Parallel Solver Essentials for Computational Scientists

Part 1: Shared Memory Systems

Karl Rupp

me@karlrupp.net
https://karlrupp.net/
http://github.com/karlrupp/slides
@karlrupp

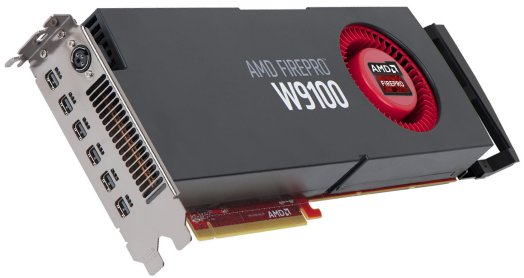
now:
Freelance Scientist

formerly and soon again:
Institute for Microelectronics, TU Wien

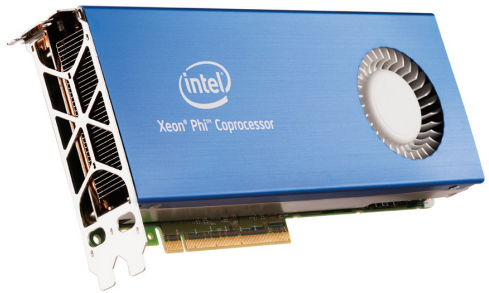
Kazan Federal University
October 19, 2016

Current Many-Core Architectures

- High FLOP/Watt ratio
- High memory bandwidth
- Attached via PCI-Express



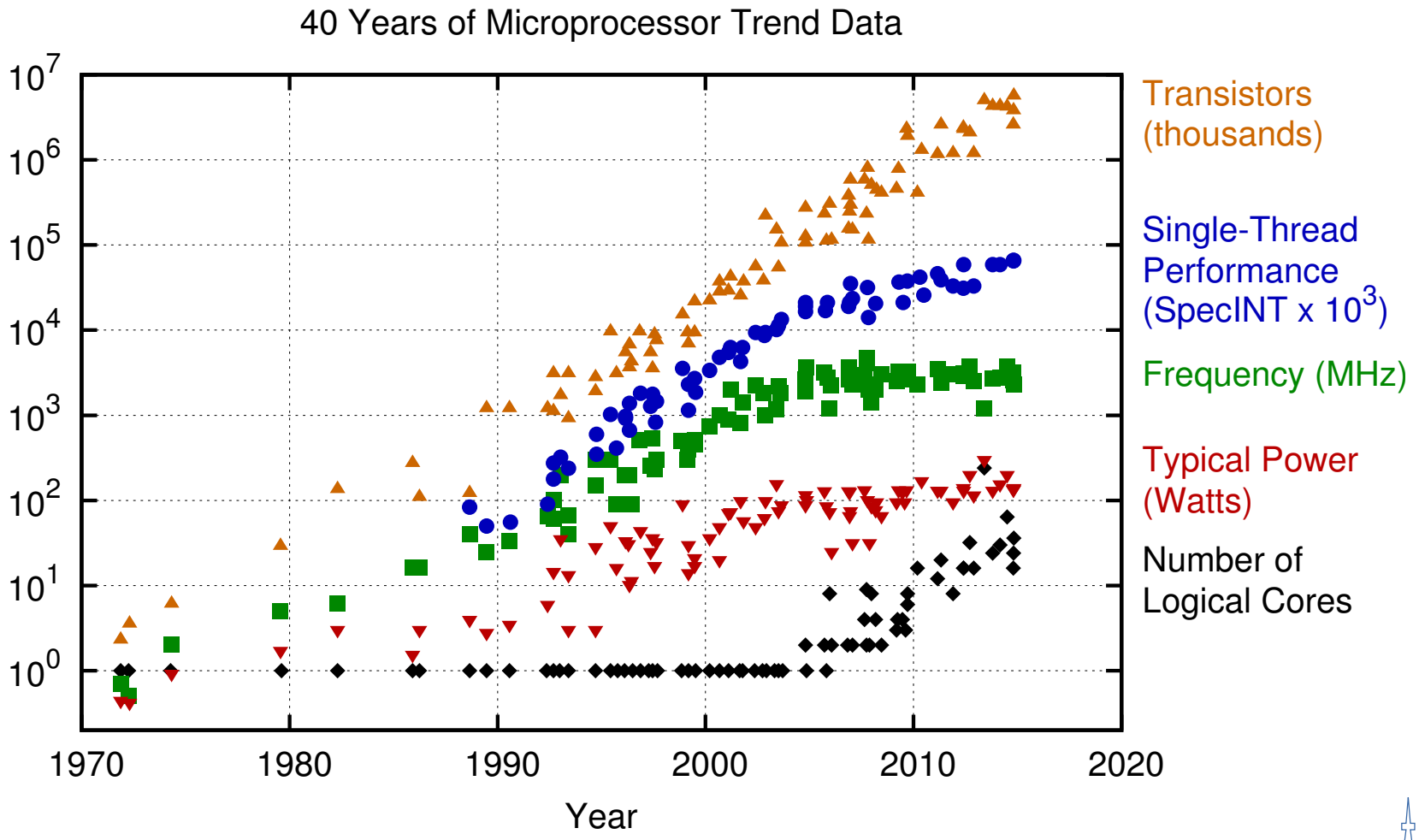
AMD FirePro W9100
320 GB/sec



INTEL Xeon Phi
320 (220?) GB/sec

NVIDIA Tesla K20
250 (208) GB/sec

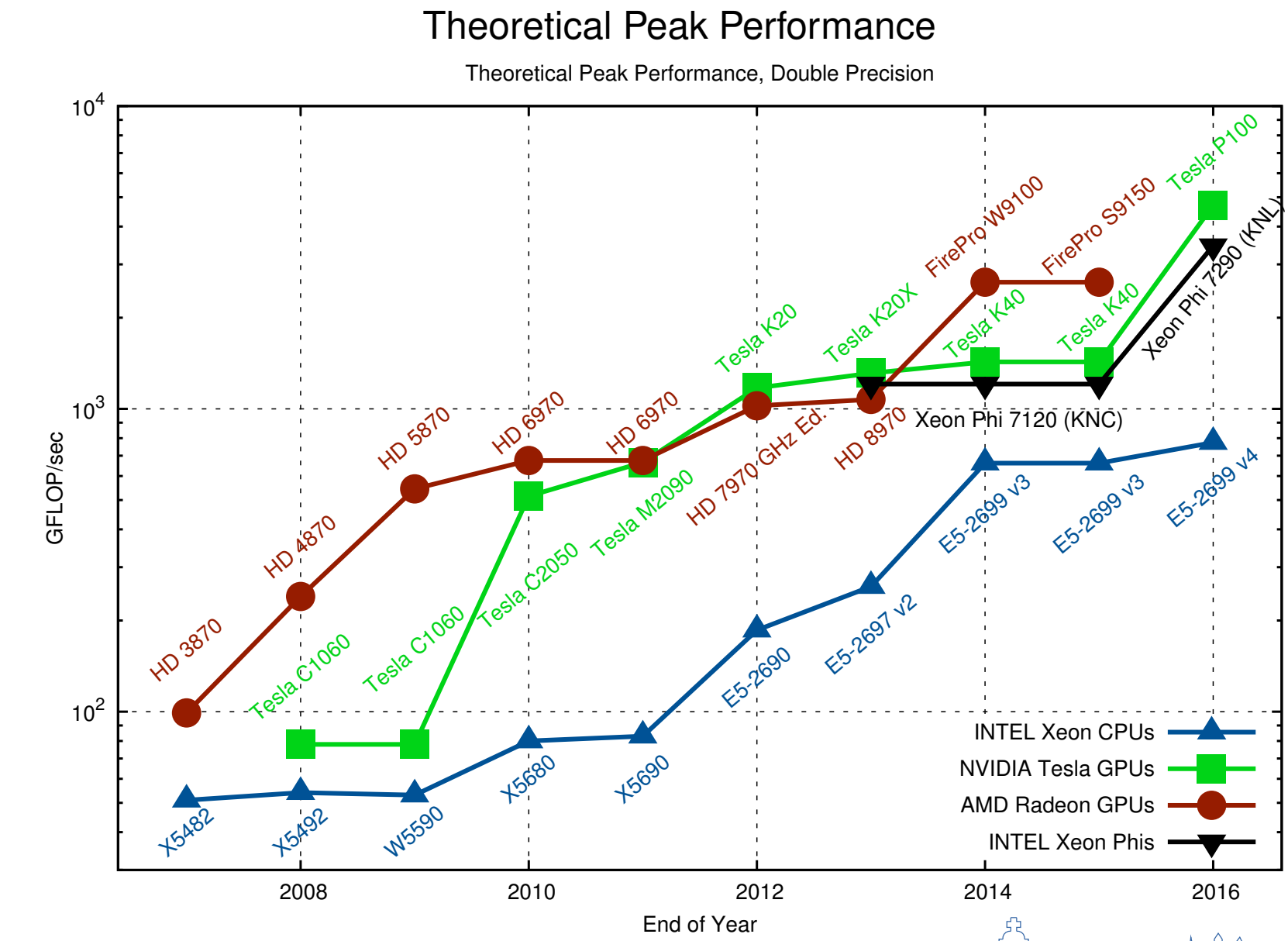
Introduction



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

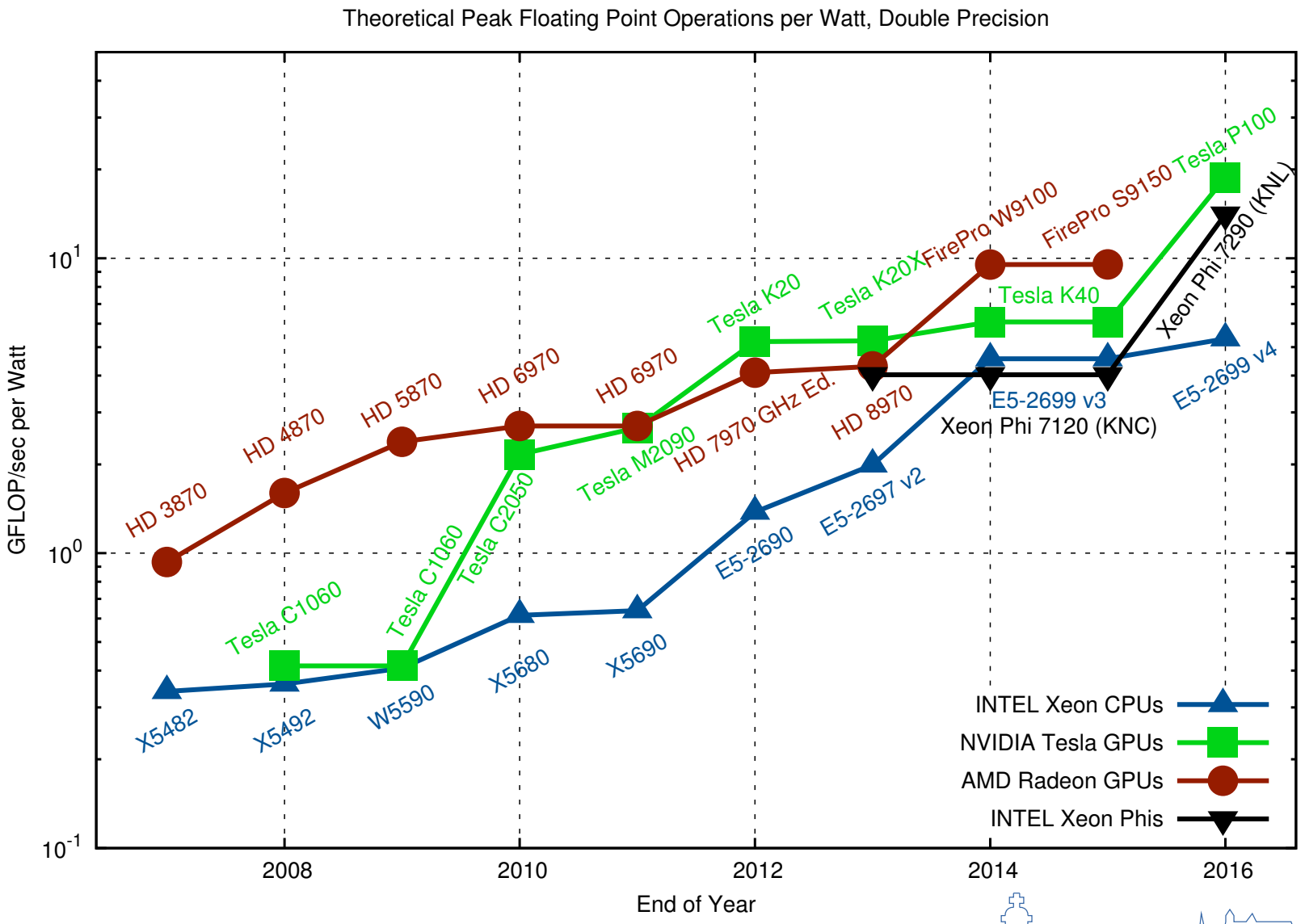
Introduction



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Introduction

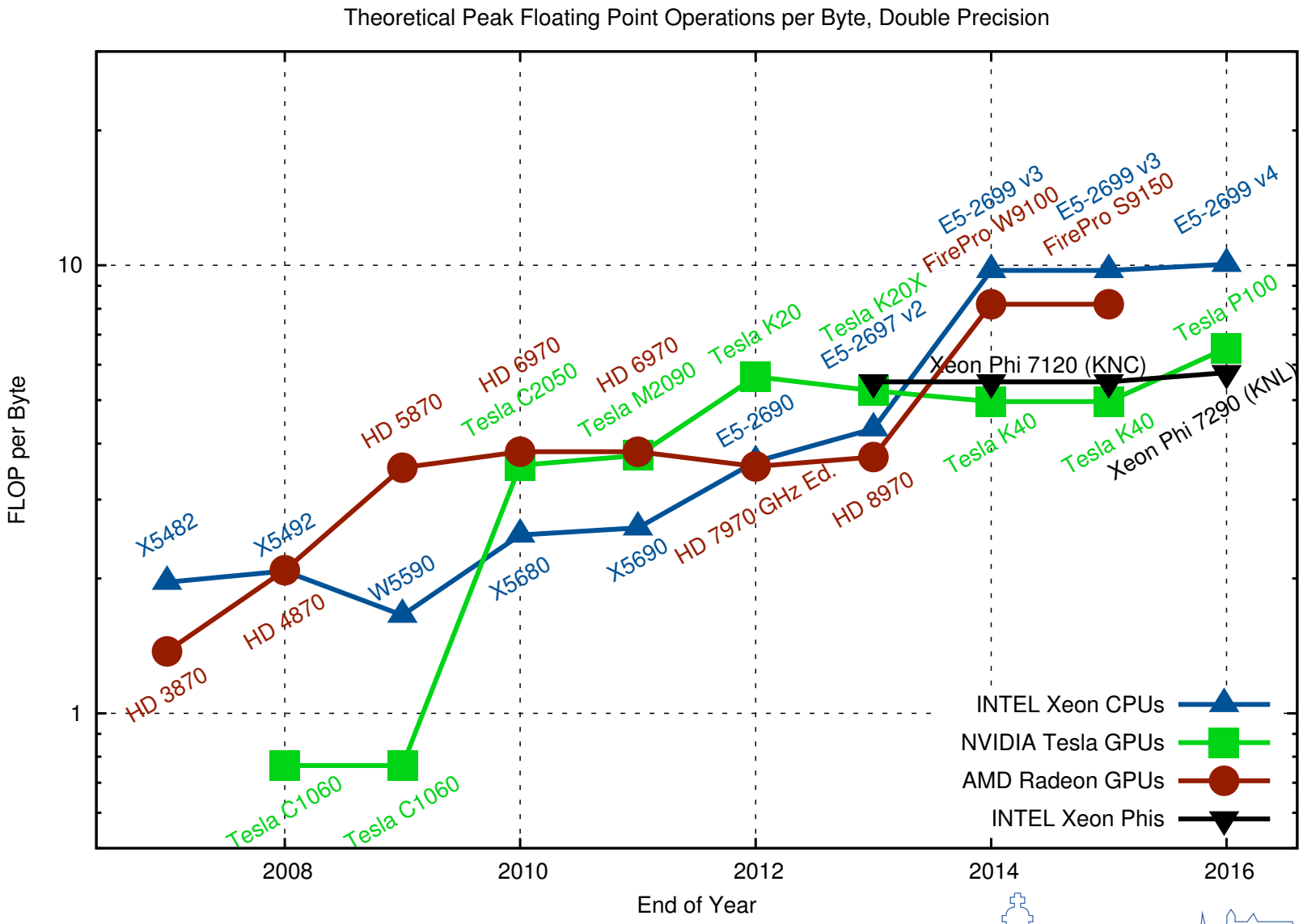
Theoretical Peak Performance per Watt



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Introduction

Theoretical Peak Performance (FLOPs) per Byte of Memory Bandwidth



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

CUDA

About

Initial release in 2007

Proprietary programming model by NVIDIA

C++ with extensions

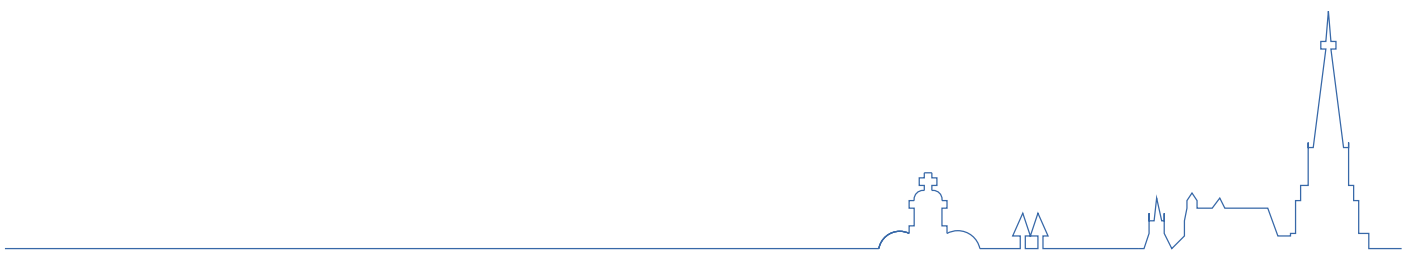
Proprietary compiler extracts GPU kernels

Software Ecosystem

Vendor-tuned libraries: cuBLAS, cuSparse, cuSolver, cuFFT, etc.

Python bindings: pyCUDA

Community projects: CUSP, MAGMA, VexCL, ViennaCL, etc.



Programming in CUDA

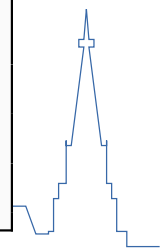
```
void work(double *x, double *y, double *z, int N)
{


  for (size_t i=0; i<N; ++i)
    z[i] = x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  ...

  ...
  work(x, y, z, N); // call kernel
  ...


  free(x);
}
```

Programming in CUDA

```
void work(double *x, double *y, double *z, int N)
{

  #pragma omp parallel for
  for (size_t i=0; i<N; ++i)
    z[i] = x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  ...

  ...
  work(x, y, z, N); // call kernel
  ...


  free(x);
}
```



Programming in CUDA

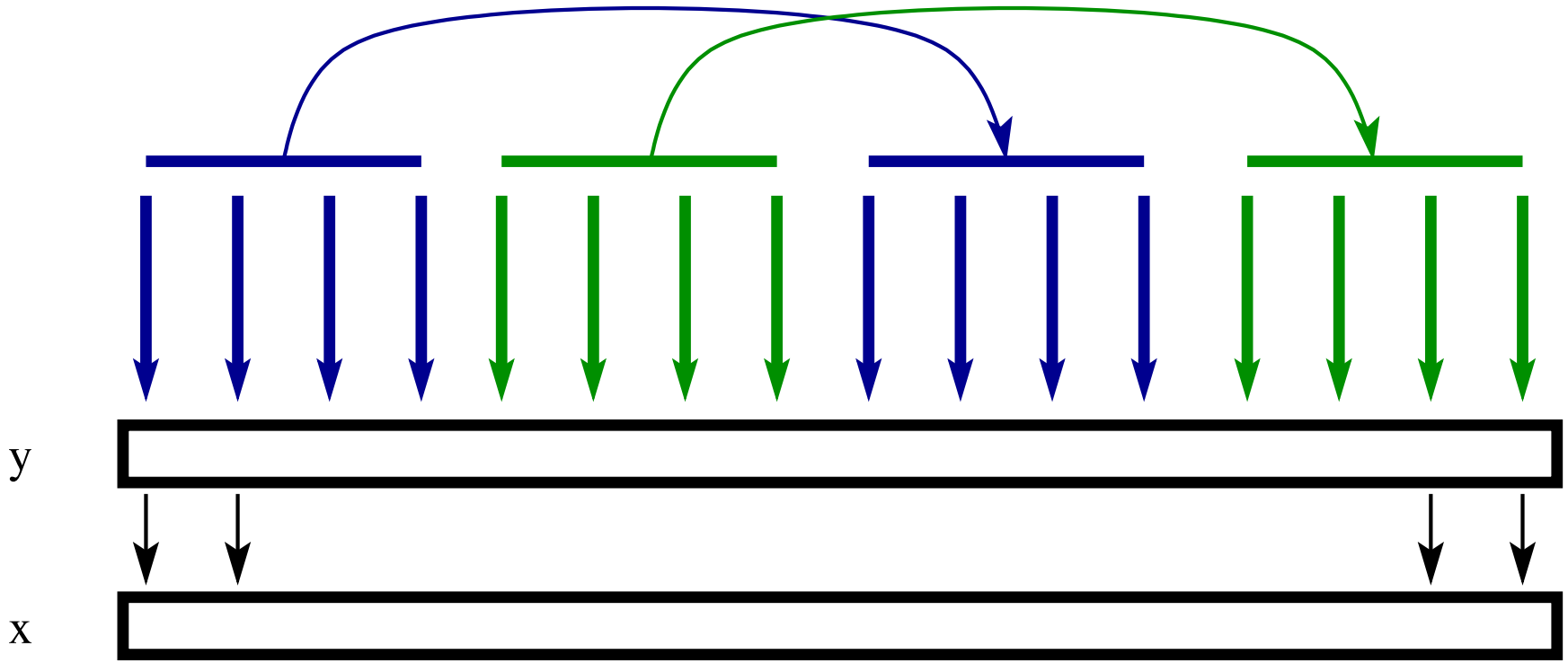
```
void work(double *x, double *y, double *z, int N)
{
  #pragma omp parallel
{ int thread_id = omp_get_thread_num();
  for (size_t i=thread_id; i<N; i += omp_get_num_threads())
    z[i] = x[i] + y[i];
} }
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  ...

  ...
  work(x, y, z, N); // call kernel
  ...


  free(x);
}
```

Programming in CUDA

```
__global__ void work(double *x, double *y, double *z, int N)
{

  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i] = x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  cudaMalloc(&gpu_x, N*sizeof(double));
  cudaMemcpy(gpu_x, x, N*8, cudaMemcpyHostToDevice);
  ...
  work<<<128, 256>>>(x, y, z, N); // call kernel
  ...
  cudaMemcpy(gpu_x, x, N*8, cudaMemcpyDeviceToHost);
  ...
  free(x);
}
```


y
x

Thread Control (1D)

Local ID in block: `threadIdx.x`

Threads per block: `blockDim.x`

ID of block: `blockIdx.x`

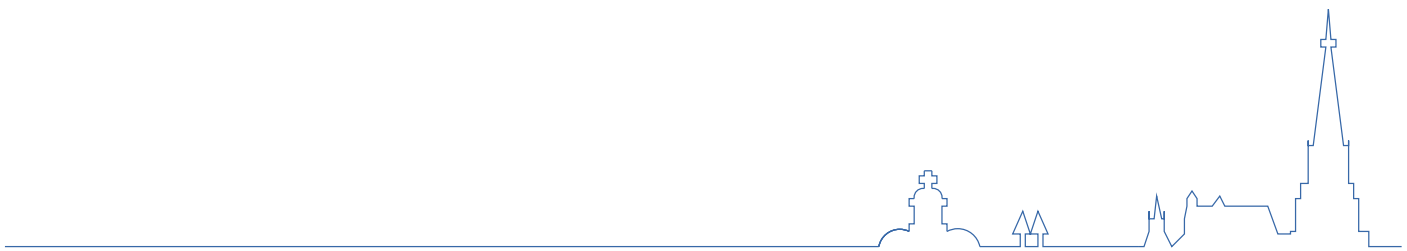
No. of blocks: `gridDim.x`

Recommended Default Values

Typical block size: 256 or 512

Typical number of blocks: 256

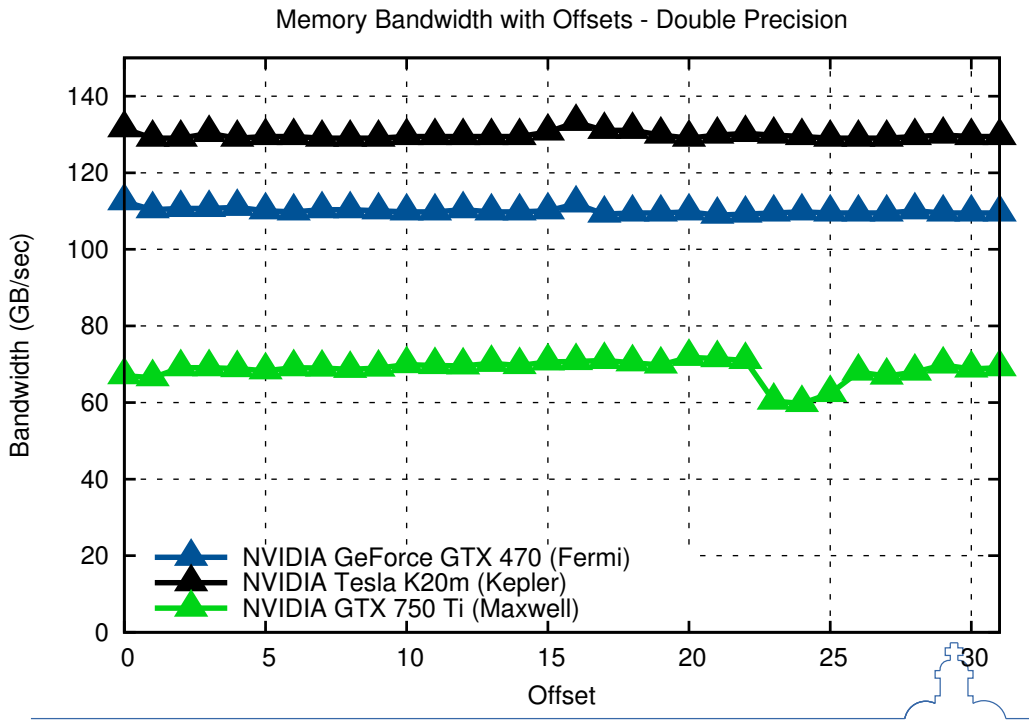
At least $10\,000$ logical threads recommended



CUDA Example

Offset Memory Access

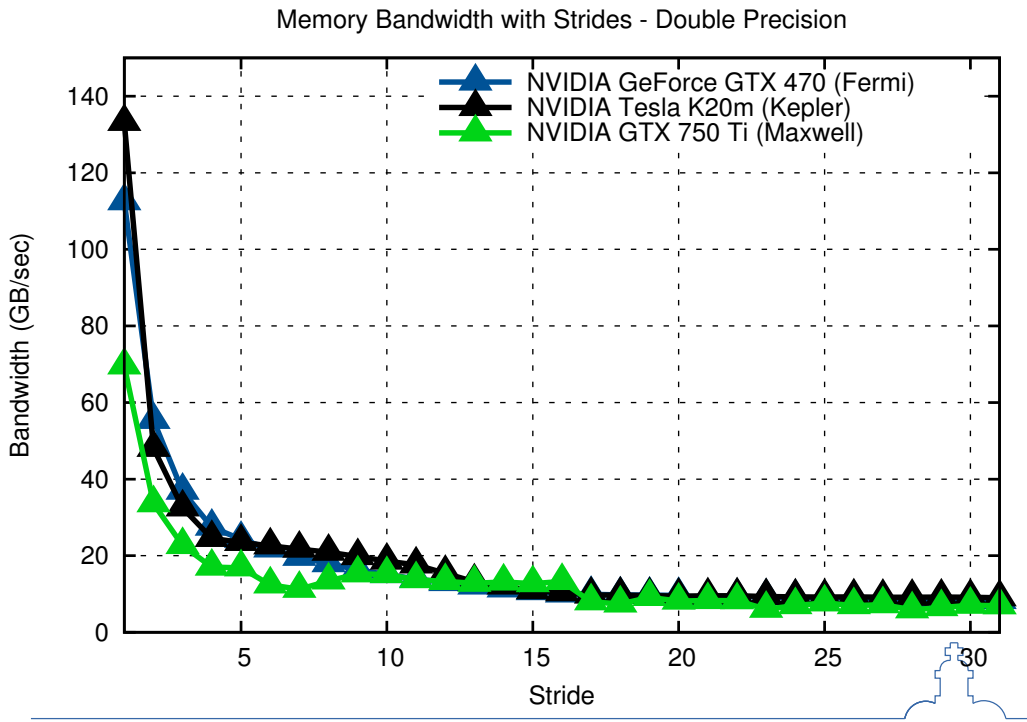
```
__global__
void work(double *x, double *y, double *z, int N, int k)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i+k] = x[i+k] + y[i+k];
}
```



CUDA Example

Strided Memory Access

```
__global__
void work(double *x, double *y, double *z, int N, int k)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i*k] = x[i*k] + y[i*k];
}
```

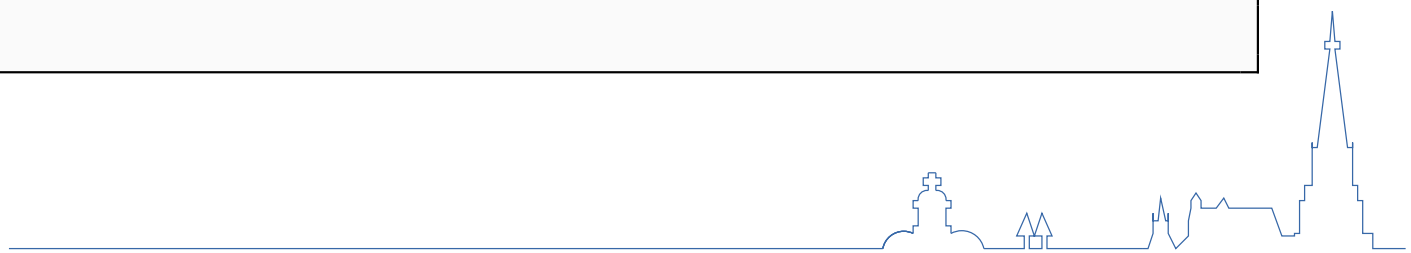


CUDA Example

Strided Memory Access

Array of structs problematic

```
typedef struct particle
{
  double pos_x; double pos_y; double pos_z;
  double vel_x; double vel_y; double vel_z;
  double mass;
} Particle;

__global__
void increase_mass(Particle *particles, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i=thread_id; i<N; i += blockDim.x * gridDim.x)
    particles[i].mass *= 2.0;
}
```

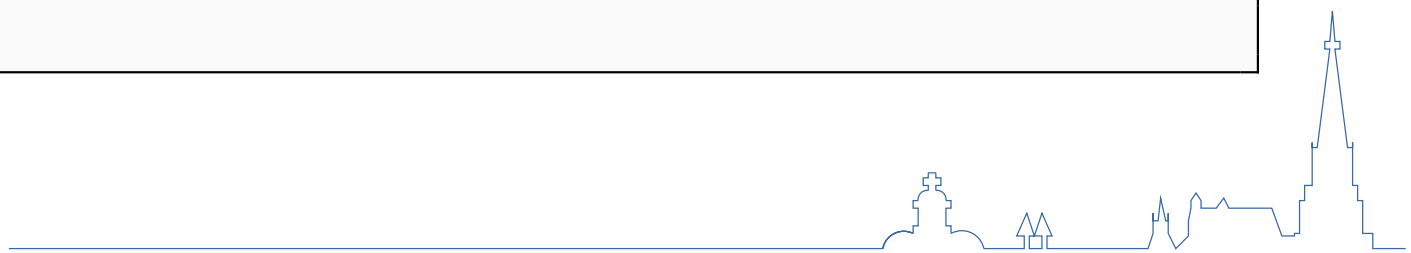


CUDA Example

Strided Memory Access

Workaround: Structure of Arrays

```
typedef struct particles
{
  double *pos_x; double *pos_y; double *pos_z;
  double *vel_x; double *vel_y; double *vel_z;
  double *mass;
} Particle;

__global__
void increase_mass(Particle *particles, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i=thread_id; i<N; i += blockDim.x * gridDim.x)
    particles.mass[i] *= 2.0;
}
```



Parallel Primitives

Reductions

Use N values to compute 1 result value

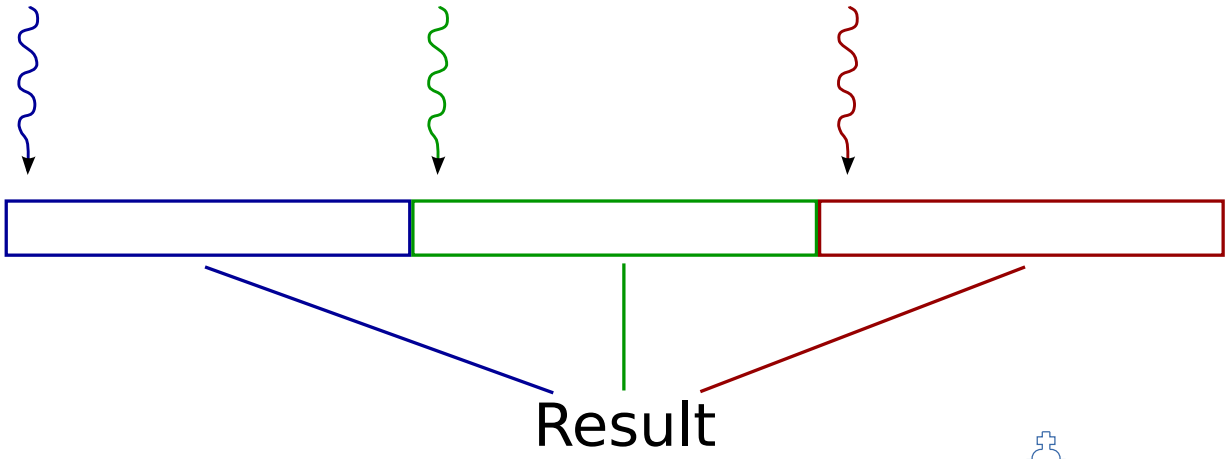
Examples: Dot-products, vector norms, etc.

Reductions with Few Threads

Decompose N into chunks for each thread

Compute chunks in parallel

Merge results with single thread

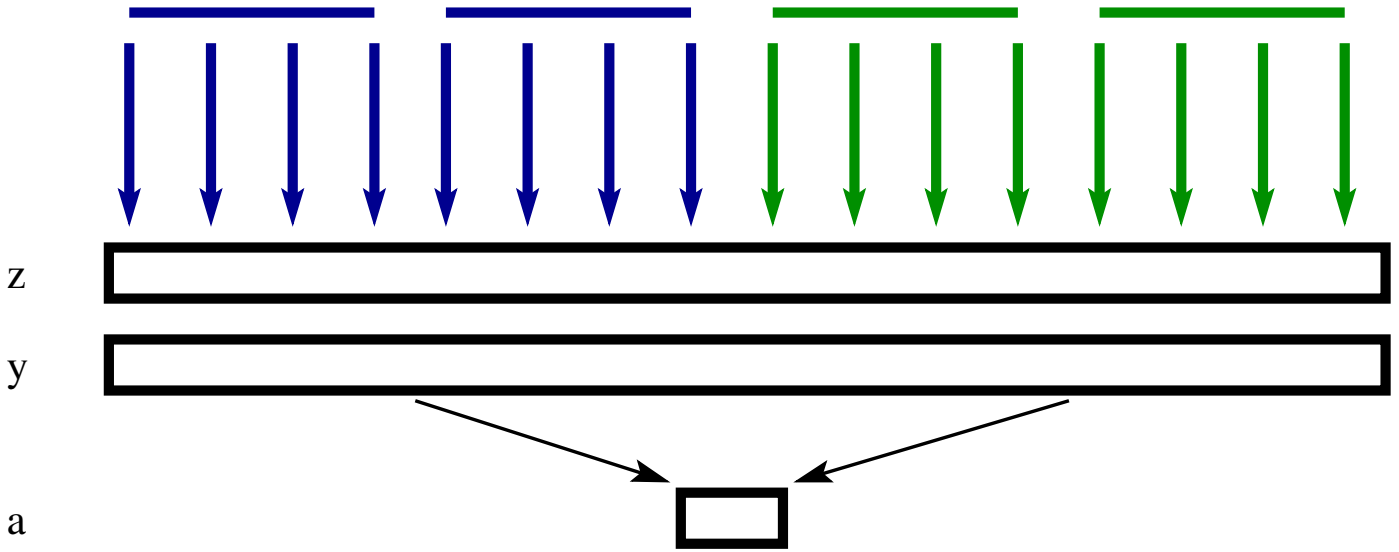


Reductions with Many Threads

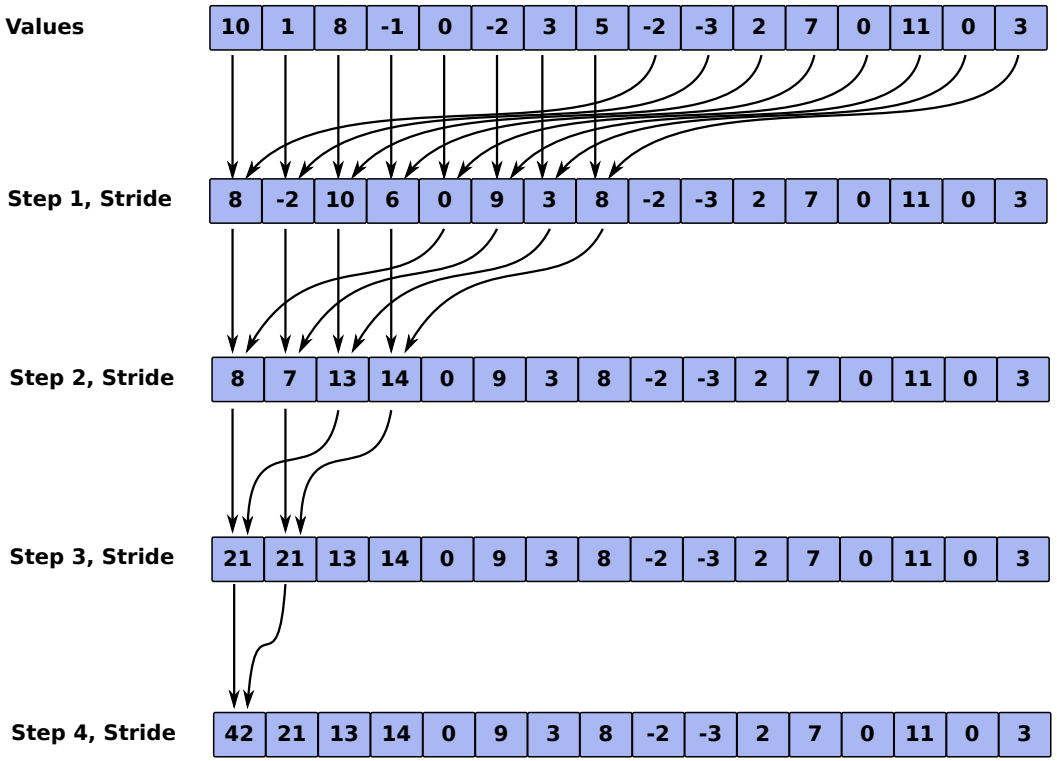
Decompose N into chunks for each workgroup

Use fast on-chip synchronization within each workgroup

Sum result for each workgroup separately



Reductions with Many Threads




```
shared_m[threadIdx.x] = thread_sum;
for (int stride = blockDim.x/2; stride>0; stride/=2) {
  __syncthreads();
  if (threadIdx.x < stride)
    shared_m[threadIdx.x] += shared_m[threadIdx.x+stride];
}
```

Parallel Primitives

Prefix Sum

Inclusive: Determine $y_i = \sum_{k=1}^{i} x_k$

Exclusive: Determine $y_i = \sum_{k=1}^{i-1} x_k$, $y_1 = 0$

Example

x: 4, 3, 6, 5, 4, 7, 4, 4, 4

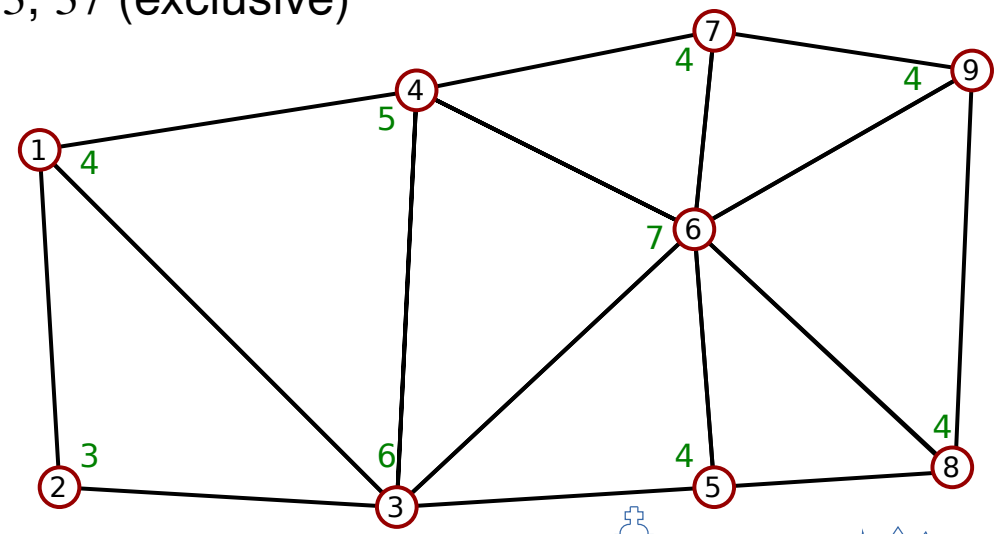
y: 4, 7, 13, 18, 22, 29, 33, 37, 41 (inclusive)

y: 0, 4, 7, 13, 18, 22, 29, 33, 37 (exclusive)

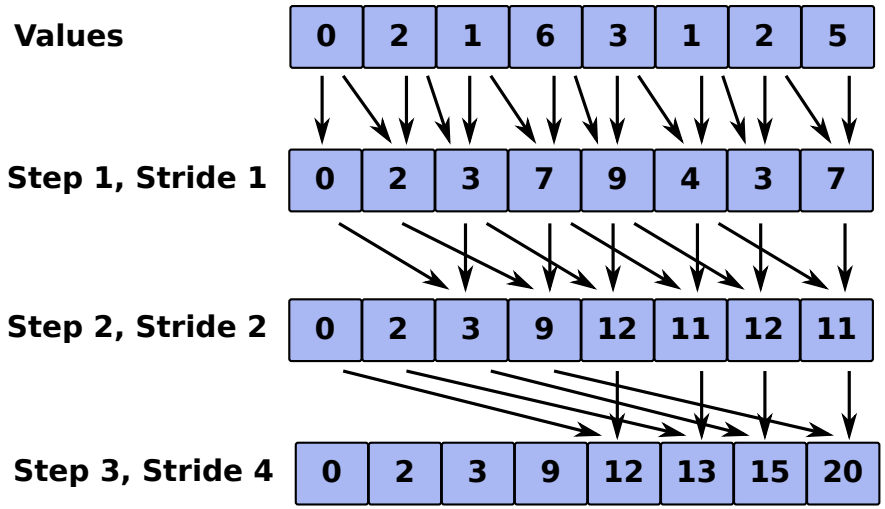
Applications

Sparse matrix setup

Graph algorithms



Prefix Sum Implementation




```
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
  __syncthreads();
  shared_buffer[threadIdx.x] = my_value;
  __syncthreads();
  if (threadIdx.x >= stride)
    my_value += shared_buffer[threadIdx.x - stride];
}
__syncthreads();
shared_buffer[threadIdx.x] = my_value;
```

Parallel Primitives

Other Parallel Primitives

Sort

Gather and Scatter

Load to shared memory and work there

etc.

GPU-Accelerated Software Libraries

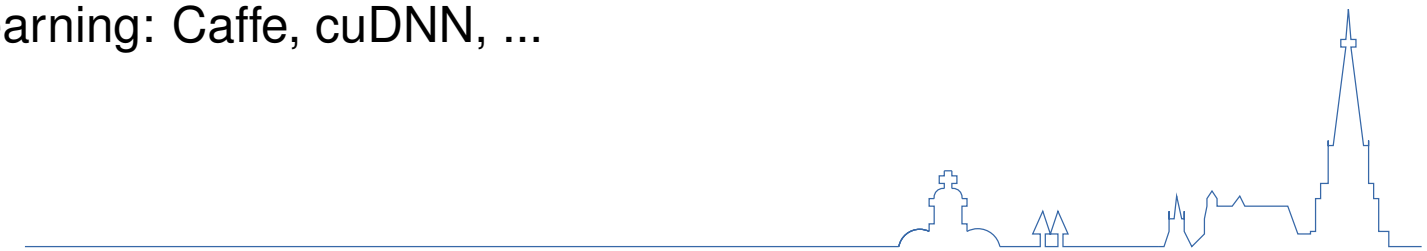
Linear Algebra: ViennaCL, MAGMA, CUSP, VexCL, ...

Solvers: ViennaCL, MAGMA, cuSolver, Paralution, clAMG, ...

FFT: cuFFT, clFFT, FFTW, ...

Primitives: VexCL, Boost.Compute, ...

Machine Learning: Caffe, cuDNN, ...



Overview

Pipelined CG

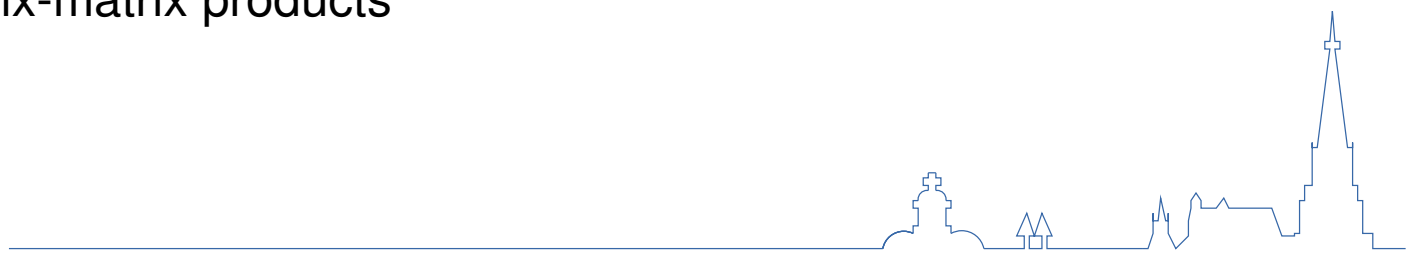
- Merge global reductions
- Kernel fusion

Parallel Incomplete LU Factorizations

- Level scheduling
- Nonlinear relaxation

Algebraic Multigrid

- Parallel aggregation
- Sparse matrix-matrix products



Performance Modeling: Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

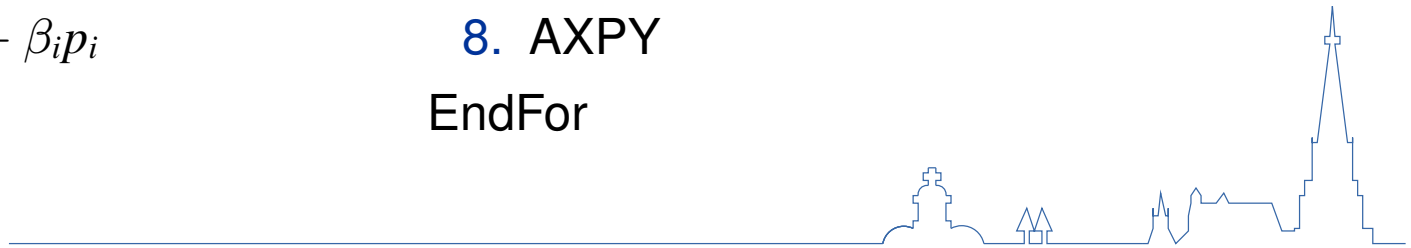
EndFor

BLAS-based Implementation

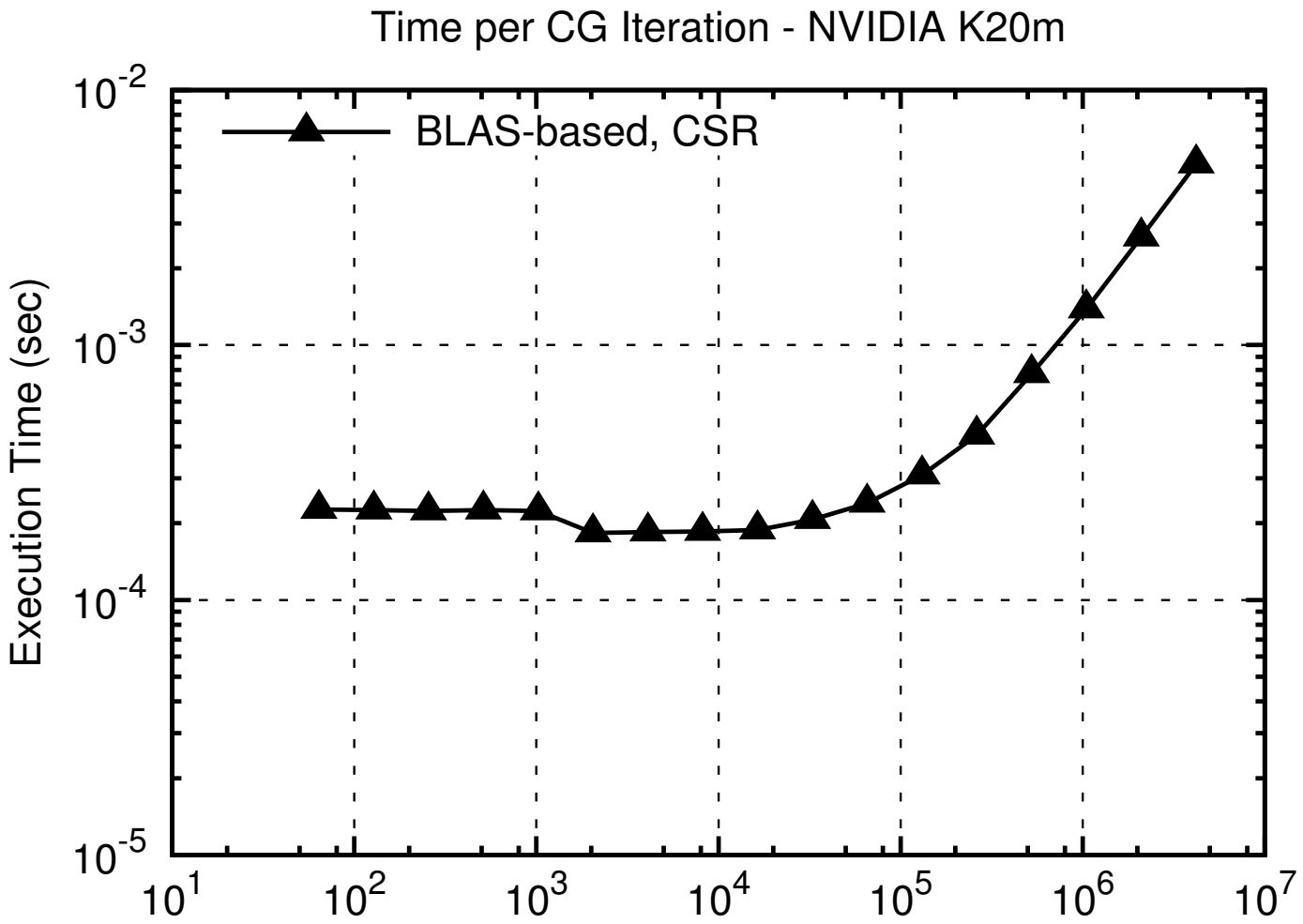
-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV ← No caching of Ap_i
2. DOT ← Global sync!
3. -
4. AXPY
5. AXPY ← No caching of r_{i+1}
6. DOT ← Global sync!
7. -
8. AXPY

EndFor



Performance Modeling: Conjugate Gradients



(Poisson, 2D, Finite Differences)

Performance Modeling: Conjugate Gradients

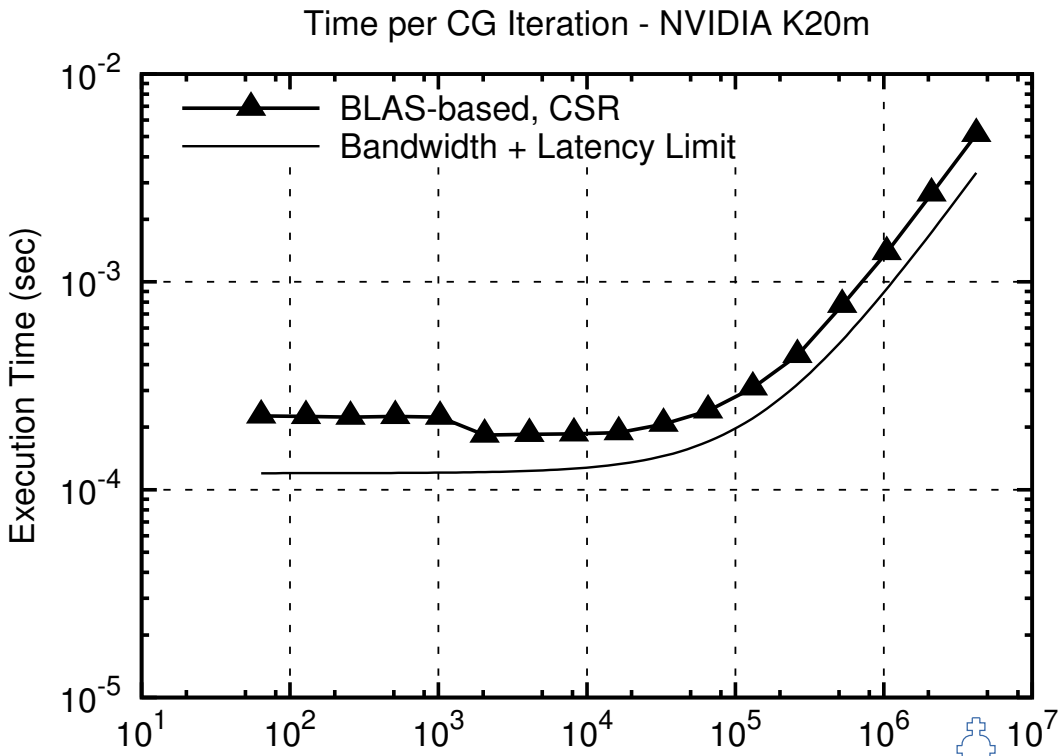
Performance Modelling

6 Kernel Launches (plus two for reductions)

Two device to host data reads from dot products

Model SpMV as seven vector accesses (5-point stencil)

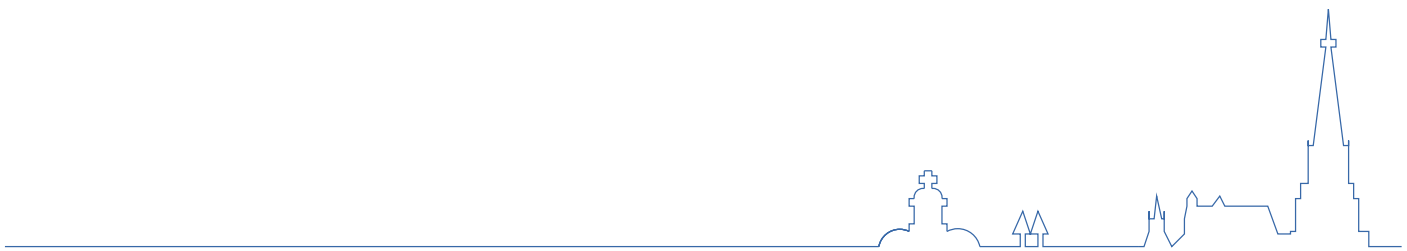
$$T(N) = 8 \times 10^{-6} + 2 \times 2 \times 10^{-6} + (7 + 2 + 3 + 3 + 2 + 3) \times 8 \times N/\text{Bandwidth}$$



Performance Modeling: Conjugate Gradient Optimizations

Optimization: Rearrange the algorithm

- Remove unnecessary reads
- Remove unnecessary synchronizations
- Use custom kernels instead of standard BLAS



Performance Modeling: Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

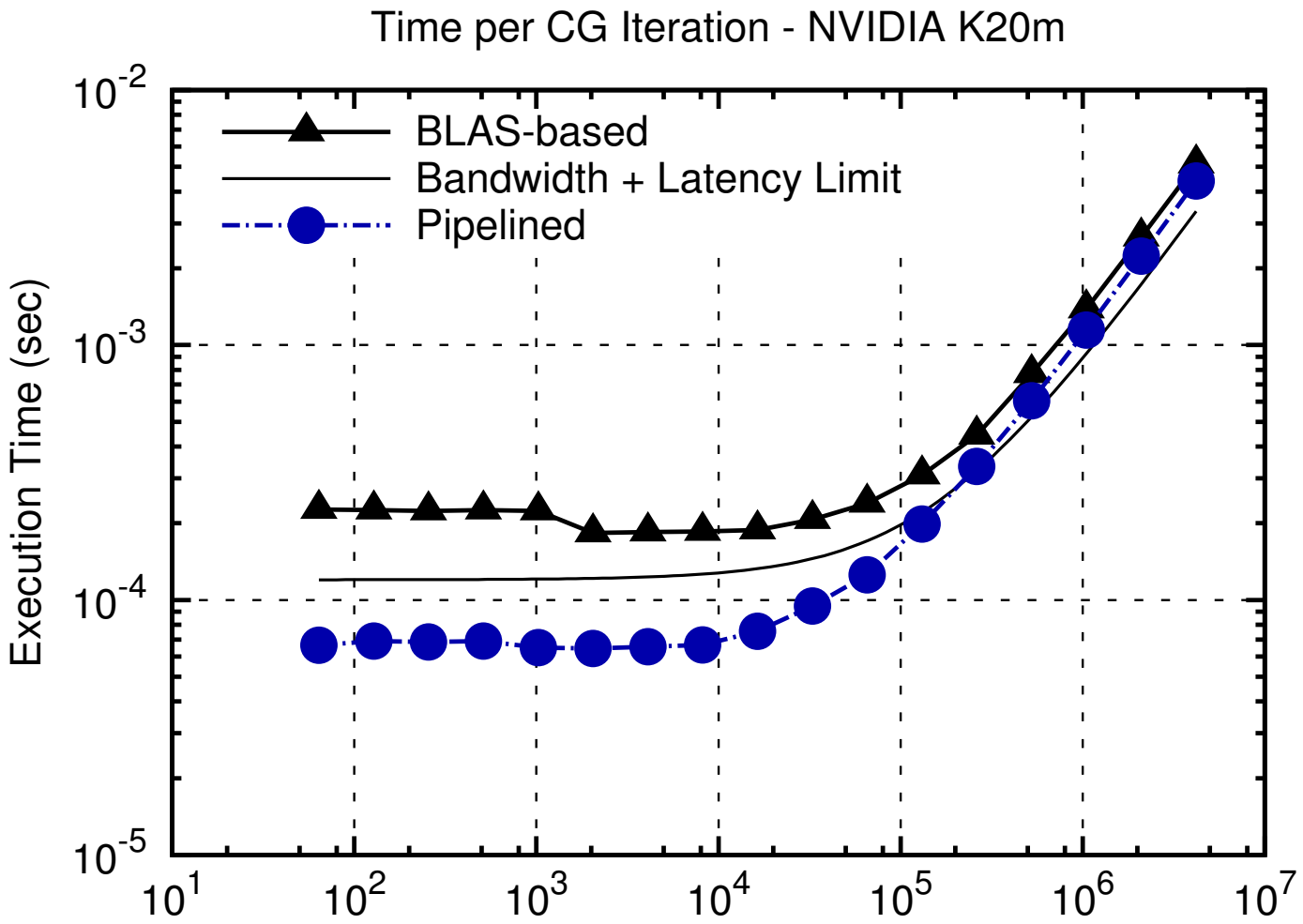
Pipelined CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

EndFor

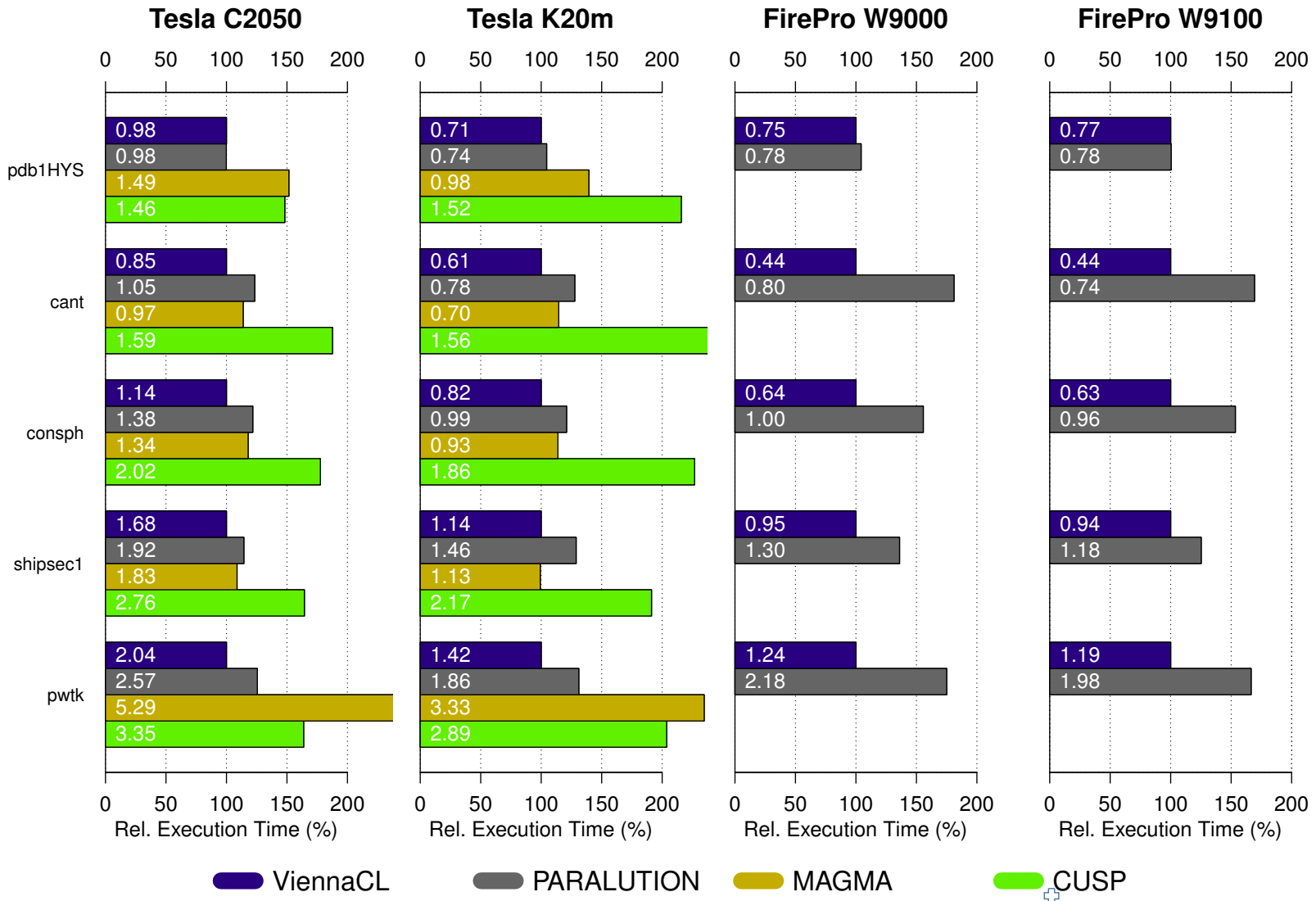
Performance Modeling: Conjugate Gradients



(Poisson, 2D, Finite Differences)

Performance Modeling: Conjugate Gradients

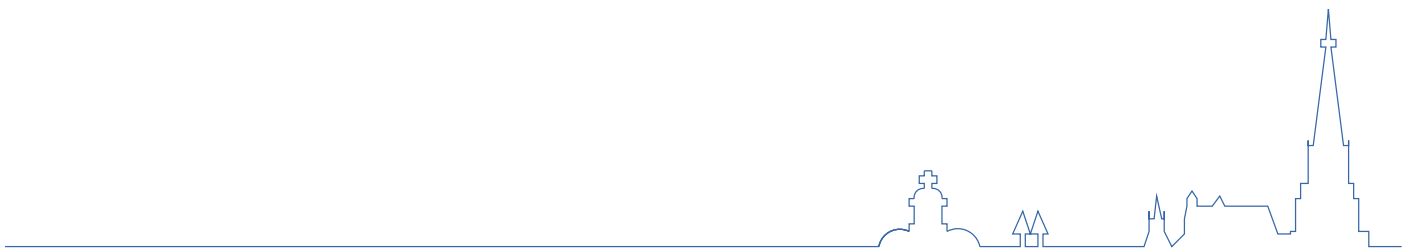
Benefits of Pipelining also for Large Matrices



Content

Parallel Incomplete LU Factorizations

- Level scheduling
- Nonlinear relaxation



Parallel ILU

ILU - Basic Idea

Factor sparse matrix $A \approx \tilde{L}\tilde{U}$

$\tilde{L}$ and $\tilde{U}$ sparse, triangular

ILU0: Pattern of $\tilde{L}$, $\tilde{U}$ equal to A

ILUT: Keep k elements per row

Solver Cycle Phase

Residual correction $\tilde{L}\tilde{U}x = z$

Forward solve $\tilde{L}y = z$

Backward solve $\tilde{U}x = y$

Little parallelism in general

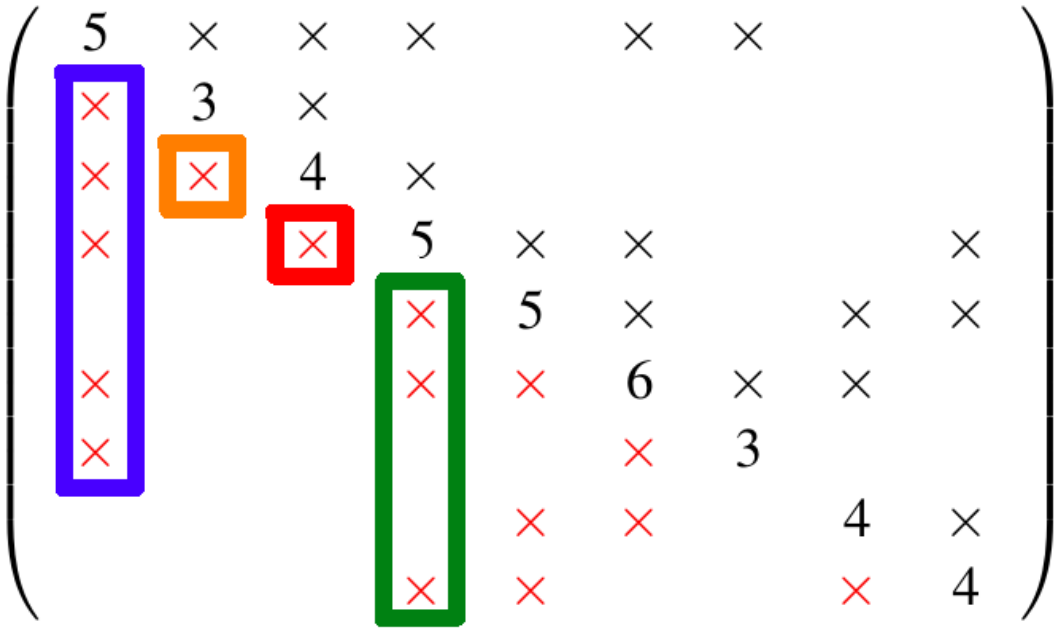
$$\begin{pmatrix}
5 & \times & \times & \times & & \times & \times & & & \\
\times & 3 & \times & & & & & & & \\
\times & \times & 4 & \times & & & & & & \\
\times & & \times & 5 & \times & \times & & & & \times \\
 & & & \times & 5 & \times & & \times & & \times \\
\times & & & \times & \times & 6 & \times & \times & & \\
\times & & & & & \times & 3 & & & \\
 & & & & \times & \times & & 4 & & \times \\
 & & & \times & \times & & & \times & & 4
\end{pmatrix}$$

ILU Level Scheduling

Build dependency graph

Substitute as many entries as possible simultaneously

Trade-off: Each step vs. multiple steps in a single kernel

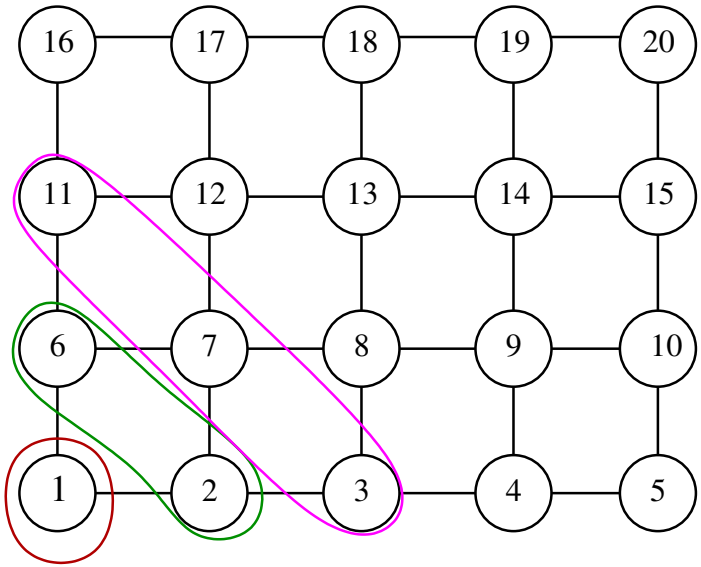


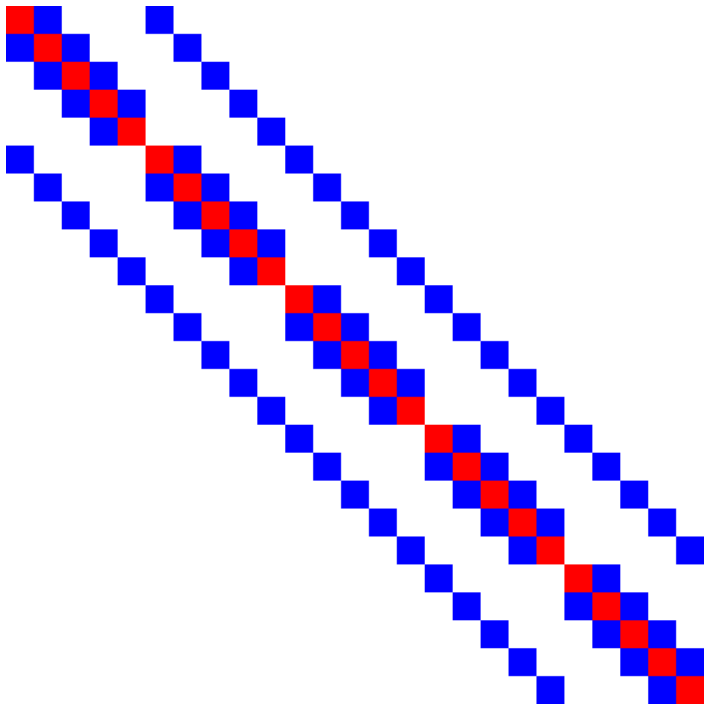
ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d



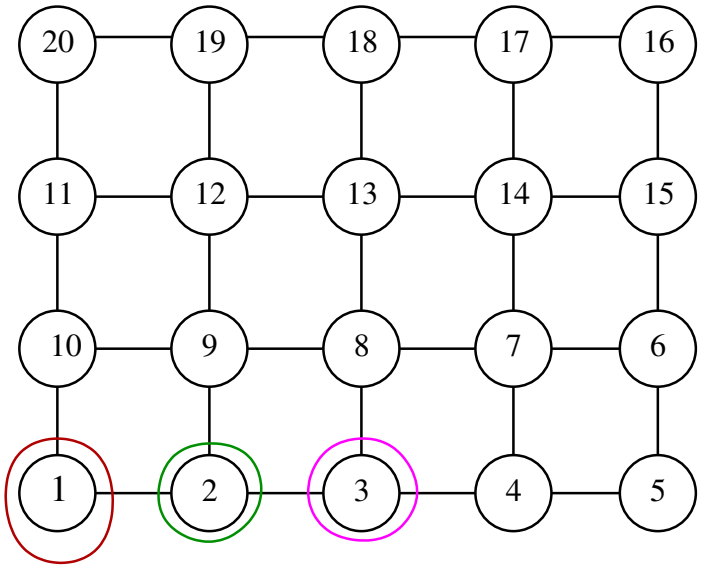


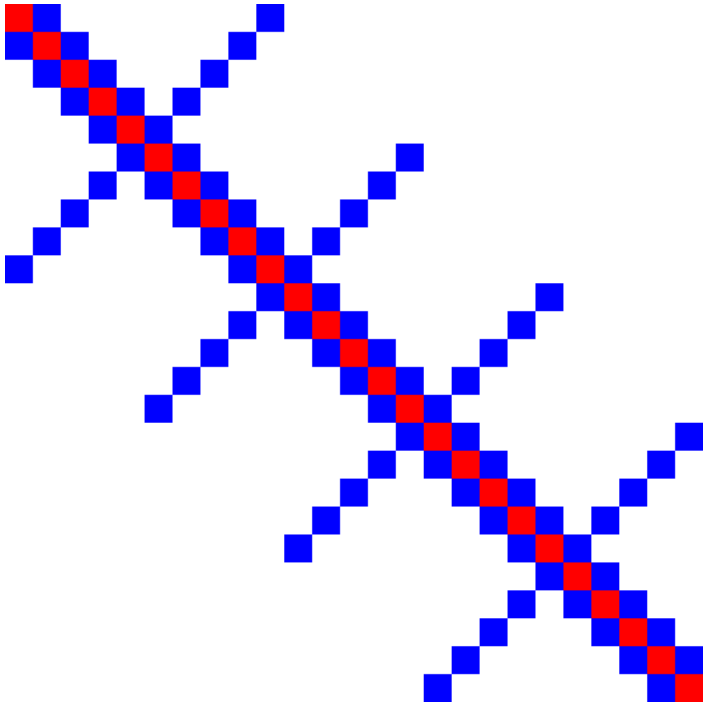
ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d



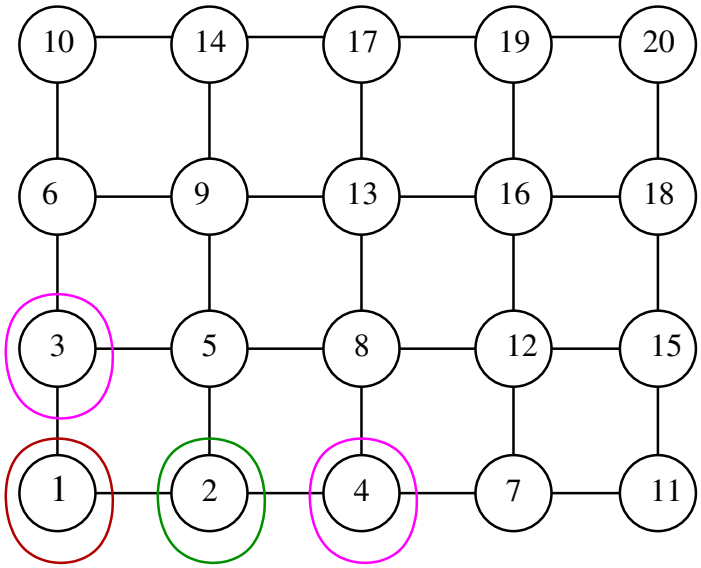


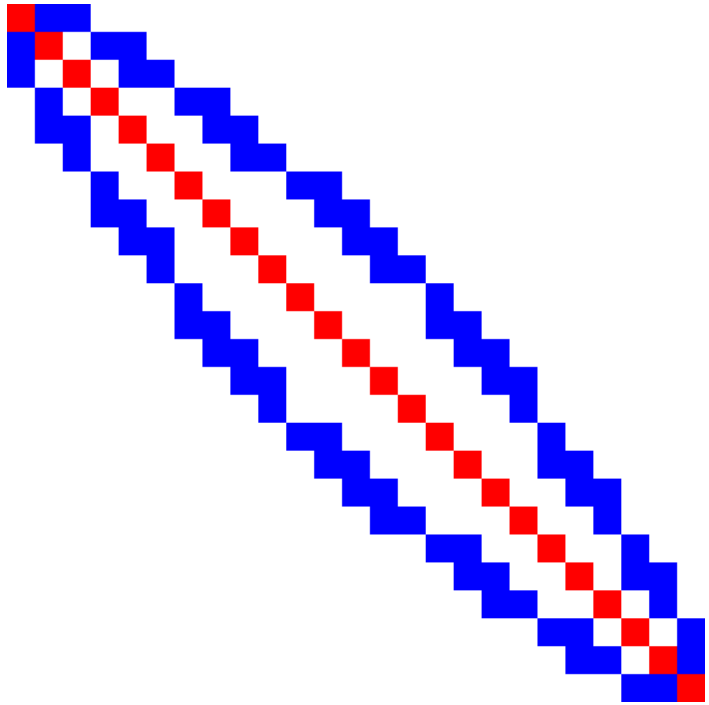
ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d



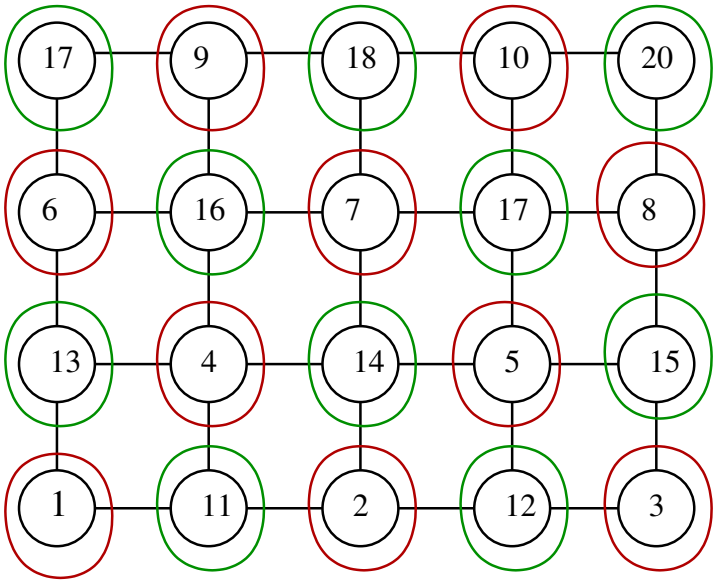


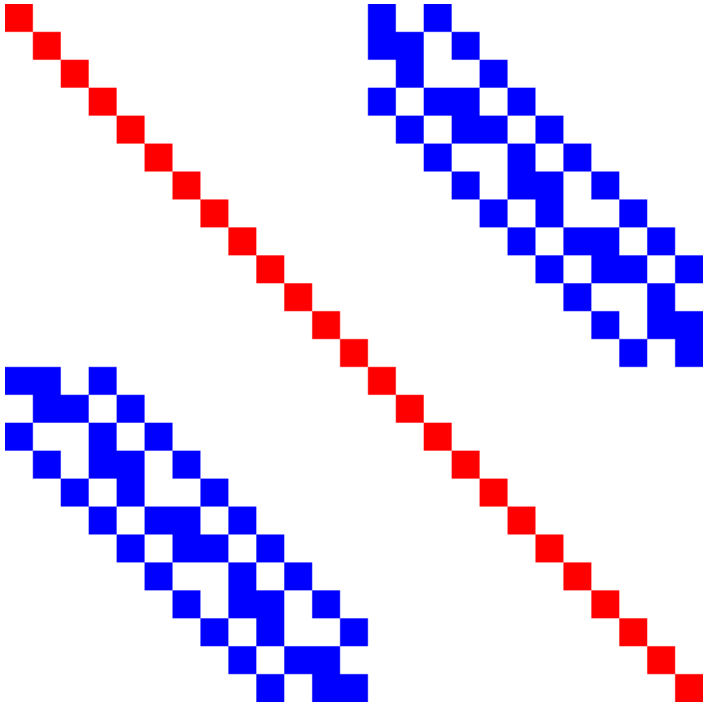
ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d





Parallel ILU

Sequential

```
for i=2..n
  for k=1..i-1, (i,k)in A
    a_ik = a_ik / a_kk
    for j=k+1..n, (i,j)in A
      u_ij = a_ij - a_ik a_kj
```

Parallel

```
for (sweep = 1, 2, ...)
  parallel for (i,j)in A
    if (i > j)
      l_ij = (a_ij - sum_{k=1}^{j=1} l_ik u_kj) / u_jj
    else
      u_ij = a_ij - sum_{k=1}^{=1} l_ik u_kj
```

Fine-Grained Parallel ILU Setup

Proposed by Chow and Patel (SISC, vol. 37(2)) for CPUs and MICs

Massively parallel (one thread per row)

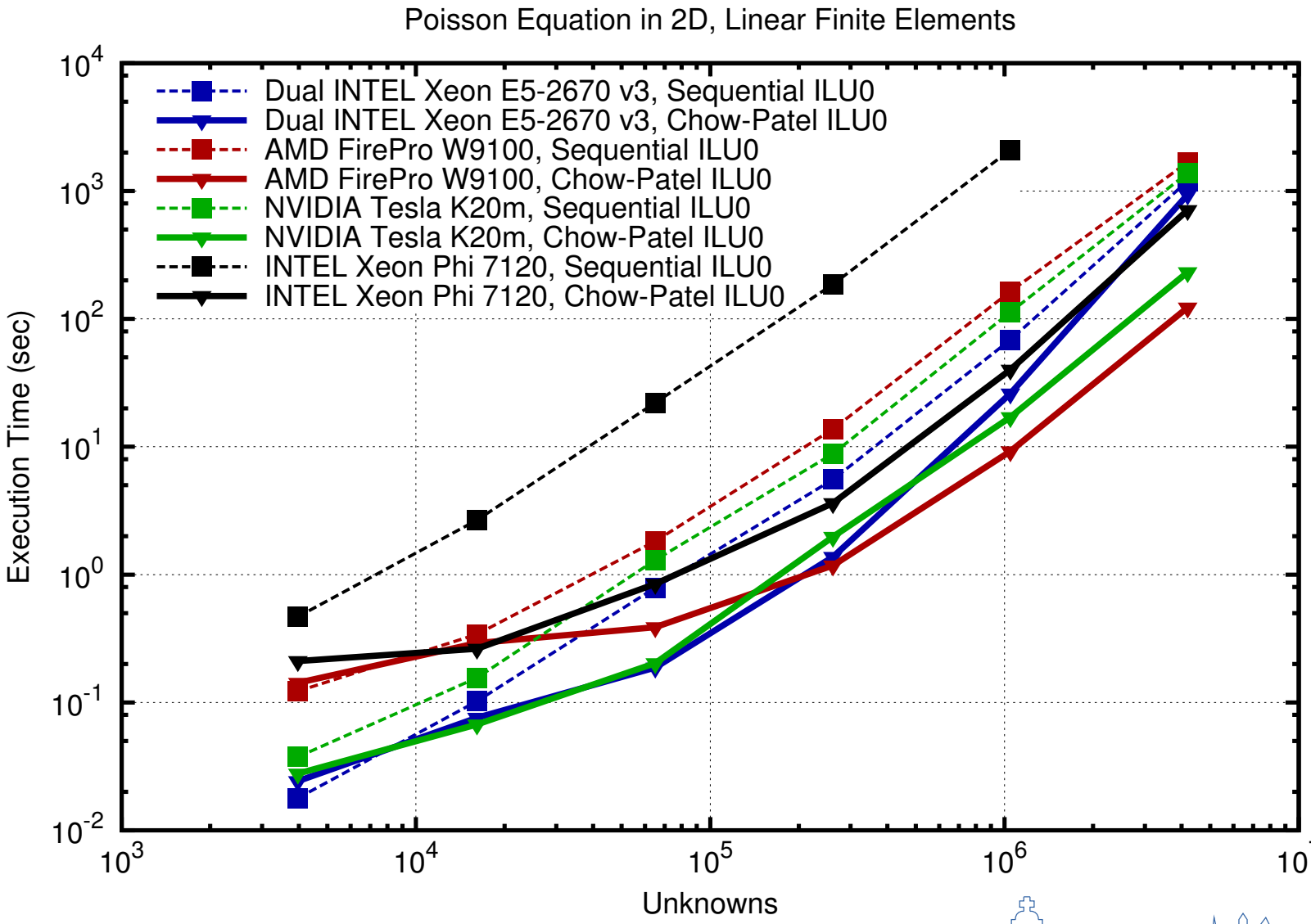
Preconditioner Application

Truncated Neumann series:

$$\mathbf{L}^{-1} \approx \sum_{k=0}^{K} (\mathbf{I} - \mathbf{L})^k, \quad \mathbf{U}^{-1} \approx \sum_{k=0}^{K} (\mathbf{I} - \mathbf{U})^k$$

Exact triangular solves not necessary

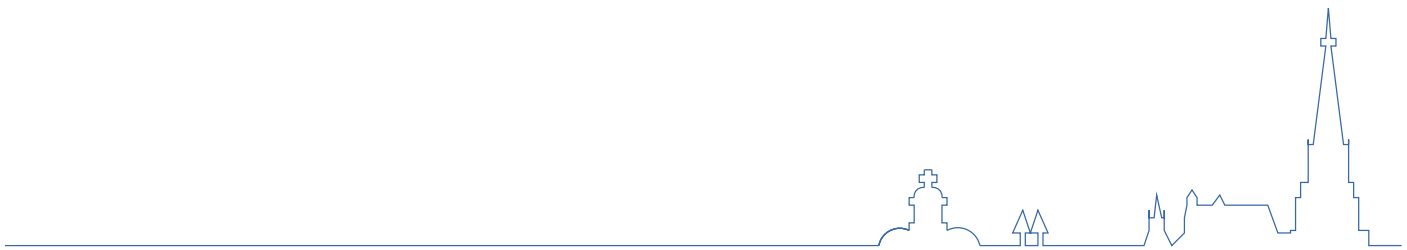
Parallel ILU



Content

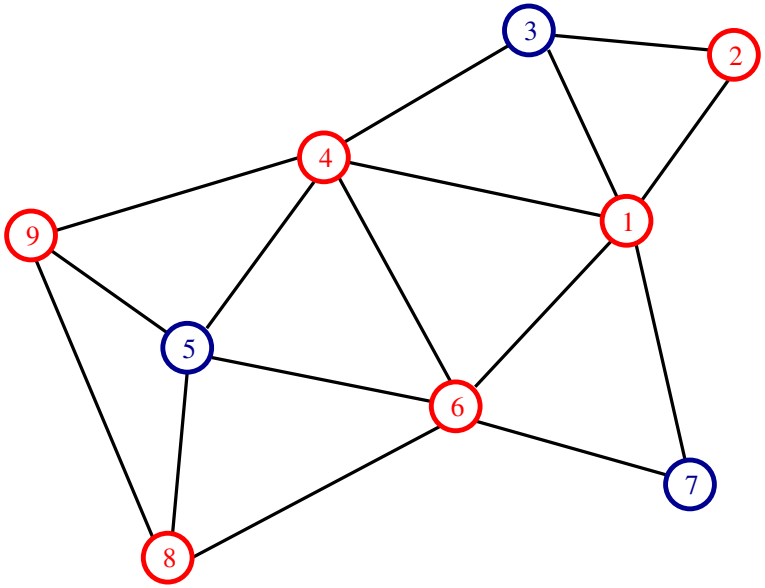
Algebraic Multigrid

- Parallel aggregation
- Sparse matrix-matrix products

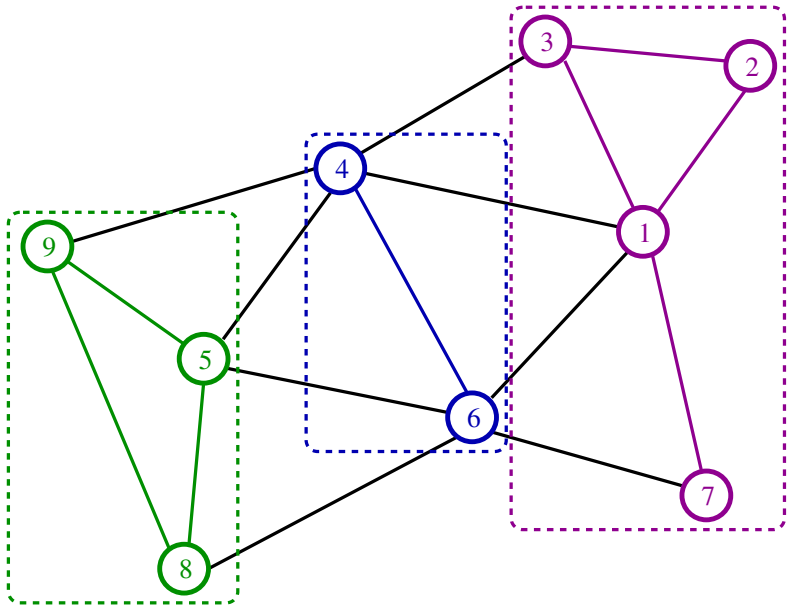


Ingredients of Algebraic Multigrid

- Smoother (Relaxation schemes, etc.)
- Coarsening
- Interpolation (Inter-grid transfer)



Classical coarsening

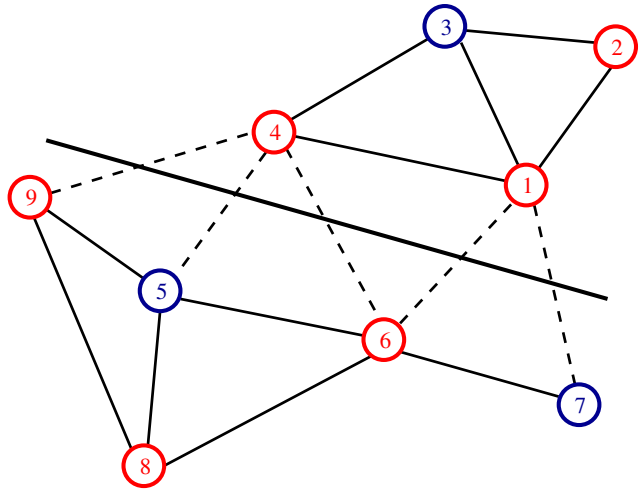


Aggregation coarsening

Multigrid Parallelization

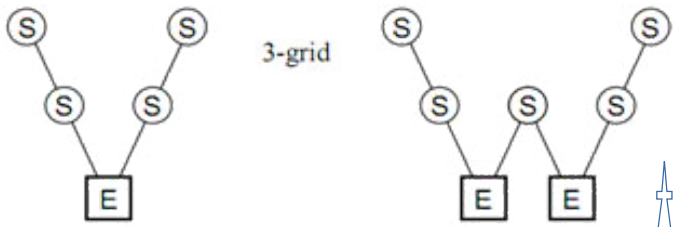
Setup Phase

- Determination of coarse points in parallel by graph splitting
- Compute coarse operators $A^{k+1} = R^k A^k P^k$ (where $A^0 = A$)
- Datastructures: analyze and allocate
- Limited fine-grained parallelism



Cycle Phase

- Parallel Jacobi Smoother
- Restriction $R^k x^k$, prolongation $P^k x^{k+1}$
- Direct solution on coarsest level
- Static datastructures
- Enough fine-grained parallelism



AMG Sparse Matrix-Matrix Multiplication

Coarse Grid Operator

$$A^{\text{coarse}} = RA^{\text{fine}}P$$

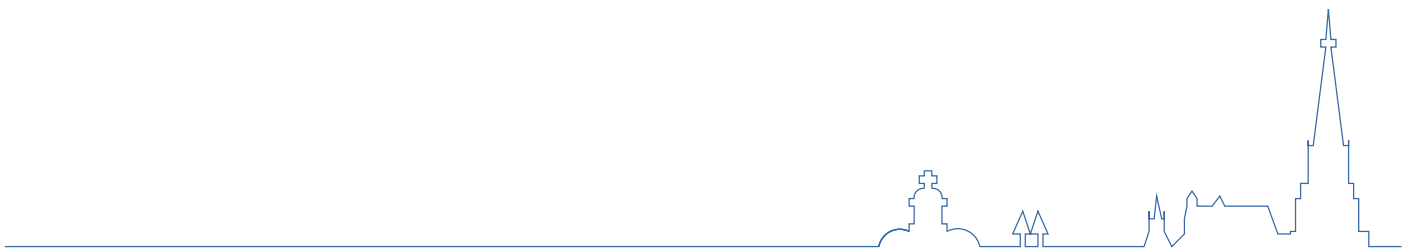
Common choice: $R = P^{\text{T}}$

Computation

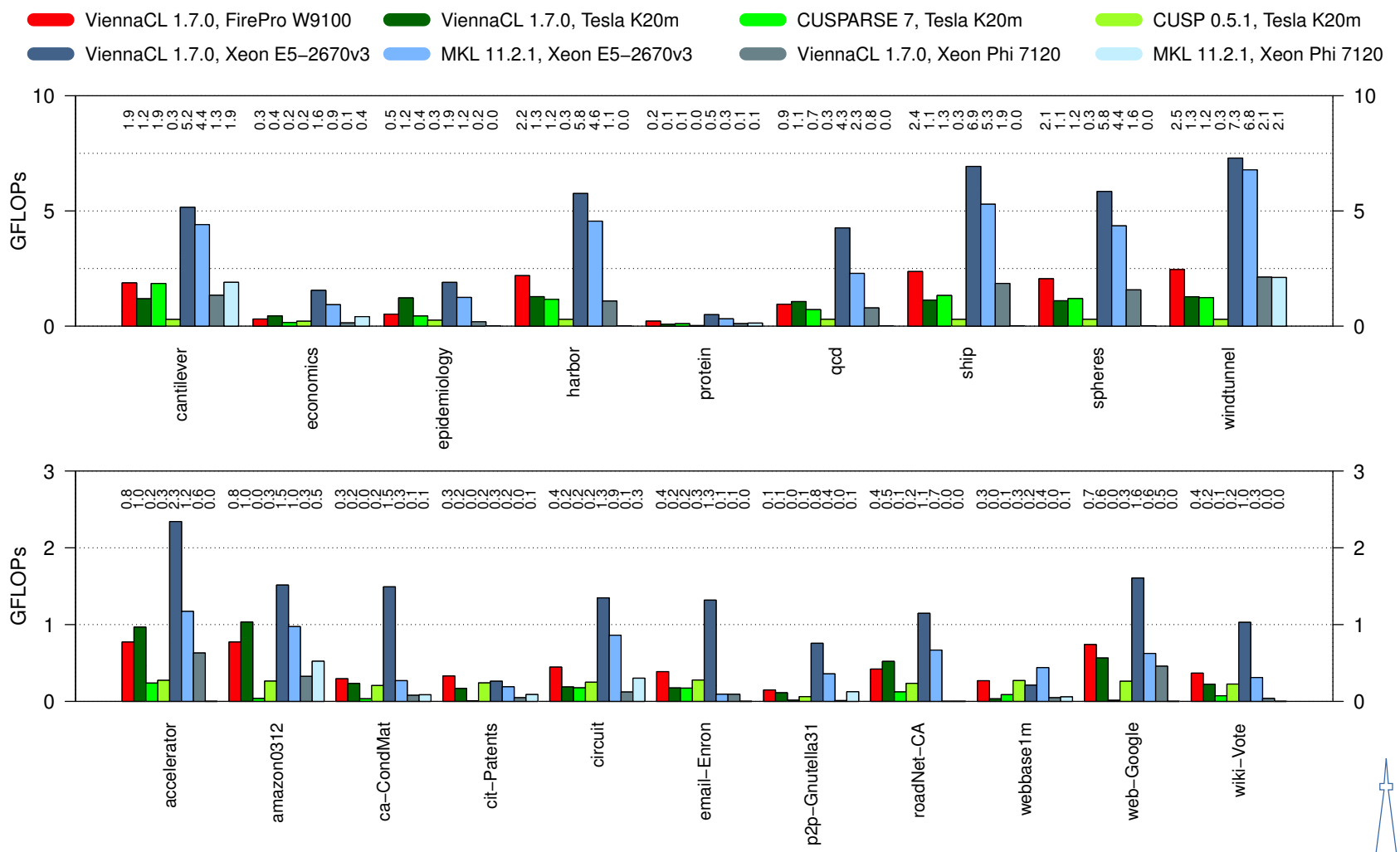
Explicitly set up $R = P^{\text{T}}$ (hard in parallel)

$$C = A^{\text{fine}}P$$

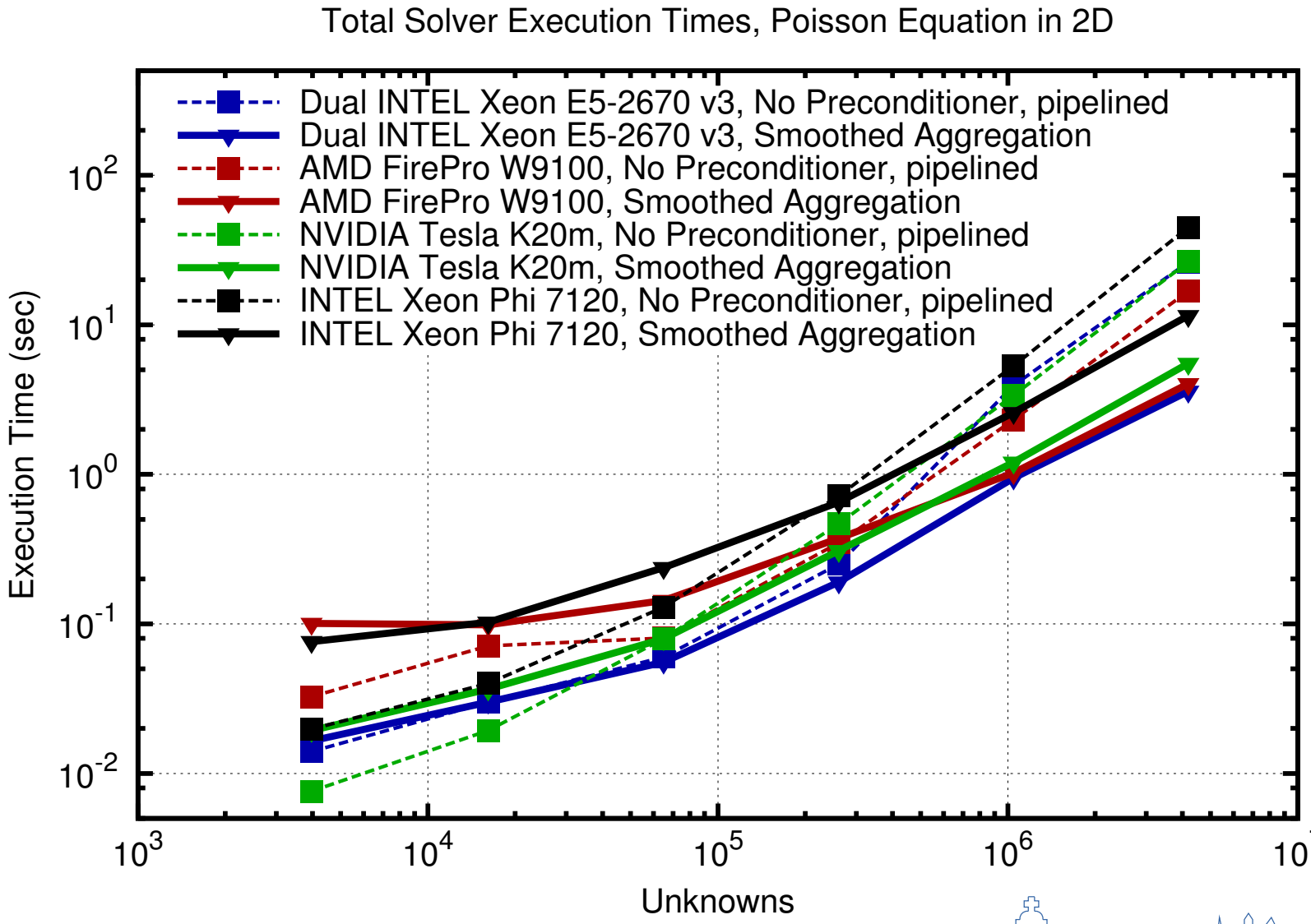
$$A^{\text{coarse}} = RC$$



AMG Sparse Matrix-Matrix Multiplication



AMG Benchmark



Summary

Parallel Primitives

- Embarassingly parallel operations (vector addition, etc.)
- Reductions
- Prefix Sums
- etc.

Solvers on Shared Memory Architectures

- Pipelining to reduce synchronization costs
- Pipelining to increase data reuse
- Replace sequential stages with parallel alternatives