  - Mapping (spatial scheduling) of tasks to resources, aware of locality

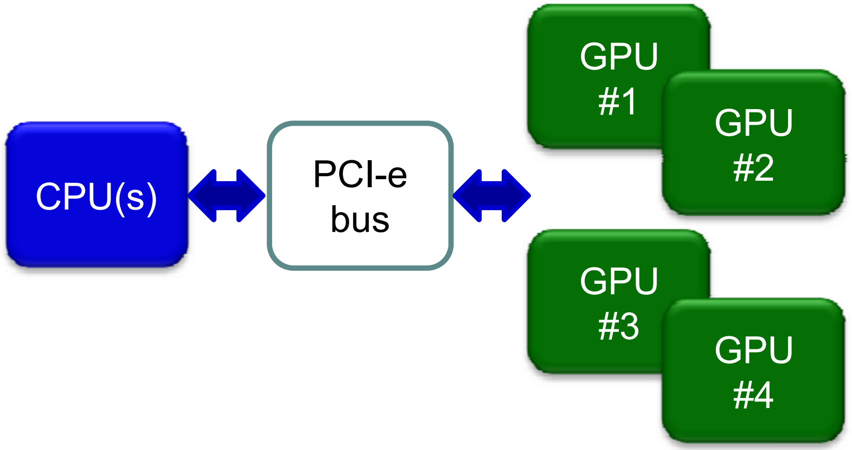


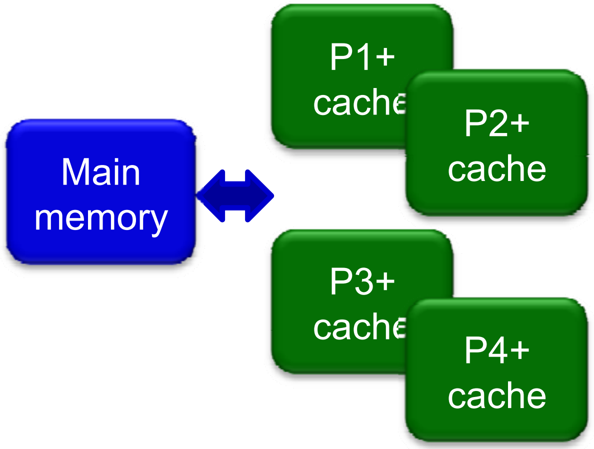


$\rightarrow$

Programming multi-GPU platforms

- Shared memory multiprocessor view





Not straight-forward → Run-time system!

- Heterogeneous system: Task scheduling (temporal+spatial)
- Multiple address spaces: Data movement

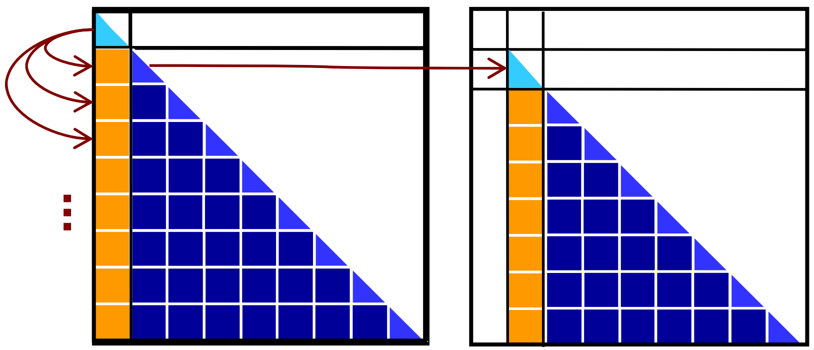
Multiple address spaces

- Software Distributed-Shared Memory (DSM)
  - Underlying distributed memory hidden from the users
  - Well-known approach, not too efficient as a middleware for general apps.

  - Regularity of dense linear algebra operations makes a difference!

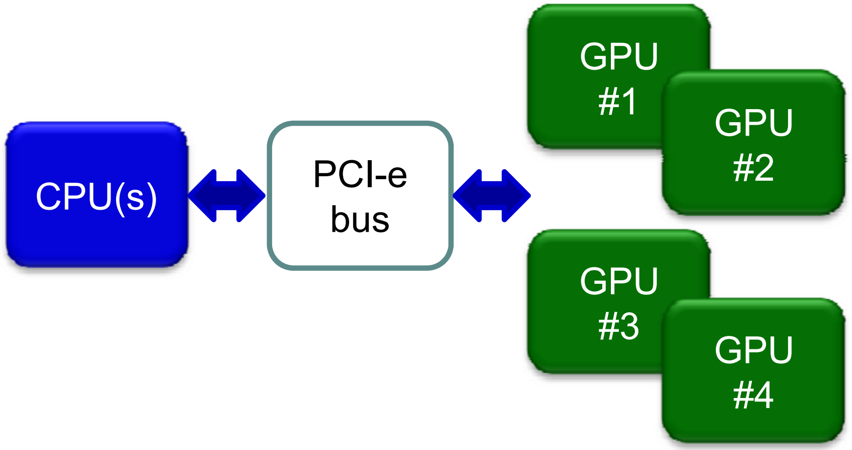
Multiple address spaces

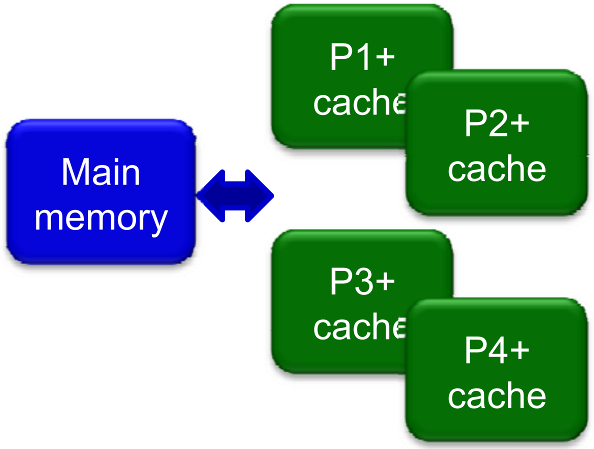
- Naive approach:
  - Before executing a kernel, copy input data to GPU memory
  - After execution, retrieve results back to CPU memory
  - Easy to program (wrappers to kernels)
  - *$O(n^3)$* Transfers between CPU and GPU



Multiple address spaces

- Shared memory multiprocessor view



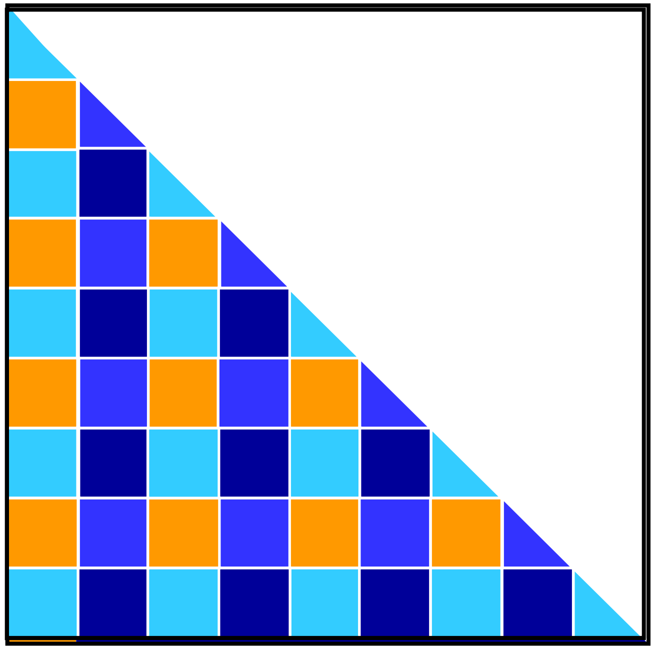


Key to reduce #data transfers!

- Static mapping/dynamic scheduling
- Software cache
- Cache/memory coherence policies

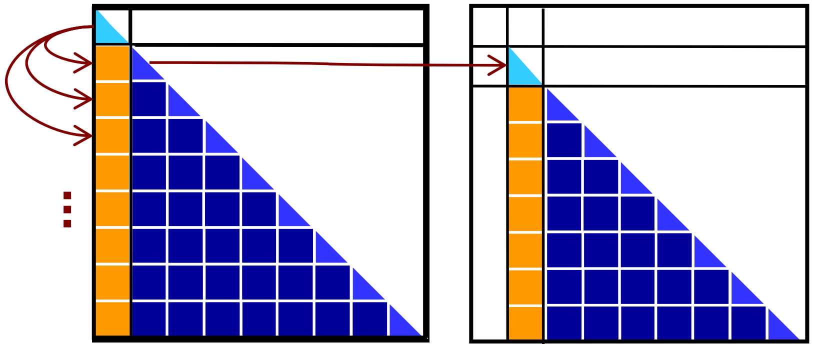
Multiple address spaces

- Where? Static mapping of tasks to resources
  - Writes to a given block are always performed by the same resource (owner-computes rule)
  - Cyclic mappings: row, column, 2-D



Multiple address spaces

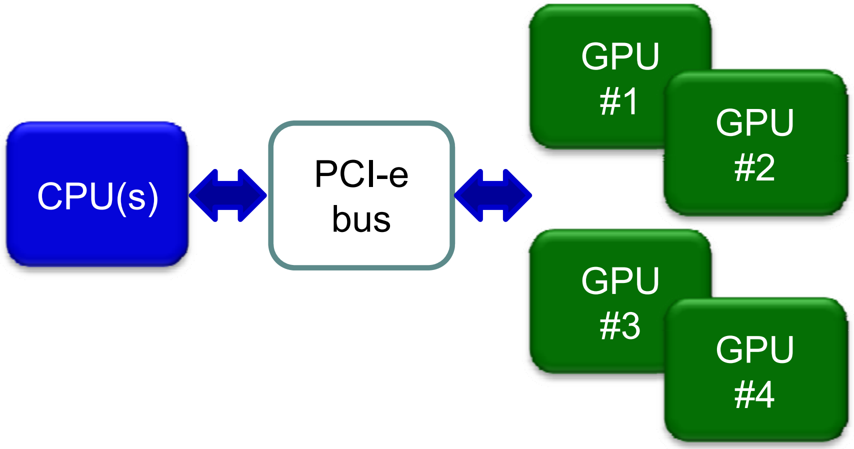
- When? Dynamic scheduling of tasks
  - As soon as data dependencies are fulfilled…
  - Possibility of prioritizing tasks in the critical path



→

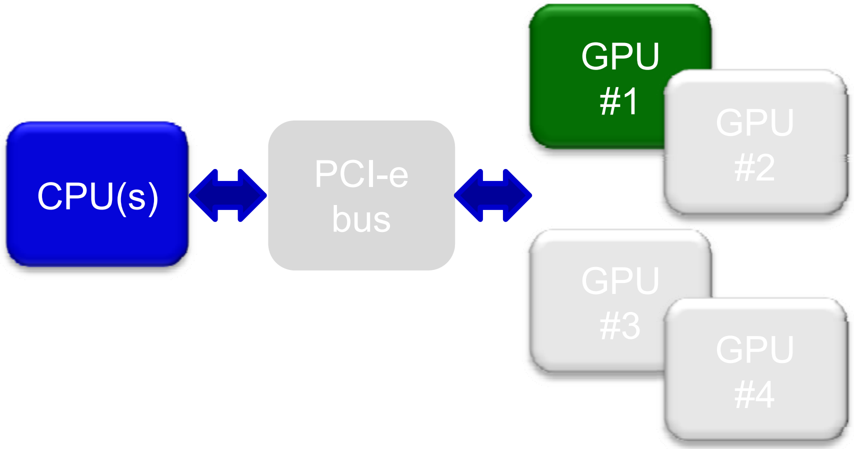
Multiple address spaces

- Software cache (flexibility vs. efficiency)
  - Maintain a map of memory
  - Operate at the block level (amortize software handling with #flops)
  - Once data is in the GPU mem., keep it there as long as possible
  - LRU (or more advanced)



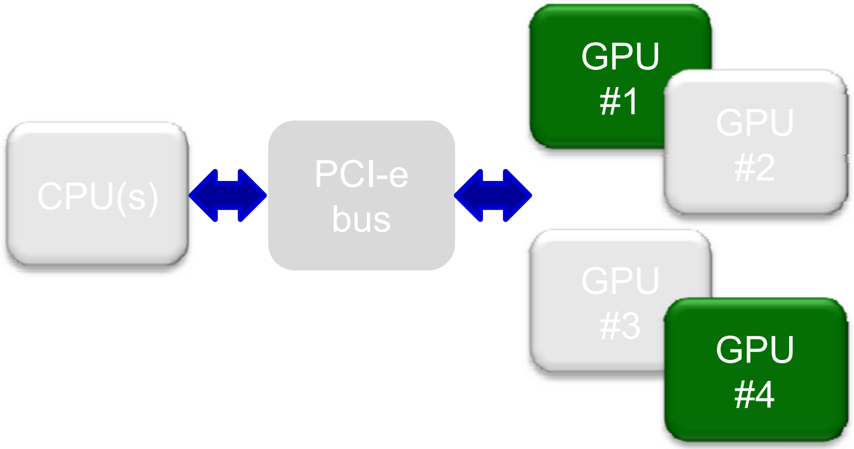
Multiple address spaces

- Coherence between GPU and main memories
  - Write-back



Multiple address spaces

- Coherence among GPU memories
  - Write-invalidate
  - Requires transfer via main memory



Run-time implementations

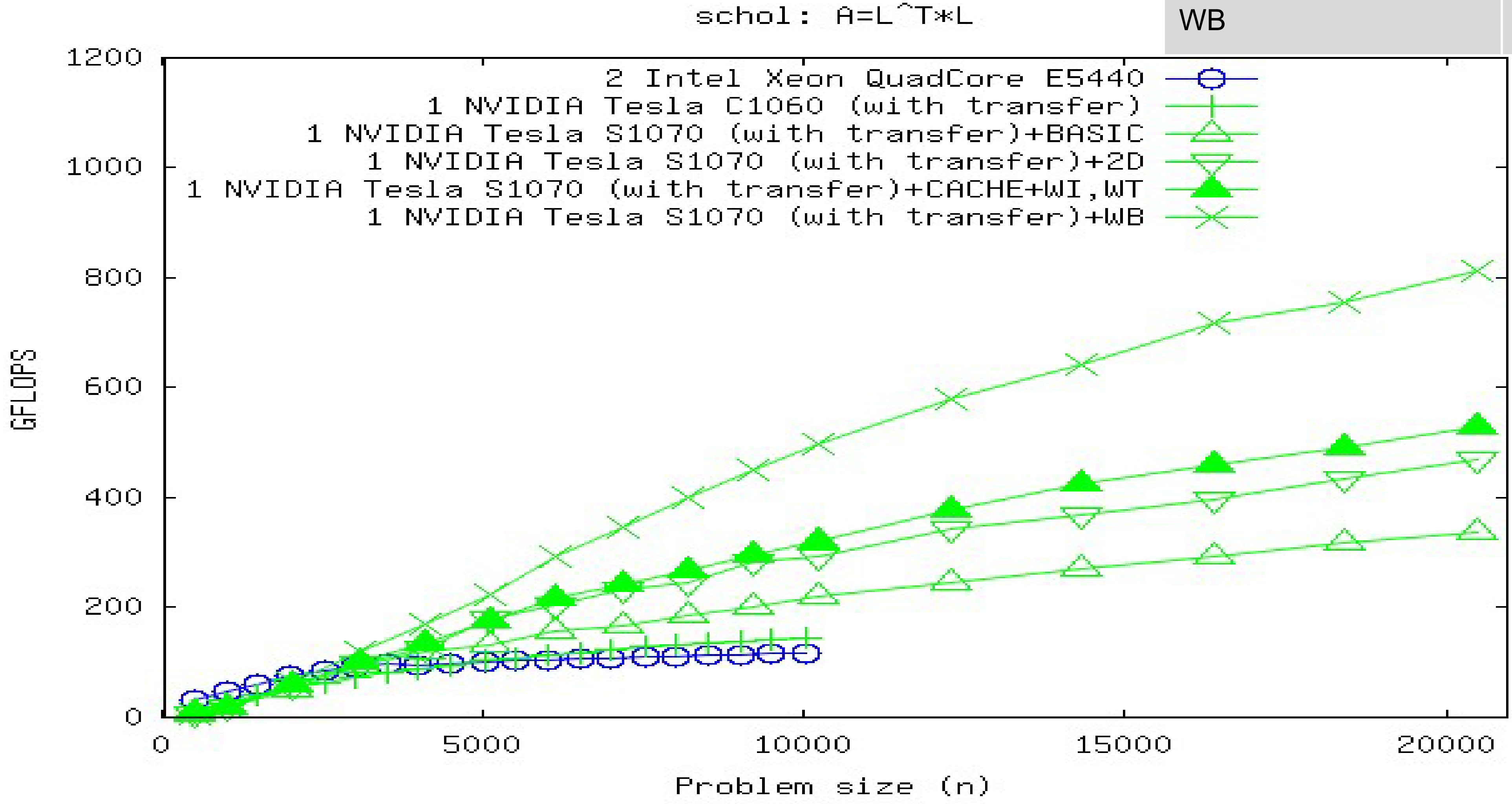
- SuperMatrix (UT@Austin and UJI)
  - Read/written blocks defined implicitly by the operations
  - Only valid for dense linear algebra operations encoded in `libflame`
- SMPSs (BSC) and GPUSs (BSC and UJI)
  - OpenMP-like languages

```
#pragma css task inout(A[b*b])
void Chol(double *A);
```

  - Applicable to task-parallel codes on different platforms: multi-core, multi-GPU, multi-accelerators, Grid,…

Programming multi-GPU platforms

Version	S_p
C1060	1.25
BASIC	2.91
2D	4.04
CACHE+WI,WT	4.55
WB	7.00

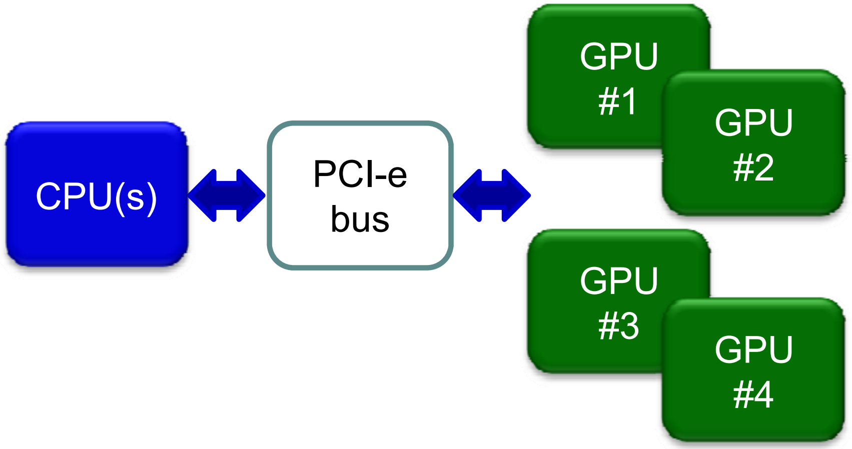


Outline

- Dense linear algebra libraries
- Optimizations for single-GPU platforms
- Programming multi-GPU platforms:
  - Multi-GPU platforms
  - Clusters equipped with GPUs

Programming multi-GPU platforms

- How do we program these?

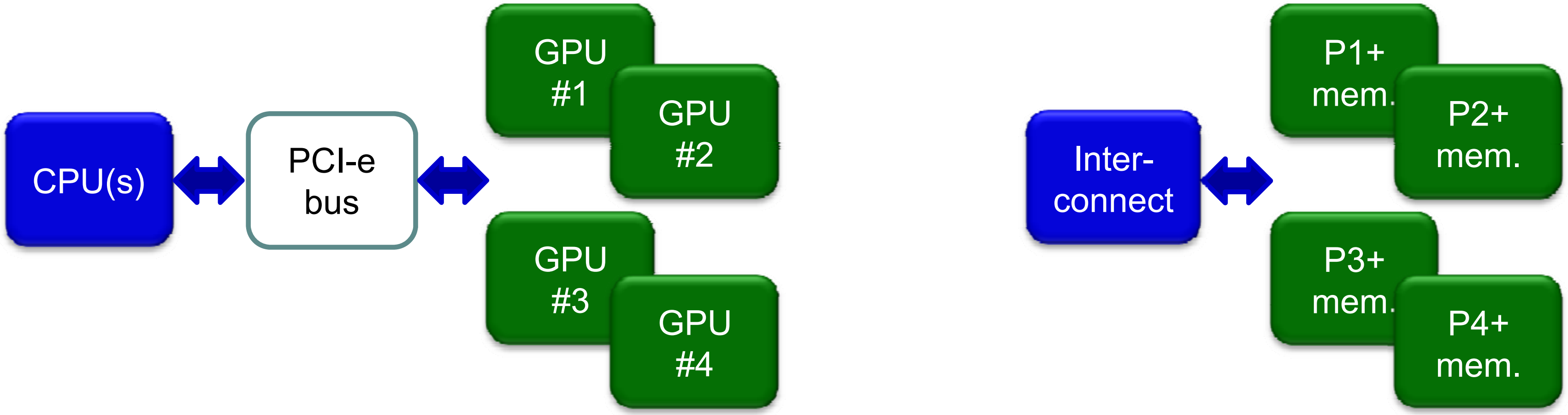


View as a…

- Shared-memory multiprocessor
- Cluster (distributed-memory): valid also for true clusters!

Programming multi-GPU platforms

- Cluster view

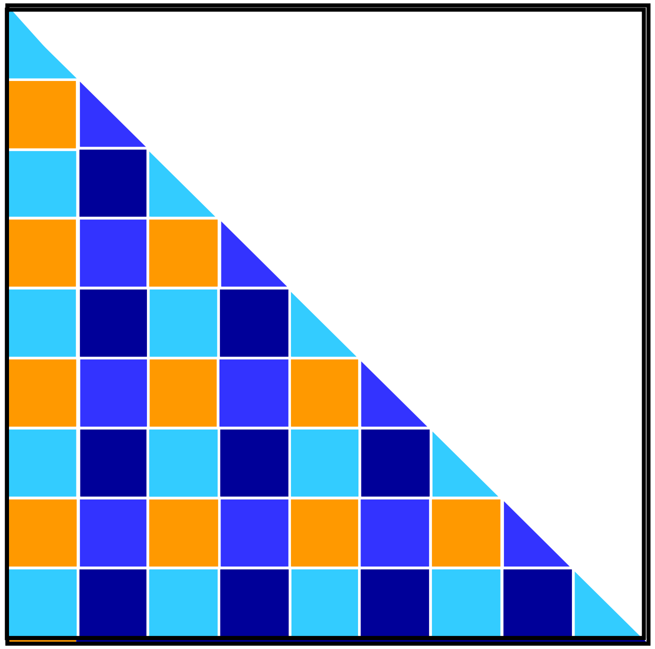


Differences:

- Processes instead of threads
- Message-passing (MPI) application

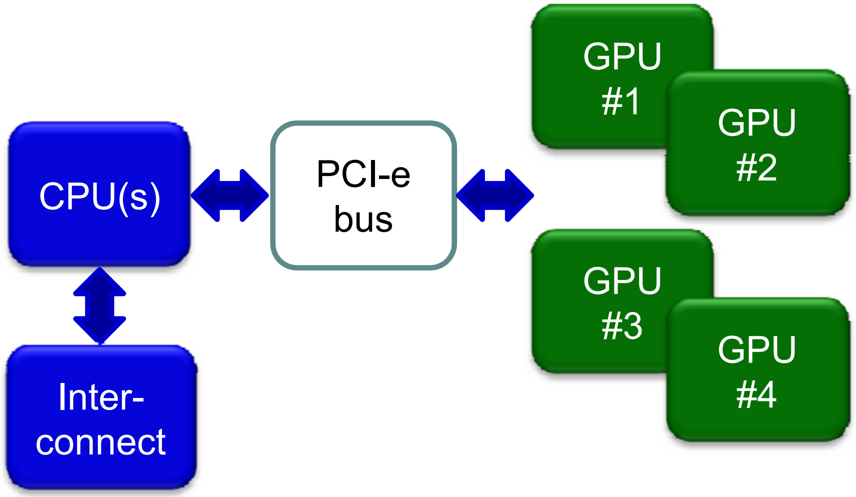
Programming multi-GPU platforms

- Where and where?
  - Static mapping of data and tasks to resources
  - Data transfers embedded in the MPI code



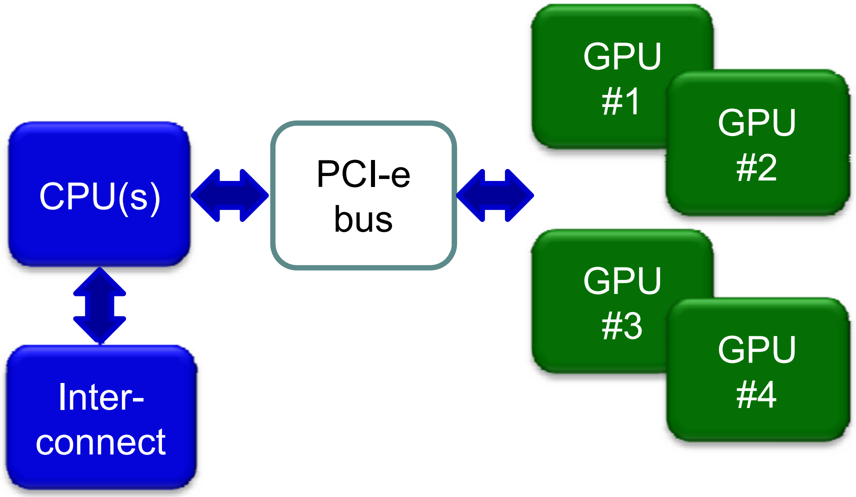
Programming multi-GPU platforms

- Naïve approach: Data in node main memory
  - Before executing a kernel, copy input data to GPU memory
  - After execution, retrieve results back to node main memory
  - Easy to program (wrappers to kernels)
  - Copies linked to kernel execution: $O(n^3)$ transfers between CPU and GPU



Programming multi-GPU platforms

- Alternative approach: Data in GPU memory
  - Before sending a piece of data, retrieve it back to node main memory (compact on the fly)
  - After reception, copy contents to GPU memory
  - Easy to program (wrappers to MPI calls)
  - Copies linked to communication, not kernel execution: *$O(n^2)$* transfers between CPU and GPU

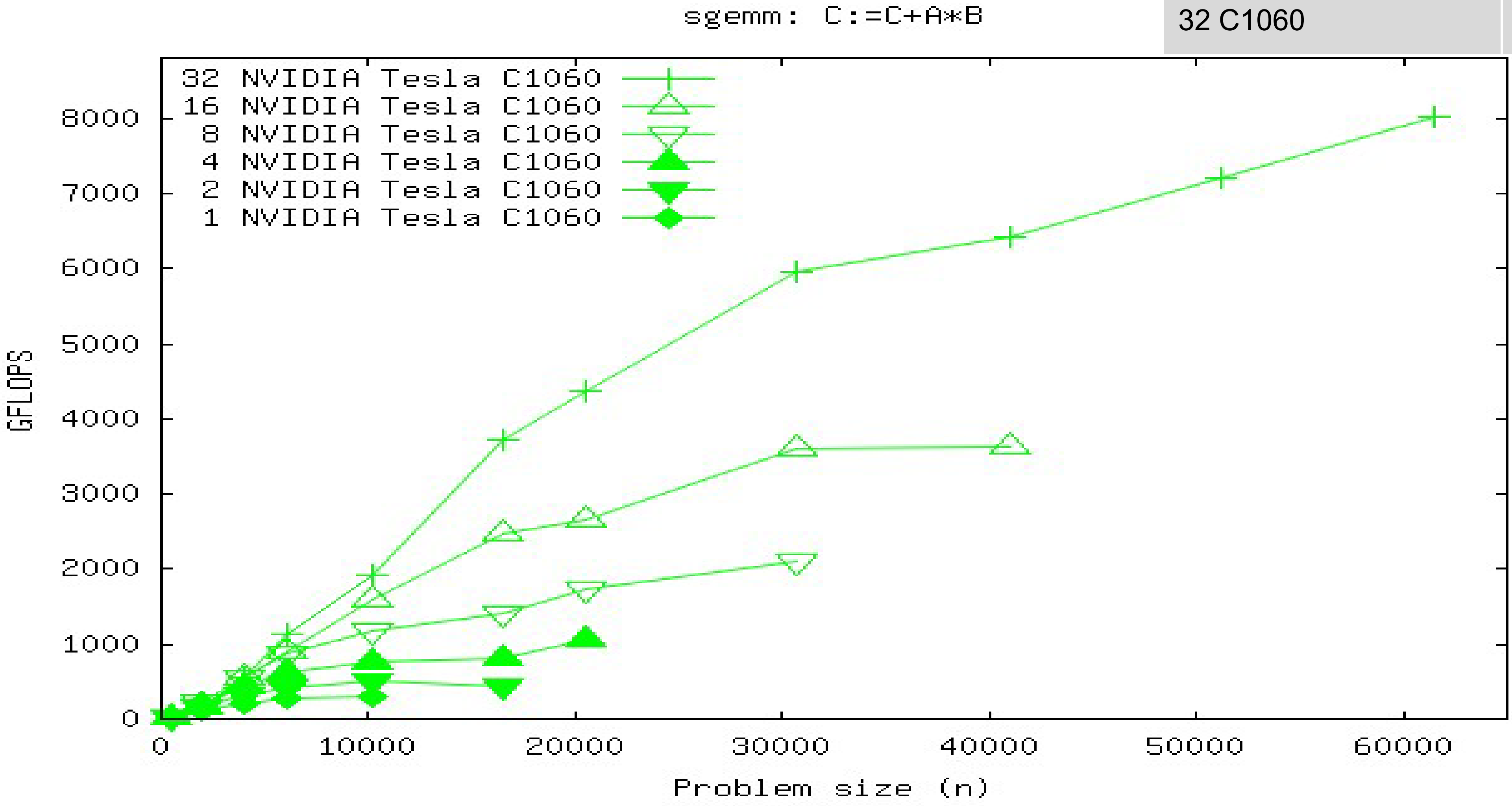


Message-passing implementations

- PLAPACK (UT@Austin)
  - Use of objects (`PLA_Obj`), vectors, matrices, projected vectors, etc., with layout embedded
  - PMB distribution
  - Layered and modular design: all communication is done via copies (`PLA_Copy`) and reductions (`PLA_Reduce`) from one object type to another
- Elemental (Jack Poulson)
  - Based on PLAPACK, but C++
  - Element-wise cyclic data layout

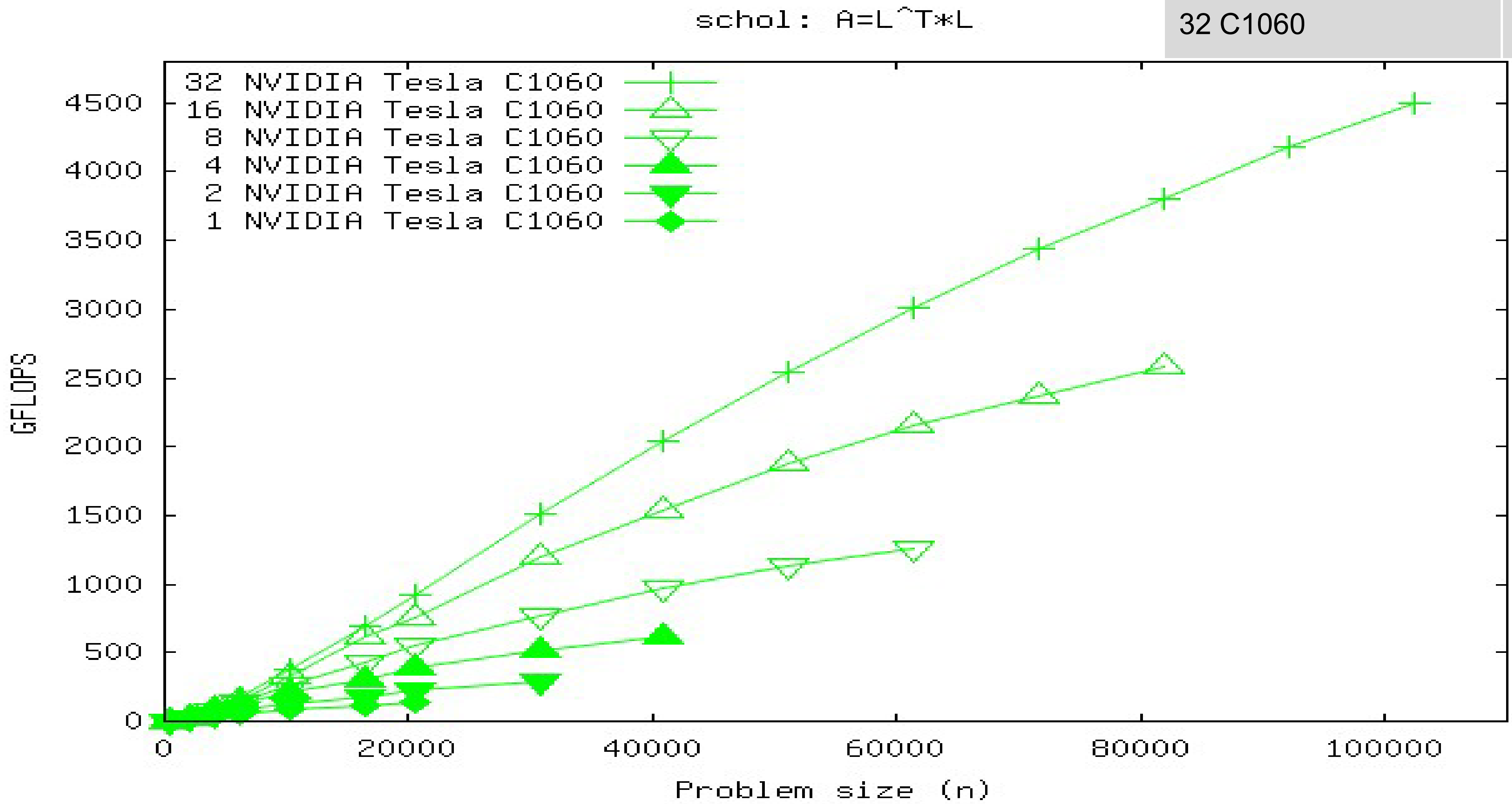
Programming multi-GPU platforms

Version	S_p
2 C1060	1.40
4 C1060	3.45
8 C1060	6.82
16 C1060	11.77
32 C1060	26.01



Programming multi-GPU platforms

Version	S_p
2 C1060	1.40
4 C1060	6.13
8 C1060	8.42
16 C1060	12.54
32 C1060	21.84



Acks. & support

- UJI
  - F. Igual, G. Quintana
- UT
  - The FLAME team
- BSC
  - Computer Sciences Department

More information…

- libflame (UT & UJI)
  - http://www.cs.utexas.edu/users/flame
- GPUSs (BSC & UJI)
  - http://www.bsc.es
  - http://www.bsc.es/plantillaG.php?cat_id=385

Farewell

Thanks for your attention!*

*Hope you enjoyed this as much as Barcelona's beach