# On Convenience and Inconvenience in GPU Computing

Karl Rupp
rupp@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory

LANS Seminar

June 26th, 2013

# 100x Speed-Up?

## Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee†, Changkyu Kim†, Jatin Chhugani†, Michael Deisher†, Daehyun Kim†, Anthony D. Nguyen†, Nadathur Satish†, Mikhail Smelyanskiy†, Srinivas Chennupaty*, Per Hammarlund*, Ronak Singhal* and Pradeep Dubey†

victor.w.lee@intel.com

†Throughput Computing Lab, Intel Corporation

*Intel Architecture Group, Intel Corporation

### ABSTRACT

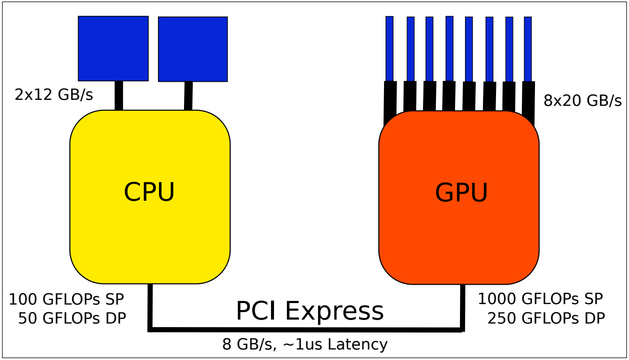
Recent advances in computing have led to an explosion in the amount of data being generated. Processing the ever-growing data in a timely manner has made throughput computing an important aspect for emerging applications. Our analysis of a set of important throughput computing kernels shows that there is an ample amount of parallelism in these kernels which makes them suitable for today's multi-core CPUs and GPUs. In the past few years there have been many studies claiming GPUs deliver substantial speedups (between 10X and 1000X) over multi-core CPUs on these kernels. To understand where such large performance difference comes from, we perform a rigorous performance analysis and find that after ap-

### 1. INTRODUCTION

The past decade has seen a huge increase in digital content as more documents are being created in digital form than ever before. Moreover, the web has become the medium of choice for storing and delivering information such as stock market data, personal records, and news. Soon, the amount of digital data will exceed exabytes ($10^{18}$) [31]. The massive amount of data makes storing, cataloging, processing, and retrieving information challenging. A new class of applications has emerged across different domains such as database, games, video, and finance that can process this huge amount of data to distill and deliver appropriate content to users. A distinguishing feature of these applications is that they

Proc. ISCA 2010

## Computing Architecture Schematic

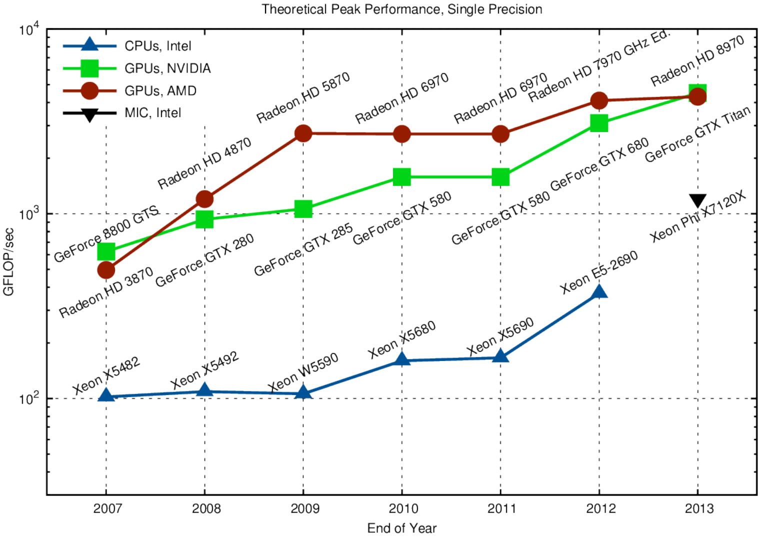



Good for large FLOP-intensive tasks, high memory bandwidth

PCI-Express can be a bottleneck

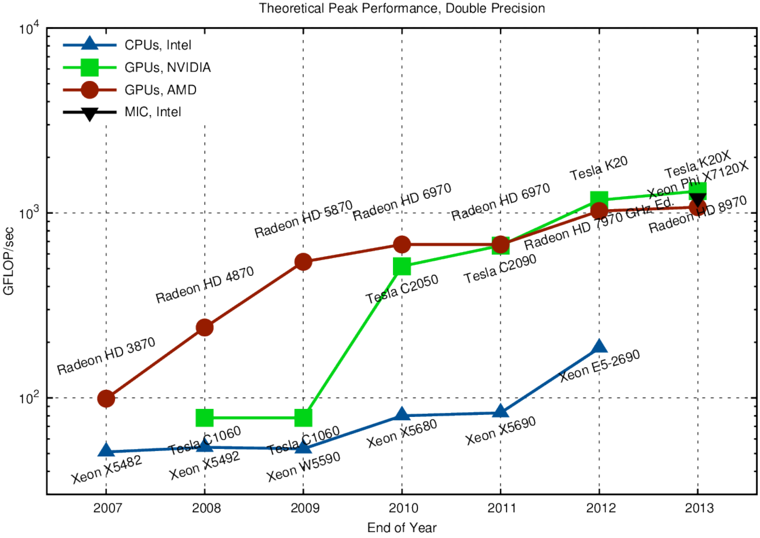
$\gg 10$-fold speedups (usually) not backed by hardware

# 100x Speed-Up?



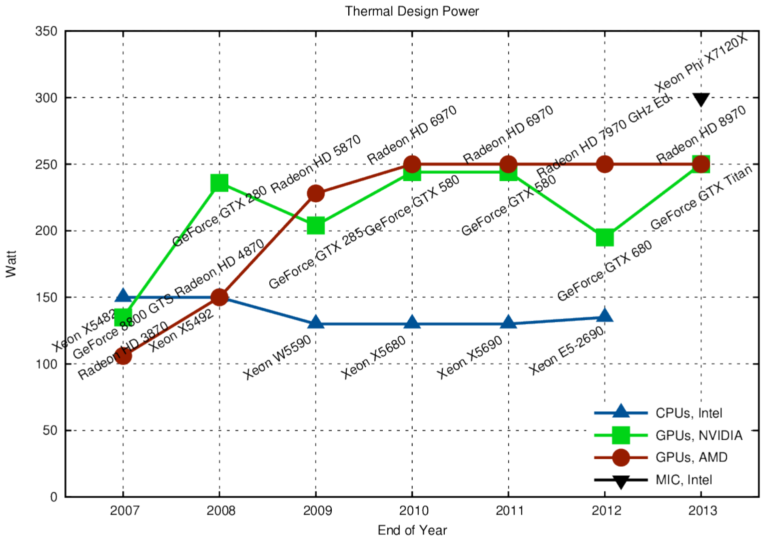

Single Precision Peak GFLOP/sec

# 100x Speed-Up?



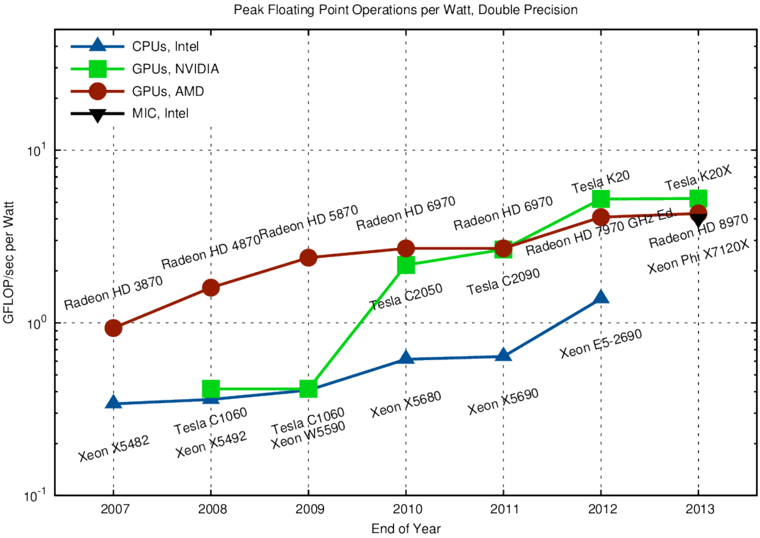

Double Precision Peak GFLOP/sec

# 100x Speed-Up?



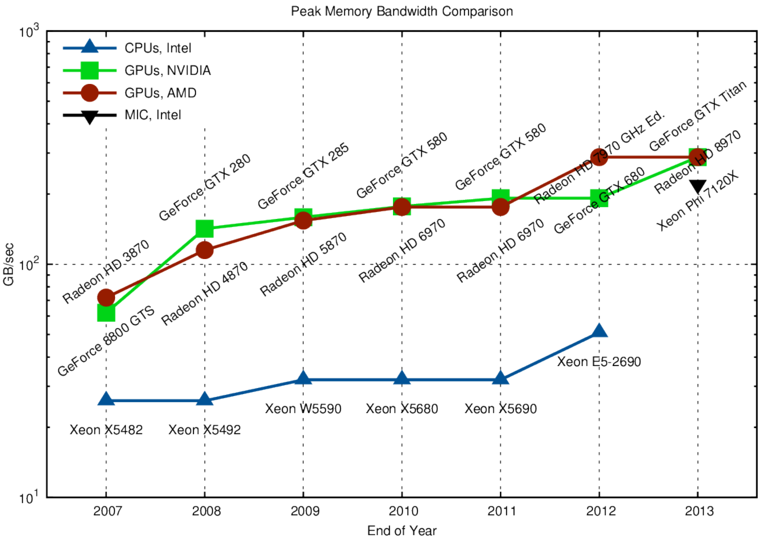

Thermal Design Power: 2 CPUs vs. 1 GPU for Fair Comparison

# 100x Speed-Up?




GFLOP/sec per Watt

# 100x Speed-Up?




Memory Bandwidth

# Which Accelerator is Right for Me?

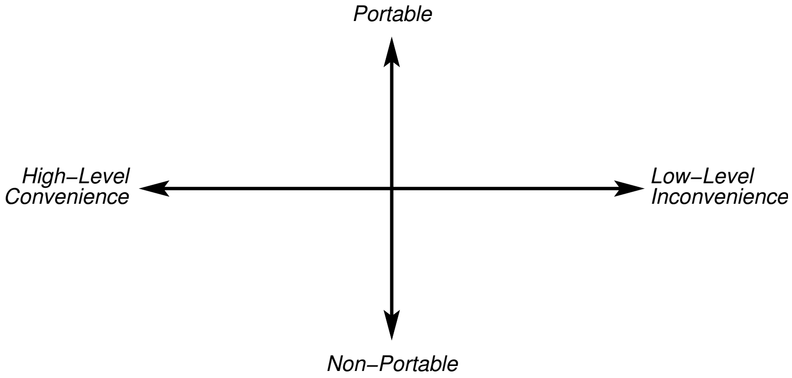
## Available Accelerators (Rough Sketch, Theoretical Peaks)

| Name | TFLOP/s | RAM (GB) | GB/s | TDP | Price |
|---|---|---|---|---|---|
| NVIDIA GTX 580 | 3.0/∼0.2 | 1.5-3.0 | 192 | 244 | $500 |
| NVIDIA GTX Titan | 4.5/1.3 | 6.0 | 288 | 250 | $∼1k |
| NVIDIA Tesla 2050 | 1.3/0.5 | 3.0-6.0 | 150 | 225 | $∼2k |
| NVIDIA K20 | 3.5/1.2 | 5.0 | 200 | 220 | $∼3k |
| AMD HD 7970 | 3.5/∼0.9 | 3.0-6.0 | 264 | 250 | $400 |
| AMD FirePro W9k | 4.0/1.0 | 6.0 | 264 | 274 | $∼3k |
| Intel Xeon Phi | ∼2.0/∼1.0 | 8.0 | 320 | 225 | $∼3k |
| Intel Xeon E5-264x | 0.2/0.1 | ∼64 | ∼48 | 100 | $∼1k |

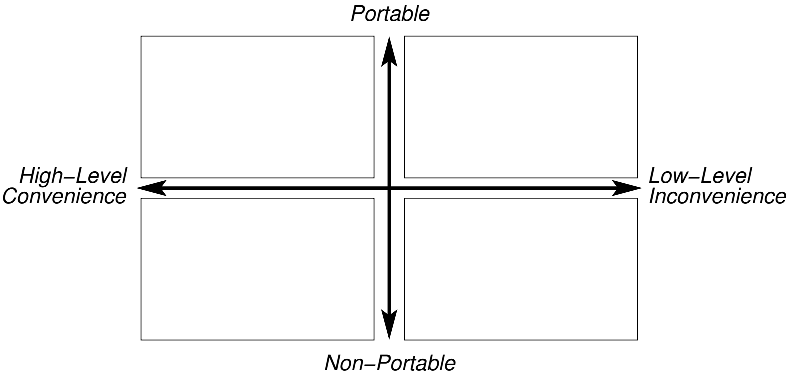
## PETSc Considerations

- Single precision performance doesn't matter
- Essentially all kernels memory bandwidth limited
- Memory access patterns rather irregular

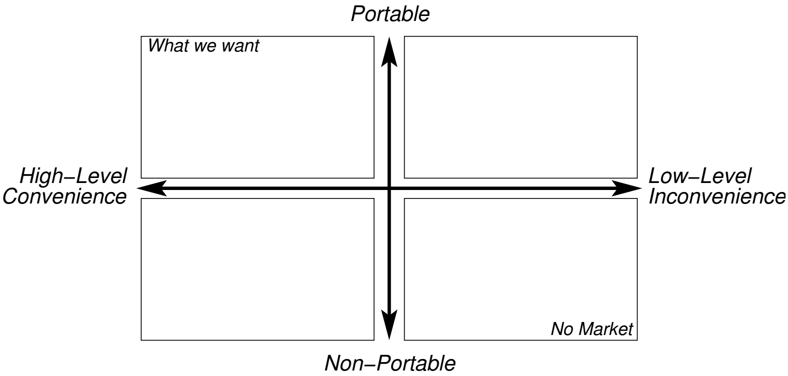
# GPU Programming Frameworks



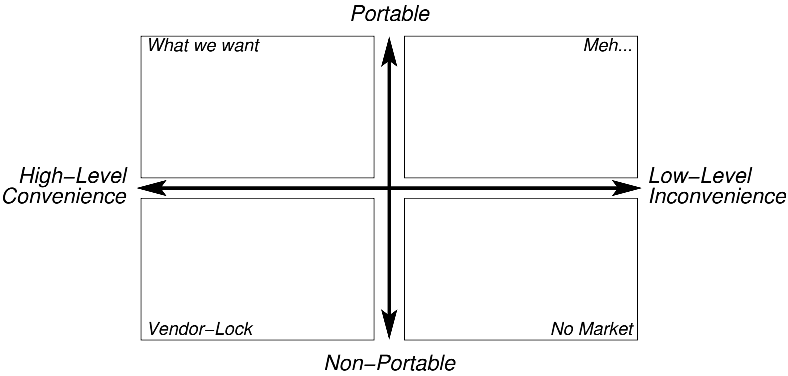
# GPU Programming Frameworks



# GPU Programming Frameworks



# GPU Programming Frameworks



# GPU Programming Frameworks

## NVIDIA CUDA

```
// GPU kernel:
__global__ void kernel(double *buffer)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  buffer[idx] = 42.0;
}

// host code:
int main()
{
  ...
  cudaMalloc((void**)&buffer,size);
  kernel<<<blocknum, blockdim>>>(buffer);
  ...
}
```

Convenient: Almost no additional code required

Non-Portable: Vendor-lock

Non-Portable: Relies on `nvcc` being available

## NVIDIA CUDA Driver API

```
// host code:
int main()
{
  ...
  cudaMalloc((void**)&buffer,size);
  void *args[1] = { &buffer };
  cuModuleLoad(&module, module_file);
  cuModuleGetFunction(&function, module, kernel_name);
  cuLaunchKernel(function,
                 N, 1, 1, // Nx1x1 blocks
                 1, 1, 1, // 1x1x1 threads
                 0, 0, args, 0)
  ...
}
```

Relief: No dependency on `nvcc`

Inconvenient: Pseudo-Assembly PTX code

Non-Portable: PTX updated with each device generation

## NVIDIA CUDA Driver API

```
// Generated by NVIDIA NVVM Compiler
// Compiler built on Sun Aug 19 23:20:45 2012
// Driver 304.43

.version 3.0
.target sm_13, texmode_independent
.address_size 32

.entry _k0(
 .param .u32 .ptr .global .align 4 _k0_param_0,
 .param .u32 _k0_param_1,
 .param .u32 .ptr .global .align 4 _k0_param_2,
 .param .u32 _k0_param_3,
 .param .u32 .ptr .global .align 4 _k0_param_4,
 .param .u32 _k0_param_5,
 .param .u32 _k0_param_6,
 .param .u32 _k0_param_7,
 .param .u32 .ptr .global .align 4 _k0_param_8,
 .param .u32 _k0_param_9,
 ...
)
```

## OpenACC and Friends

```
void func(...) {
  #pragma acc data pcopyin(A[0:size][0:size])
  {
    #pragma acc kernels loop
    for(int i=0; i< size; i++)
      for(int j=0; j < size; j++)
        A[i][j] = 42;
  }
}

int main()
{
  double A[1337][1337];
  func(A);
}
```

Convenient: Simple OpenMP-type pragma annotations

Non-Portable: Compiler support?

Non-Portable: Insufficient control over memory transfers?

## OpenCL

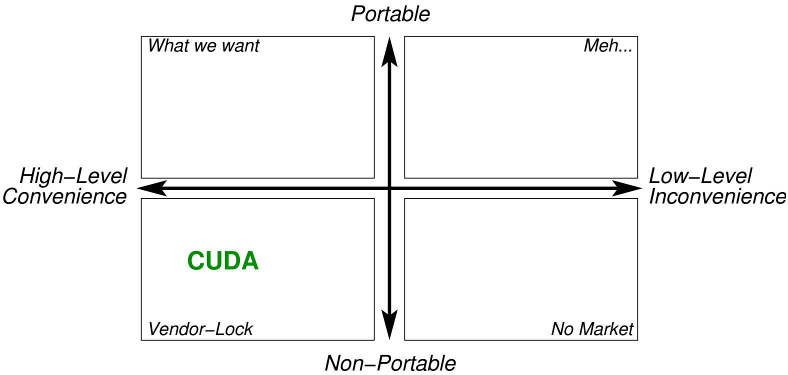
```
const char *kernel_string =
"__kernel void mykernel(__global double *buffer) {
  buffer[get_global_id(0)] = 42.0;
};";

int main() {
  ...
  cl_program my_prog = clCreateProgramWithSource(
         my_context,1,&kernel_string,&source_len,&err);
  clBuildProgram(my_prog,0,NULL,NULL,NULL,NULL);
  cl_kernel my_kernel = clCreateKernel(my_prog,
                                   "mykernel",&err);
  clSetKernelArg(my_kernel,0,sizeof(cl_mem),&buffer);
  clEnqueueNDRangeKernel(queue,my_kernel,1,NULL,
               &global_size,&local_size,0,NULL,NULL);
}
```

Inconvenient: Low-level boilerplate code required

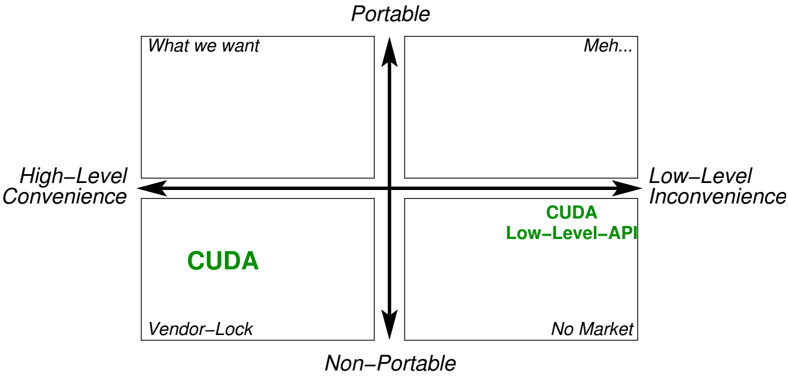
Portable: Broad hardware support (separate SDKs)

Bad News: No more development effort from NVIDIA

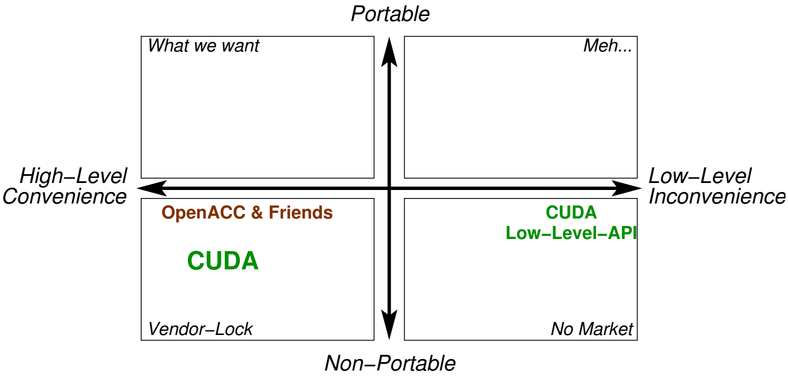
# GPU Programming Frameworks



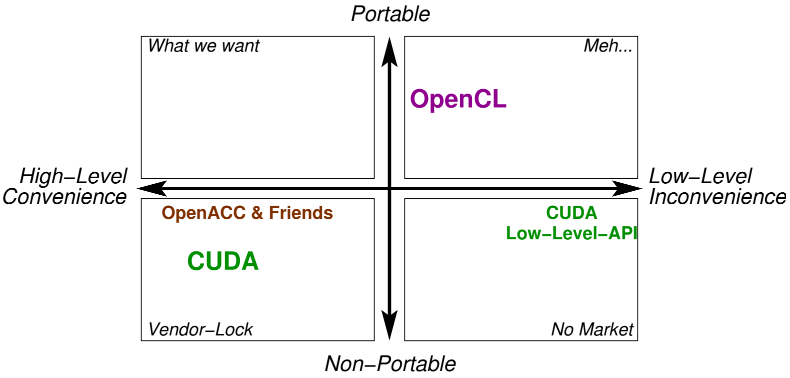
# GPU Programming Frameworks



# GPU Programming Frameworks



# GPU Programming Frameworks



# GPUs: Library Aspects

## Challenge: Hardware

- Portable performance
- Auto-tuning
- Testing requires many different machines

## Challenge: Memory

- Allocation failures?
- Multi-GPU?
- PCI-Express bottleneck

## Challenge: Programming

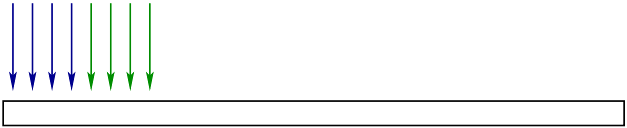
- Kernel language?
- Which low-level parameters to expose?

# Part 2: Basic Linear Algebra

## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

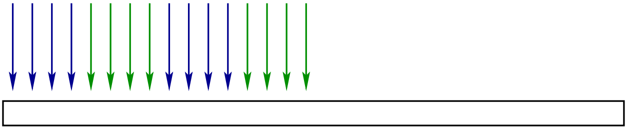
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

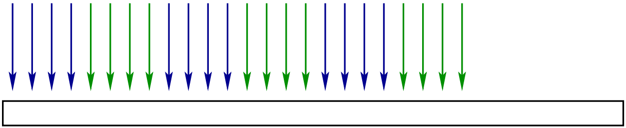
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

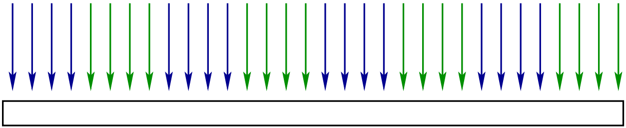
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

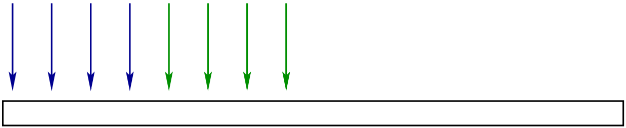
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

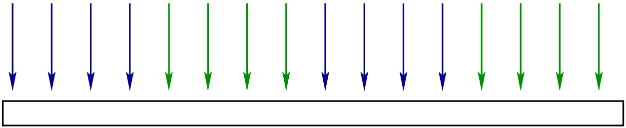
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

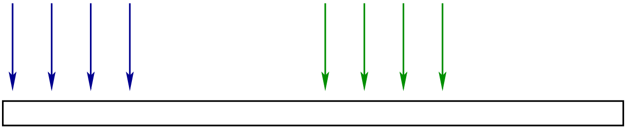
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

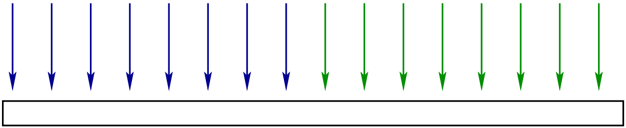
Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

Usually a simple for-loop, memory-bandwidth limited



## Consider Something Simple

$x \leftarrow y$, $x, y \in \mathbb{R}^N$, distinct

Usually a simple for-loop, memory-bandwidth limited

## Parameters

Data Types: `double`, `double2`, etc.

Blocking: Small segments vs. large blocks

Thread sizes: threads per group, number of thread groups

## OpenCL Benchmarking Baseline

**`double`**

small segments

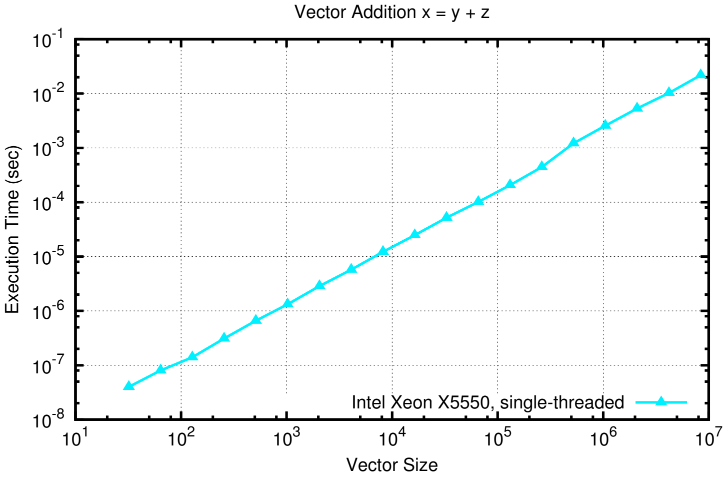
128 threads per work group

128 work groups

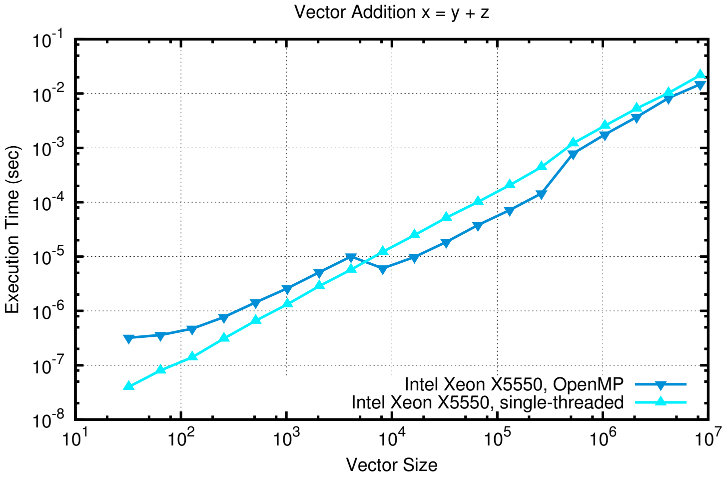
## Results (GB/sec)

| Name | **`double`**, small, 128x128 | Best | `double2`, large, 32x240 |
|---|---|---|---|
| NVIDIA GTX 285 | 101 | 134 | 134 |
| NVIDIA GTX 580 | 150 | 166 | 123 |
| AMD HD 7970 | 161 | 249 | 140 |
| INTEL E5-2670 x2 | 32 | 79 | 18 |
| INTEL Xeon Phi | 32 | 95 | 21 |

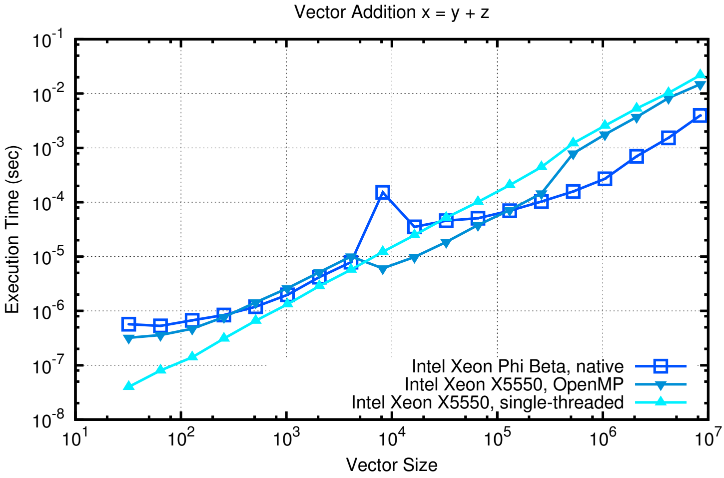
# Vector Addition Benchmarks



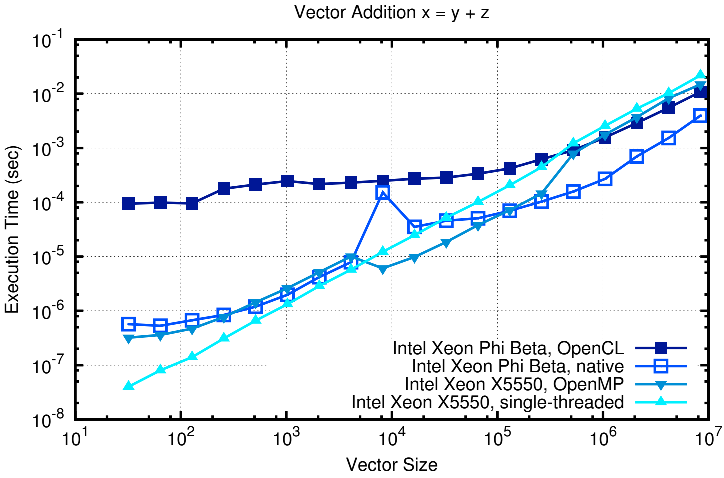
# Vector Addition Benchmarks



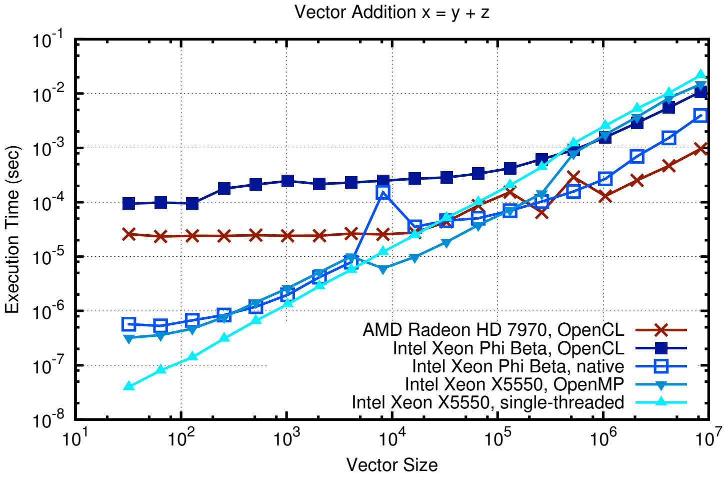
# Vector Addition Benchmarks



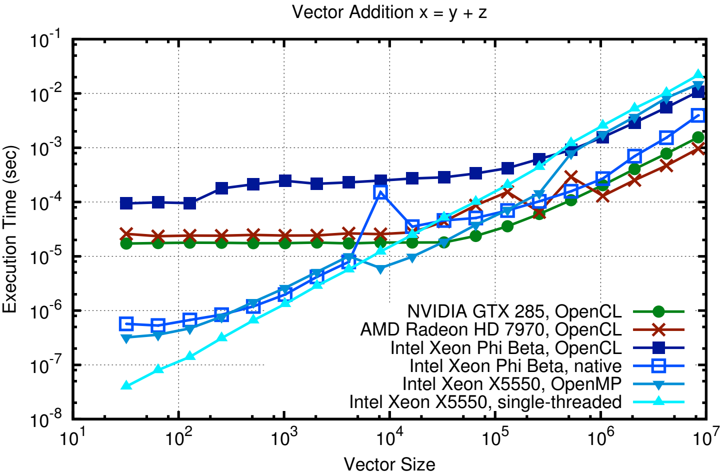
# Vector Addition Benchmarks



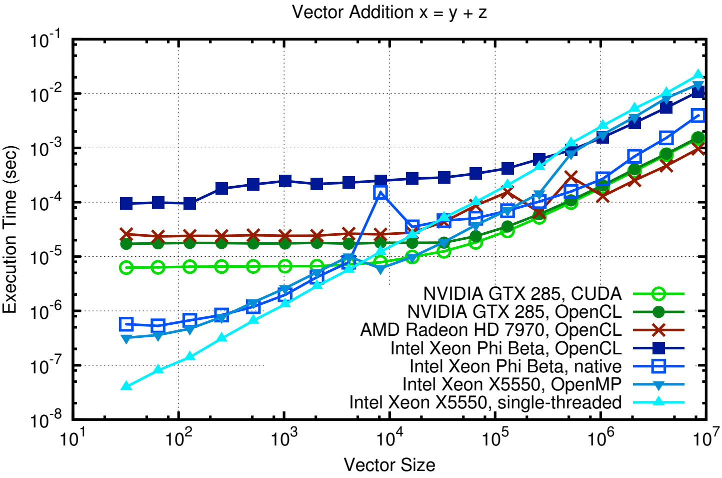
# Vector Addition Benchmarks



# Vector Addition Benchmarks



# Vector Addition Benchmarks



# Matrix-Matrix Multiplication

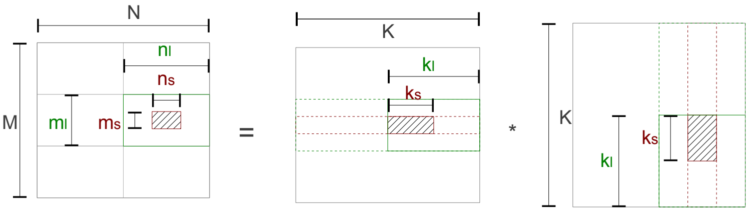
## Parameter Space Explosion

Global Block Sizes

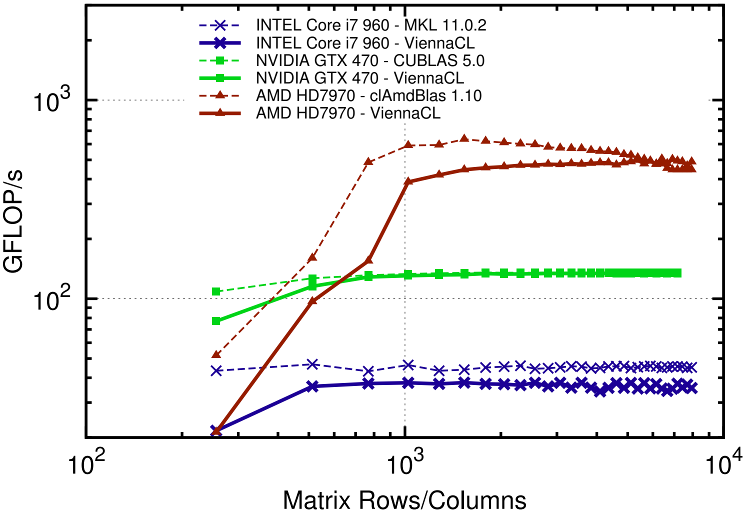
Shared Mem Block Sizes

Thread-Private Block Sizes

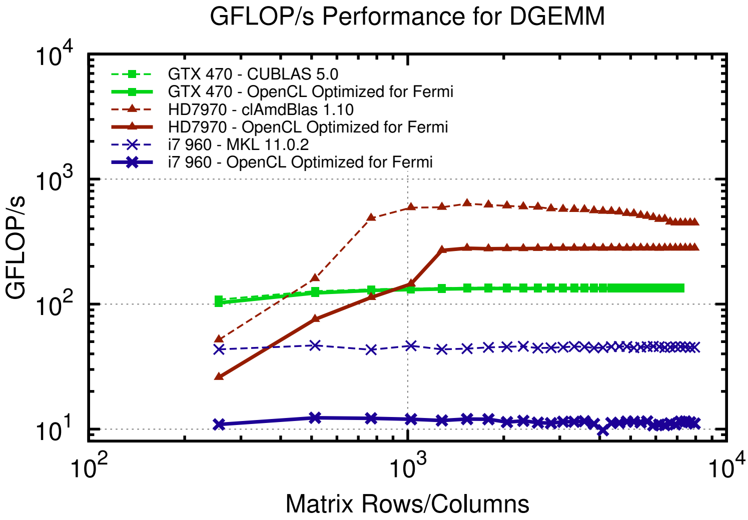
Loop Unrolling



# Matrix-Matrix Multiplication



# Matrix-Matrix Multiplication



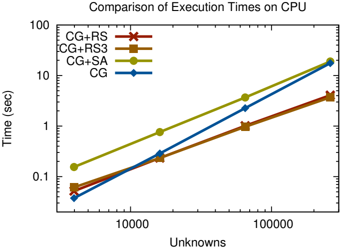
# Part 3: Preconditioners

## Poisson Equation in 2D

Unstructured triangular grid

Finite element discretization

## Various Algebraic Multigrid Methods




Intel Core i7 960 CPU

# Algebraic Multigrid Preconditioners

## Poisson Equation in 2D

Unstructured triangular grid

Finite element discretization

## Various Algebraic Multigrid Methods


Intel Core i7 960 CPU


NVIDIA GTX 470 GPU

# Algebraic Multigrid Preconditioners

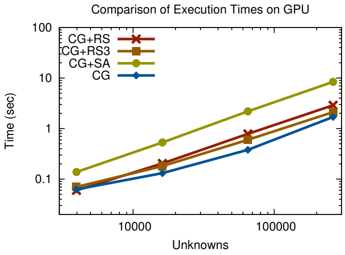
## Poisson Equation in 2D

Unstructured triangular grid

Finite element discretization

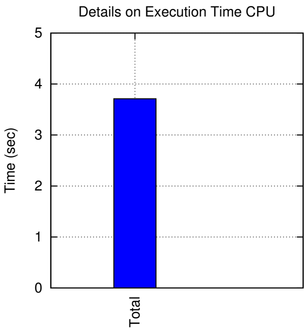
## Various Algebraic Multigrid Methods


Intel Core i7 960 CPU

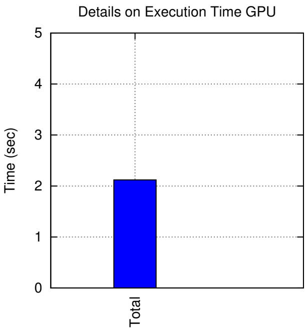



NVIDIA GTX 470 GPU

## A Closer Look

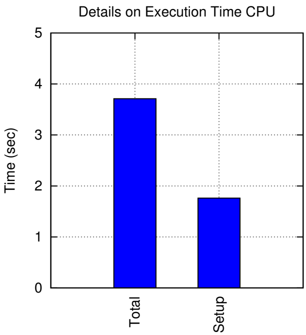



Intel Core i7 960 CPU

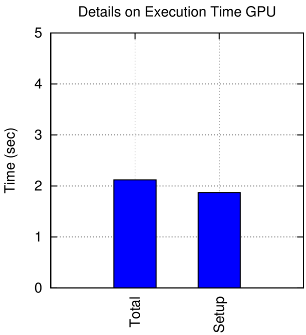



NVIDIA GTX 470 GPU

## A Closer Look




Intel Core i7 960 CPU




NVIDIA GTX 470 GPU

## Findings

Solver cycle time scales with GPU

Setup time limited by CPU $\Rightarrow$ Minimize

# Incomplete LU Factorization Preconditioners

## 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 ${\color{red}\tilde{L}}\tilde{U}x = z$

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

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

Little parallelism in general

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

# Incomplete LU Factorization Preconditioners

## Level Scheduling

- Build dependency graph
- Substitute as many entries as possible simultaneously
- Trade-off: Each step vs. multiple steps in a single kernel

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

# Incomplete LU Factorization Preconditioners

## Level Scheduling

- Build dependency graph
- Substitute as many entries as possible simultaneously
- Trade-off: Each step vs. multiple steps in a single kernel

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

## Level Scheduling

Build dependency graph

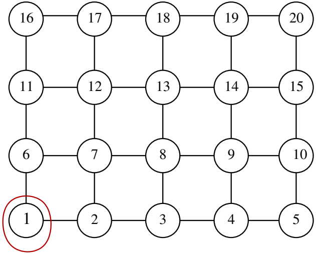
Substitute as many entries as possible simultaneously

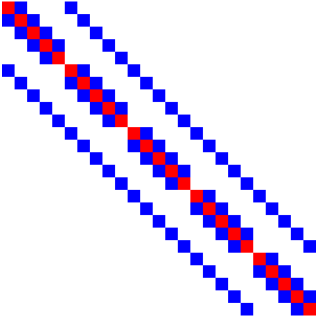
Trade-off: Each step vs. multiple steps in a single kernel

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

## Interpretation on Structured Grids

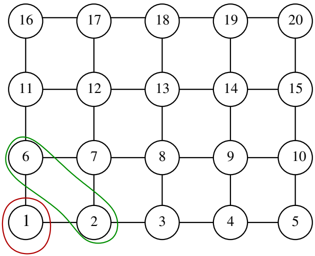
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

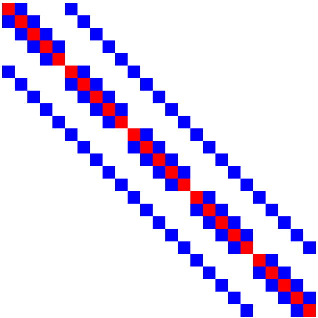


## Interpretation on Structured Grids

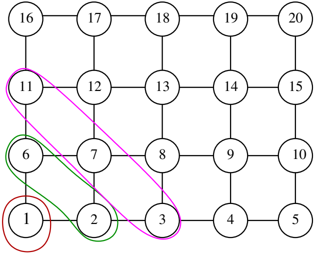
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

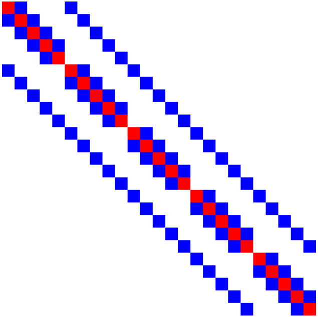


## Interpretation on Structured Grids

- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d



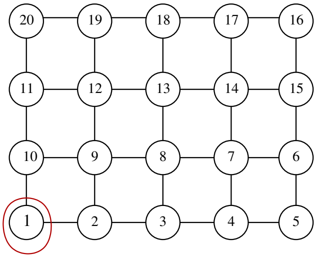



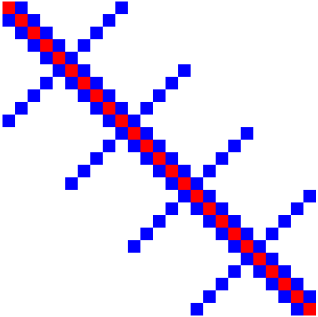
## Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d

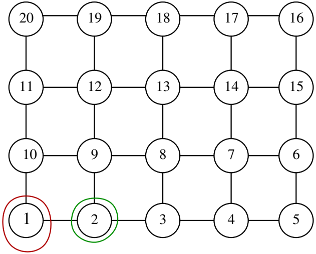


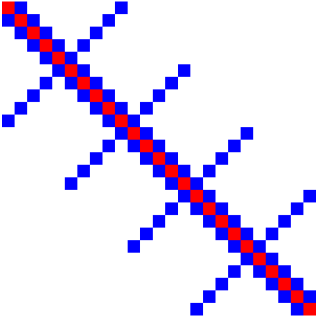



# Incomplete LU Factorization Preconditioners

## Interpretation on Structured Grids

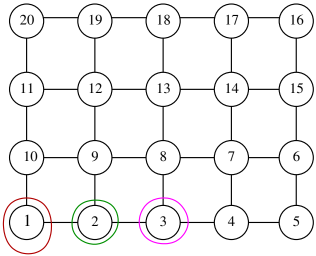
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

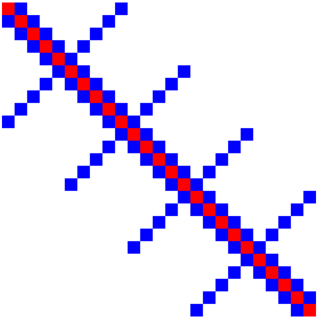


## Interpretation on Structured Grids

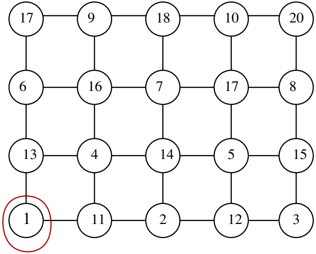
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

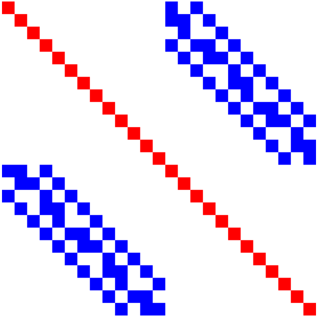


## Interpretation on Structured Grids

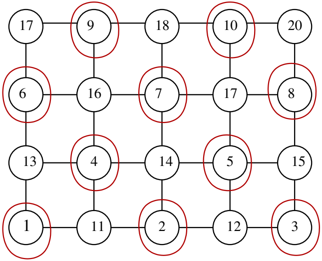
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

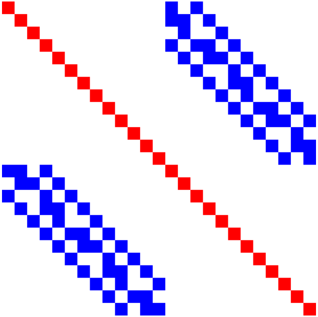


## Interpretation on Structured Grids

- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

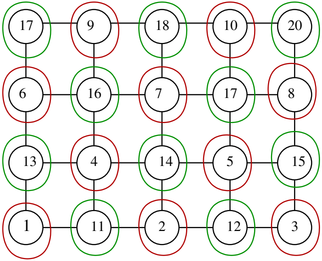


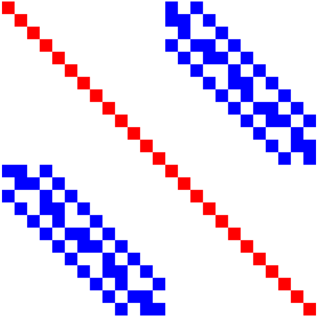



# Incomplete LU Factorization Preconditioners

## Interpretation on Structured Grids

- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d



# Incomplete LU Factorization Preconditioners

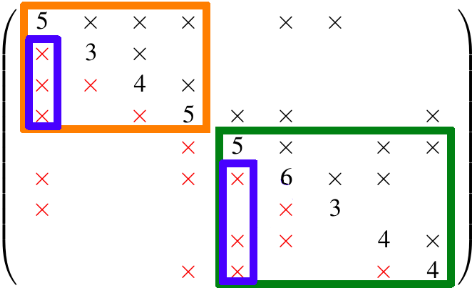
## Block-ILU

- Apply ILU to diagonal blocks
- Higher parallelism
- Usually more iterations required (problem-dependent)

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

## Block-ILU

- Apply ILU to diagonal blocks
- Higher parallelism
- Usually more iterations required (problem-dependent)



Benchmark - Setup

# Benchmark Setup

## Hardware

- NVIDIA GTX 580 (default)
- AMD HD 7970 (only for final benchmark)
- Intel Core2Quad 9550

## Numbering

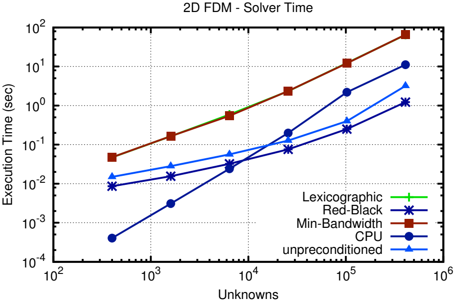
- Lexicographic
- Red-Black
- Minimum Degree

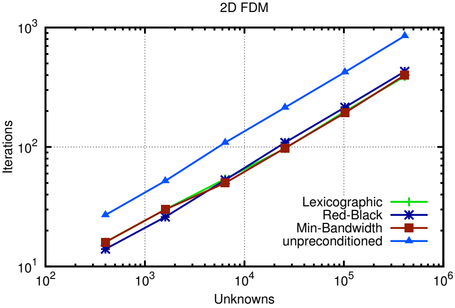
## Remarks

- Setup purely on CPU, not included
- Data transfer costs not included
- OpenCL for both GPUs

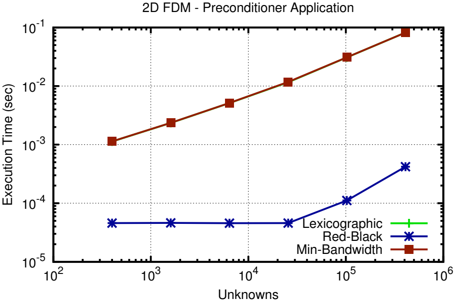
Case Study 1: 2D Poisson, Structured Grid

# Benchmarks

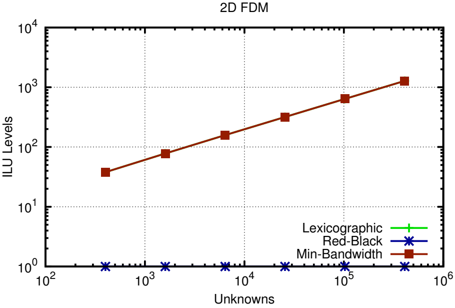

2D FDM - Solver Time
Execution Time (sec)
Unknowns
Lexicographic
Red-Black
Min-Bandwidth
CPU
unpreconditioned


2D FDM
Iterations
Unknowns
Lexicographic
Red-Black
Min-Bandwidth
unpreconditioned

# Benchmarks



2D FDM - Preconditioner Application
Execution Time (sec)
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^{2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
Unknowns
Lexicographic
Red-Black
Min-Bandwidth

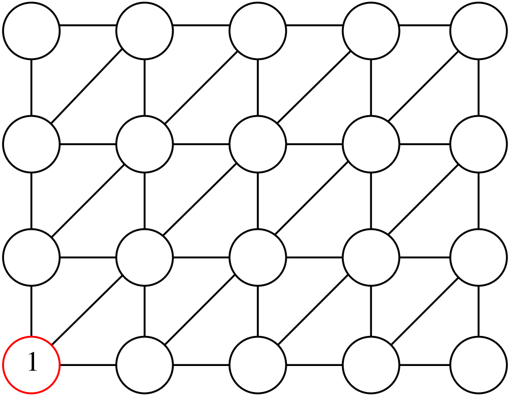



2D FDM
ILU Levels
$10^{4}$
$10^{3}$
$10^{2}$
$10^{1}$
$10^{0}$
$10^{2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
Unknowns
Lexicographic
Red-Black
Min-Bandwidth

# Incomplete LU Factorization Preconditioners

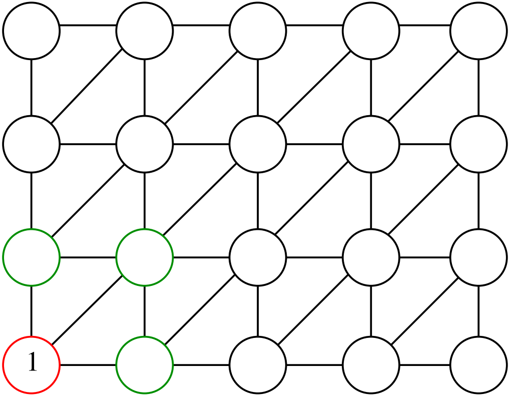
## Coloring

- Color dependency graph
- Purely algebraic



## Coloring

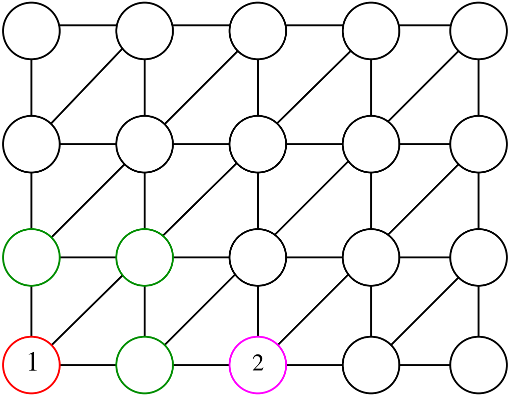
- Color dependency graph
- Purely algebraic



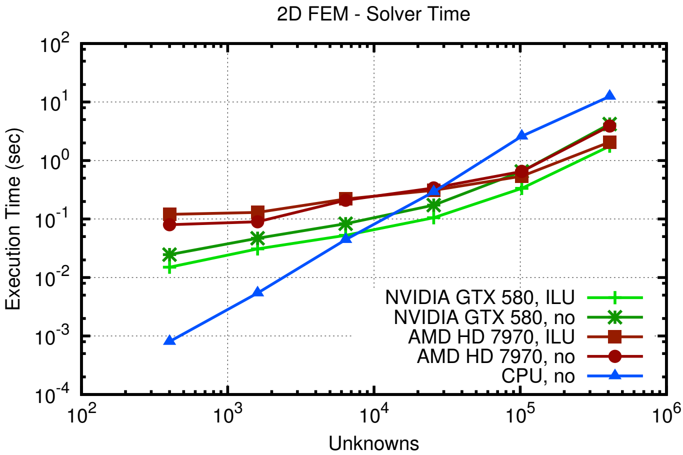
## Coloring

Color dependency graph

Purely algebraic

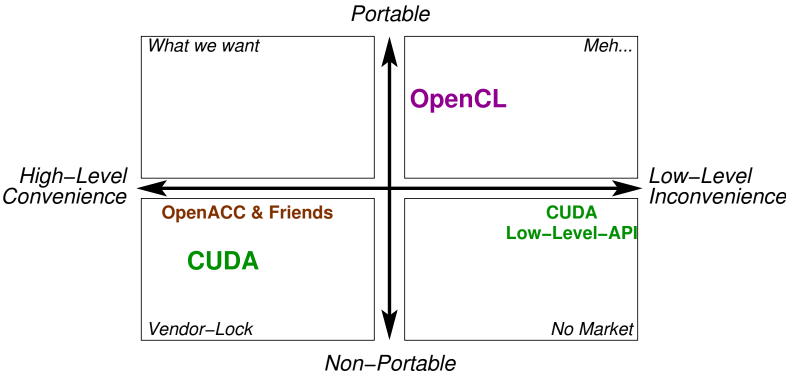


# Benchmarks



2D FEM - Solver Time
Execution Time (sec)
Unknowns
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
NVIDIA GTX 580, ILU
NVIDIA GTX 580, no
AMD HD 7970, ILU
AMD HD 7970, no
CPU, no

# GPU Programming Conclusion



# GPU Programming Conclusion