

# VSC Seminar

## Modern Many-Core Architectures for Supercomputing

Karl Rupp

<https://karlrupp.net/>  
<https://github.com/karlrupp/slides/>



*formerly:*  
Institute for Microelectronics, TU Wien

*now:*  
Freelance Scientist



December 11, 2015

## Hardware

Graphics Processing Units

Many Integrated Cores (Xeon Phi family)

## Software

CUDA

OpenCL

Parallel Primitives

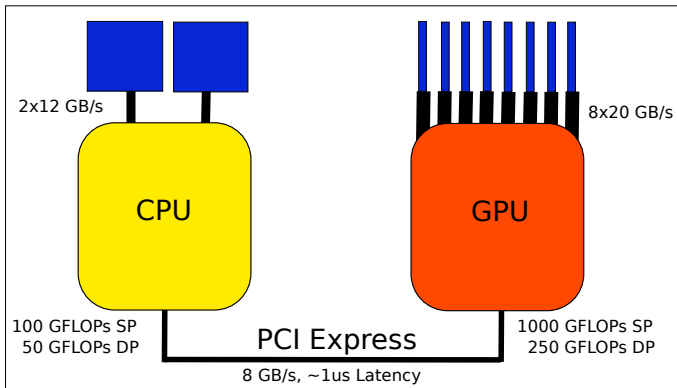
## Performance Modeling

Identifying Bottlenecks

Modeling by Example



## Computing Architecture Schematic

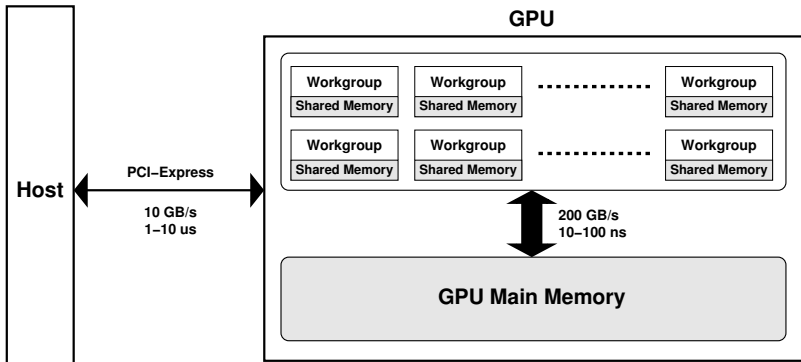


Good for large FLOP-intensive tasks, high memory bandwidth

PCI-Express can be a bottleneck

» 10-fold speedups (usually) not backed by hardware

# GPU Overview



## Details

Workgroups consist of 32-64 hardware threads

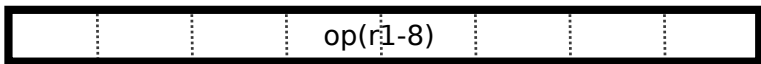
Up to 24 hardware workgroups

Shared memory small: approx. 32-64 KB



## Reminder: AVX

One instruction for all elements of a vector register

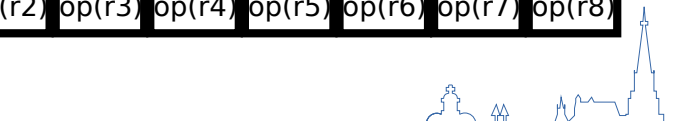
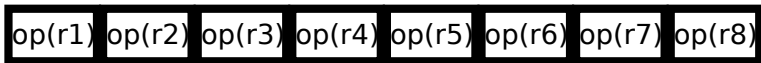


## Single Instruction Multiple Threads (SIMT)

One instruction for all threads in workgroup

Each thread has separate registers

Efficient if all threads execute the same instruction



## GDDR5

Optimized for throughput

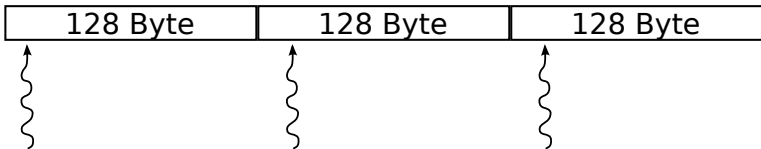
Channel width: multiple of 32 bits

High bus width: 256 bits, 384 bits

## Structured Memory Access

Memory controllers use 32/64/128 byte transactions

Partial transactions degrade effective bandwidth

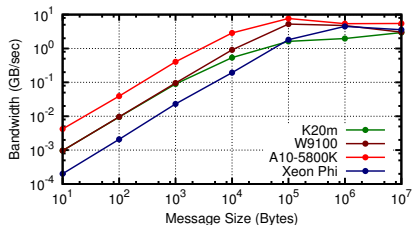
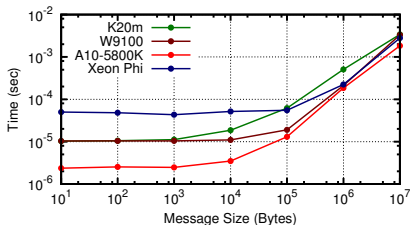


## Host-Device Communication

PCI-Express v2: 8 GB/sec max

PCI-Express v3: 16 GB/sec max

Latency: about 10  $\mu$ s



# Many Integrated Core Architecture

## Background

Many-core architecture by Intel

“Just recompile” existing OpenMP-enabled code

Current: Knights Corner (Q4 2012)

Upcoming: Knights Landing (Q1 2016)





# Many Integrated Core Architecture

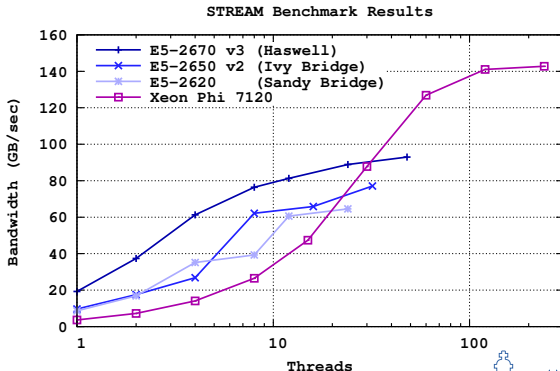
## Knight Corner: Technical Details

High memory bandwidth: 320 GB/sec ideal, 160 GB/sec real

High compute power: 1 TFLOP/sec (fp64)

Fixed main memory: 8 or 16 GB

Custom operating system on board



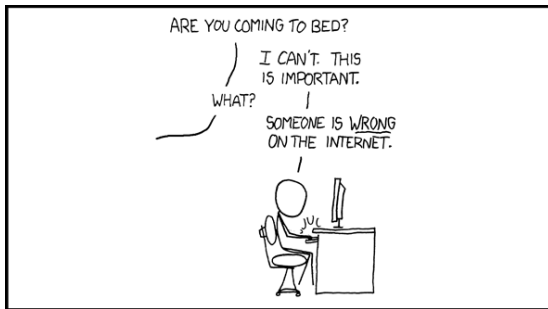
## Knights Corner: Problems

Very low single-thread performance

“Just recompile” almost never enough

PCI-Express communication slower than for GPUs

Even Intel now recommends Haswell-Xeons over Knights Corner



<http://xkcd.com/386/>

## KNL: Compute

72 cores

Energy-efficient IA cores

3x single-thread performance vs. KNC

Binary compatibility with Xeon line

3+ TFLOPS in double precision

## KNL: On-Package Memory

16 GB

5x bandwidth vs. DDR4

5x power efficiency vs. DDR4

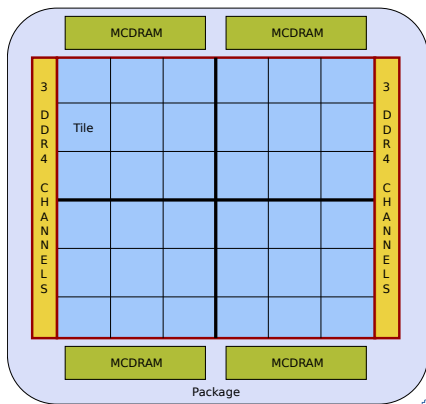


# Many Integrated Core Architecture

## KNL Schematic

36 tiles interconnected by 2D mesh

Each tile: 2 cores, 2 vector units per core, 1 MB L2 cache



## HBM: Cache Mode

No code changes necessary

Cache misses expensive

## HBM: Flat Mode

MCDRAM mapped to physical address space

Exposed as NUMA node

Accessed through memkind library or numactl

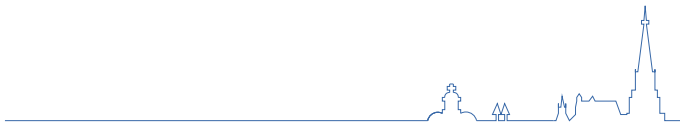
## HBM: Hybrid Mode

Combination of the above two

Get the best and worst of two worlds



Software: 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, ViennaCL, etc.



## 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
    ...

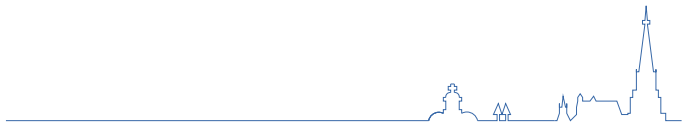
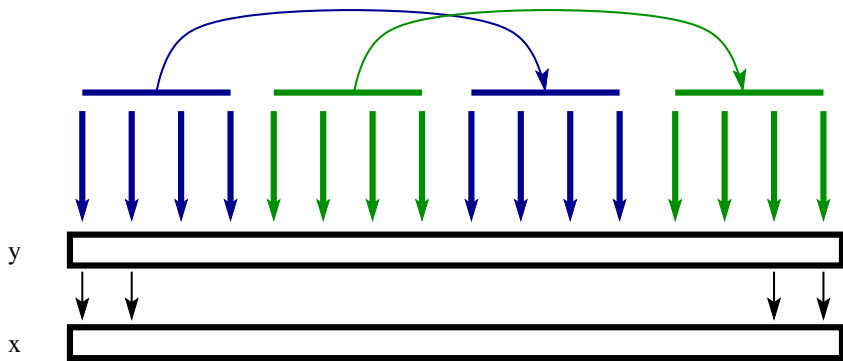
    free(x);
}
```



## 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);
}
```



## 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

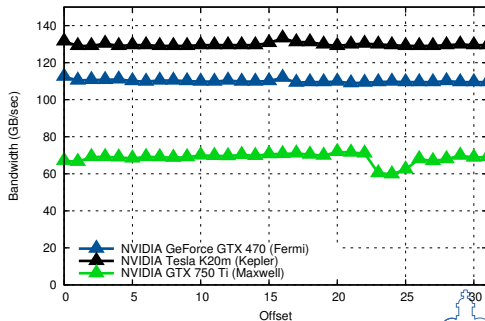


# CUDA Example

## 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];  
}
```

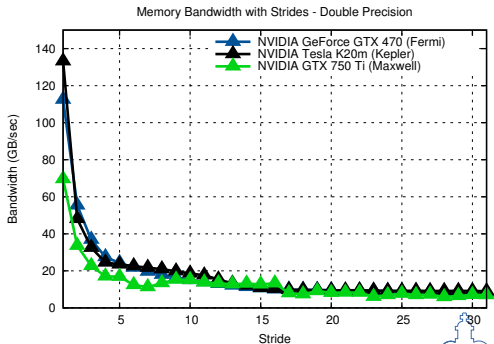
Memory Bandwidth with Offsets - Double Precision



# 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];  
}
```

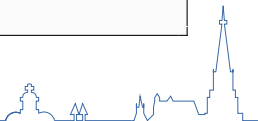


## Strided Memory Access

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;
}
```



## Strided Memory Access

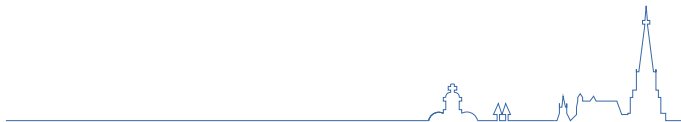
Workaround: Structure of Arrays

```
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;
}
```



Software: OpenCL





# History of OpenCL

## Prior to 2008

OpenCL developed by Apple Inc.

## 2008

OpenCL working group formed at Khronos Group  
OpenCL specification 1.0 released

## 2010

OpenCL 1.1 (multi-device, subbuffer manipulation)

## 2011

OpenCL 1.2 (device partitioning)

## 2013

OpenCL 2.0 (shared virtual memory, SPIR, etc.)



**OpenCL**



## Similar to CUDA

Kernel language is a subset of C

Explicit memory management, host-device transfers

Memory model: local, shared, global

## Different from CUDA

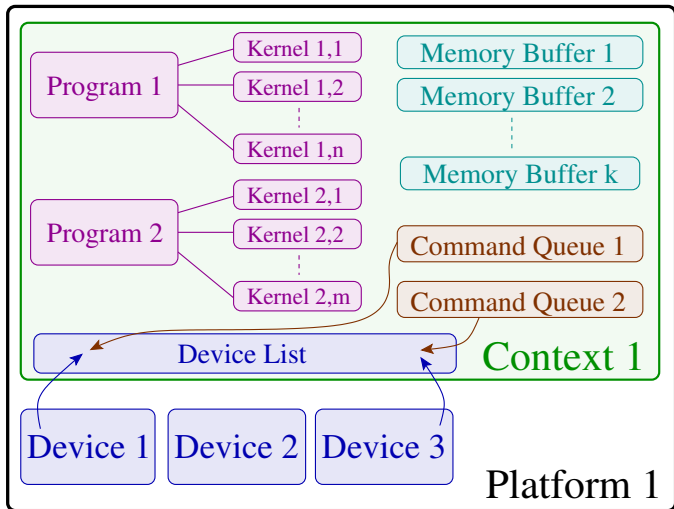
Support by many vendors

No compiler-wrapper, only a shared library

Kernel compilation usually at runtime



# OpenCL Platform Model



## OpenCL Thread Control (1D) vs. CUDA

Local ID in block: `get_local_id(0)`

`threadIdx.x`

Threads per block: `get_local_size(0)`

`blockDim.x`

ID of block: `get_group_id()`

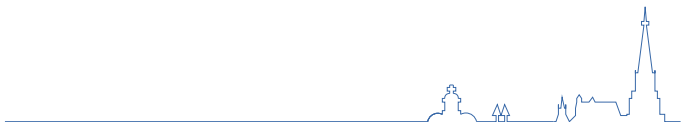
`blockIdx.x`

No. of blocks: `get_num_groups()`

`gridDim.x`

Global thread ID: `get_global_id()`

No. of threads: `get_global_size()`



## Software: Parallel Primitives



## Reductions

Use  $N$  values to compute 1 result value

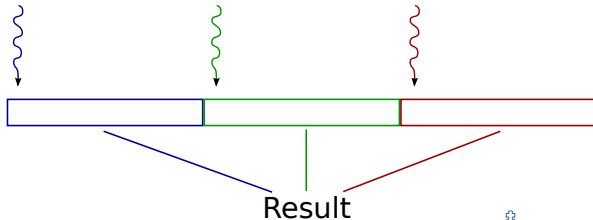
Examples: Dot-products, vector norms, etc.

## Reductions with Few Threads

Decompose  $N$  into chunks for each thread

Compute chunks in parallel

Merge results with single thread

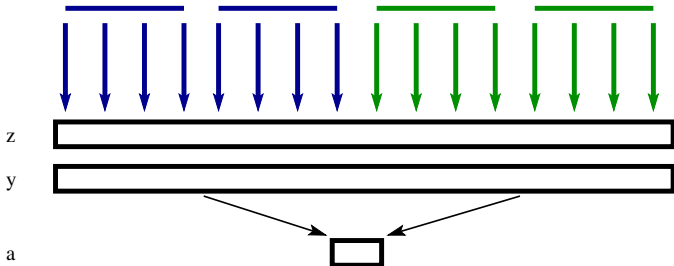


## Reductions with Many Threads

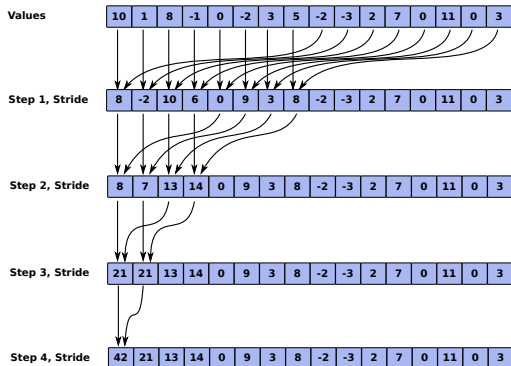
Decompose  $N$  into chunks for each workgroup

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];
}
```



## 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

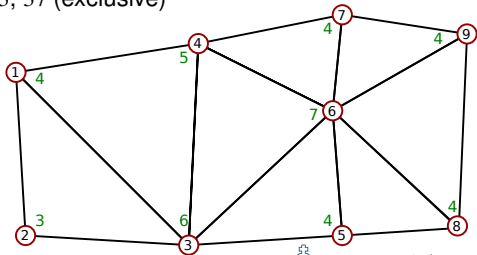
y: 4, 7, 13, 18, 22, 29, 33, 37, 41 (inclusive)

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

## Applications

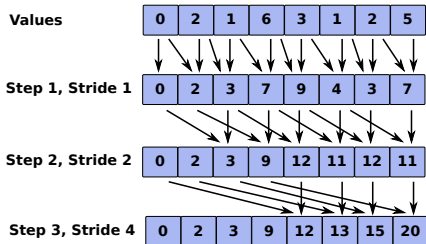
Sparse matrix setup

Graph algorithms



# Parallel Primitives

## 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;
```

## 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

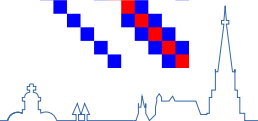
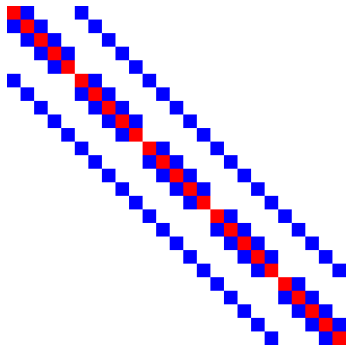
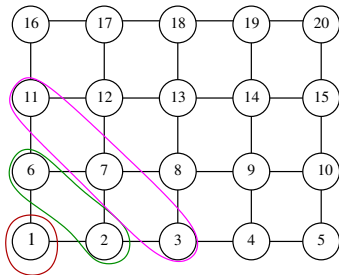
$$\begin{pmatrix}
 5 & \times & \times & \times & & \times & \times & & & \\
 \times & 3 & \times & & & & & & & \\
 \times & \times & 4 & \times & & & & & & \\
 \times & \times & \times & 5 & \times & \times & & & & \times \\
 \times & & \times & \times & 5 & \times & & \times & \times & \\
 \times &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

## ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d

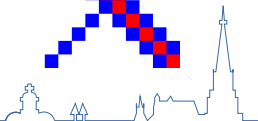
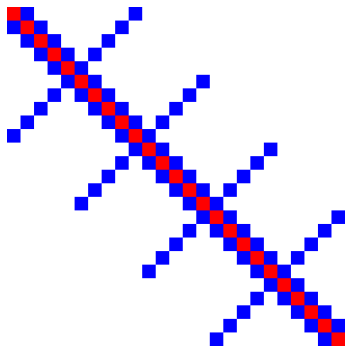
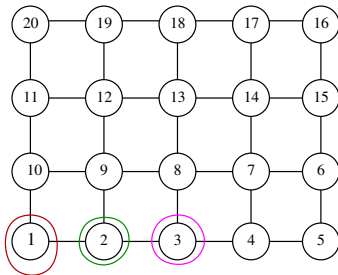


## ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d

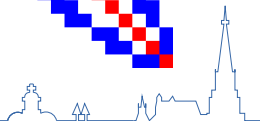
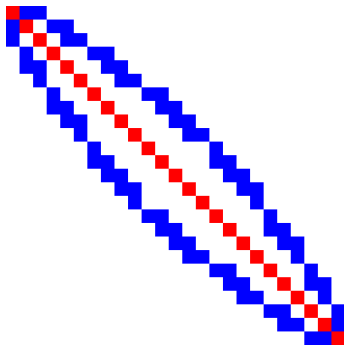
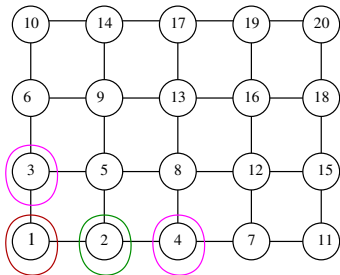


## ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d

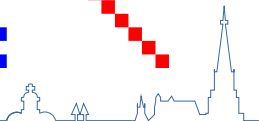
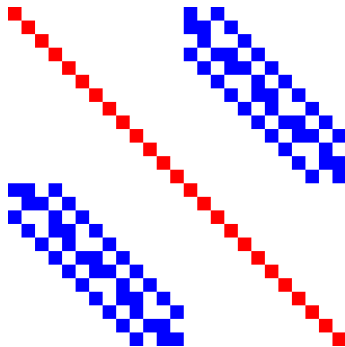
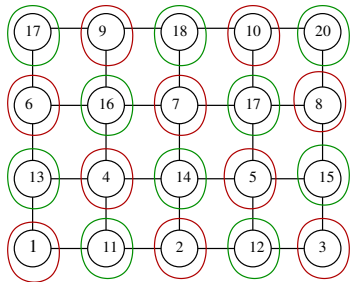


## ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d





## 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, cIAMG, ...

FFT: cuFFT, cIFFT, FFTW, ...

Primitives: VexCL, Boost.Compute, ...

Machine Learning: Caffe, cuDNN, ...



## Performance Modeling: Bottlenecks



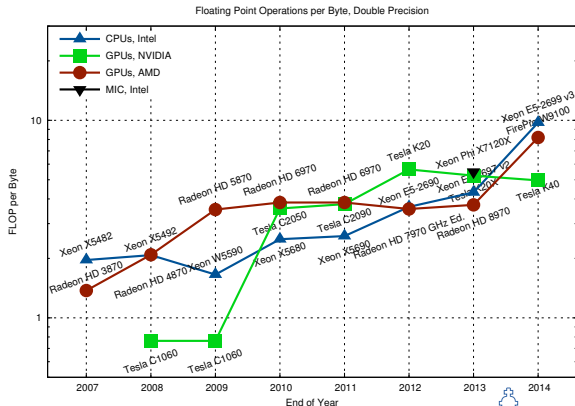
# Bottleneck Potpourri

## Arithmetic Intensity

Number of FLOPs per Byte

FLOP-limited: AI larger than 1-10

Memory-limited: AI smaller than 1-10



## Latency

Bottleneck in strong scaling limit

Ultimate limit for time stepping

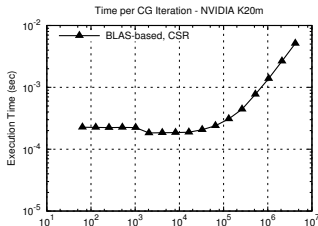
## Latency - Sources

Network latency (Ethernet  $\sim 20\mu\text{s}$ , Infiniband  $\sim 5\mu\text{s}$ )

PCI-Express latency (Kernel launches,  $\sim 10\mu\text{s}$ )

Thread synchronization (barriers, locks,  $\sim 1 - 10\mu\text{s}$ )

Memory latency ( $\sim 100\text{ns}$ )



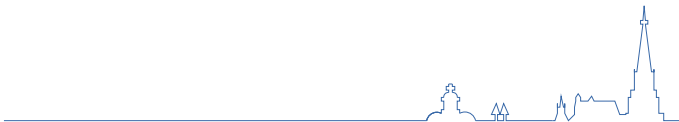
## Load Imbalance

Total execution time determined by slowest thread

Focus on making the slowest thread fast

Easy for static data structures (e.g. dense matrices)

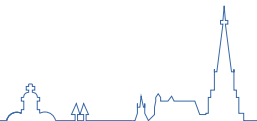
Hard for dynamic data structures (e.g. sparse matrices)



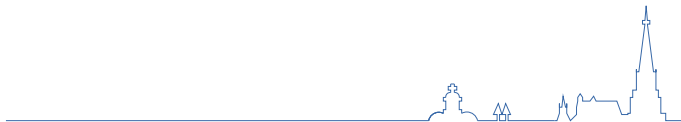
## Amdahl's Law

$$T_{\text{total}} = T_{\text{serial}} + T_{\text{parallel}} / \#\text{processors}$$

Speed-up limited by serial portion of an algorithm



## Performance Modeling: Examples

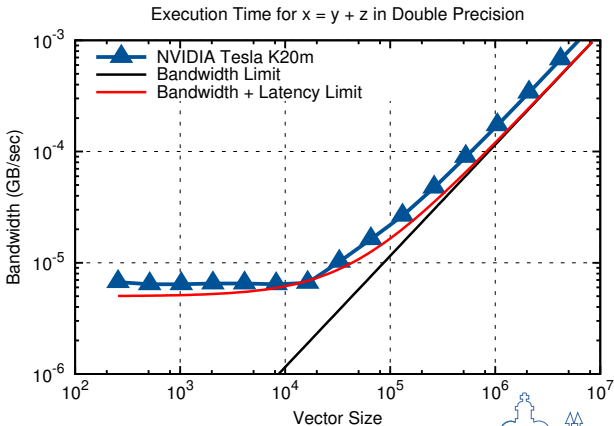


## Vector Addition

$x = y + z$  with  $N$  elements each

1 FLOP per 24 byte in double precision

Limited by memory bandwidth  $\Rightarrow T_2(N) \stackrel{?}{\approx} 3 \times 8 \times N / \text{Bandwidth} + \text{Latency}$



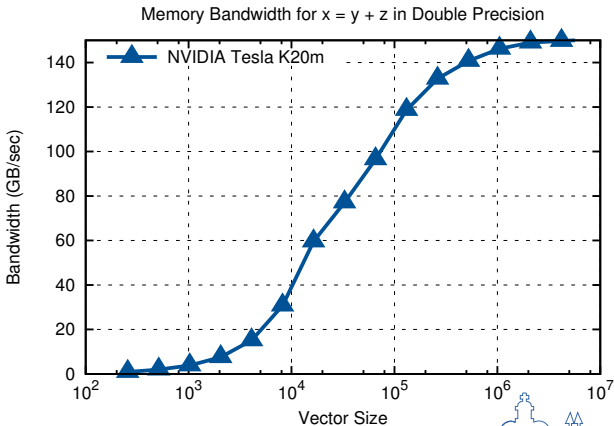


## Vector Addition

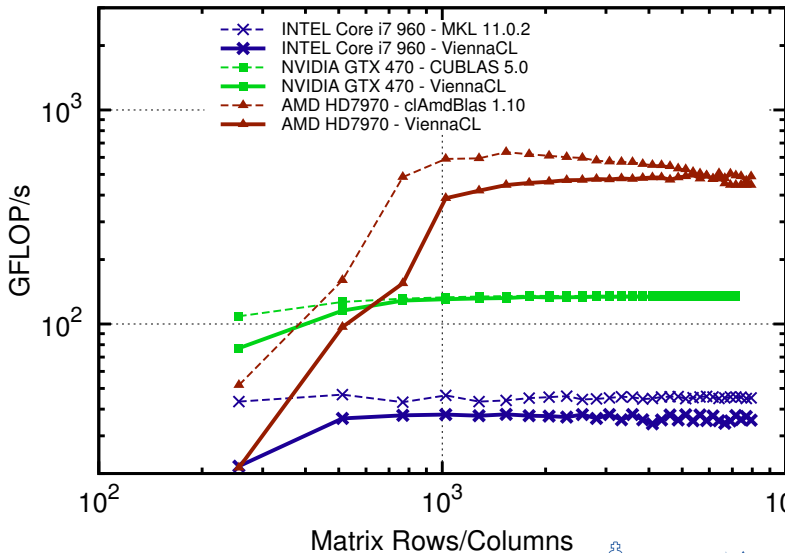
$x = y + z$  with  $N$  elements each

1 FLOP per 24 byte in double precision

Limited by memory bandwidth  $\Rightarrow T_2(N) \stackrel{?}{\approx} 3 \times 8 \times N / \text{Bandwidth} + \text{Latency}$



# Performance Modeling: Matrix-Matrix Products



## 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

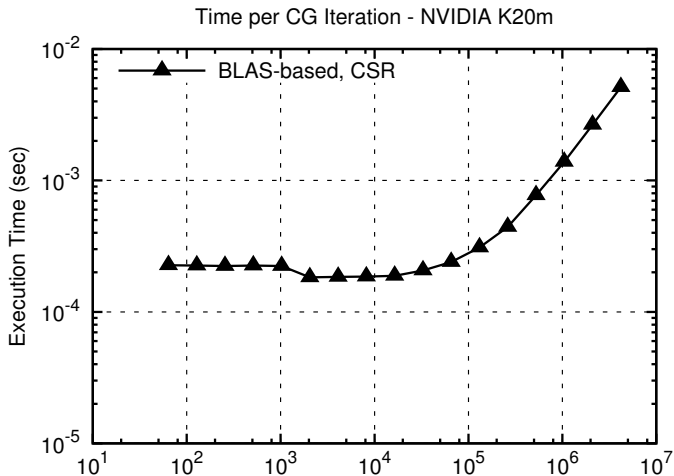
For  $i = 0$  until convergence

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

EndFor



# Performance Modeling: Conjugate Gradients



(2D Finite Difference Discretization)

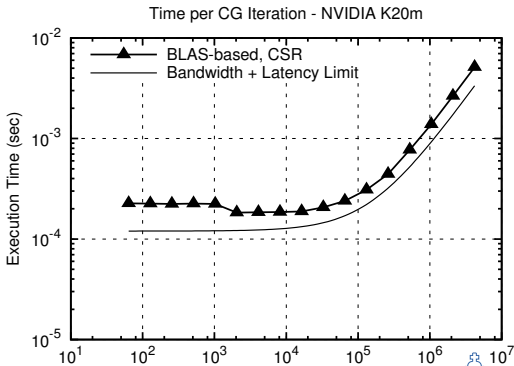
## Performance Modeling

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)

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



(2D Finite Difference Discretization)

## Optimization: Rearrange the algorithm

- Remove unnecessary reads

- Remove unnecessary synchronizations

- Use custom kernels instead of standard BLAS



## 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

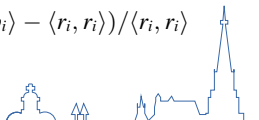
Choose  $x_0$

$$p_0 = r_0 = b - Ax_0$$

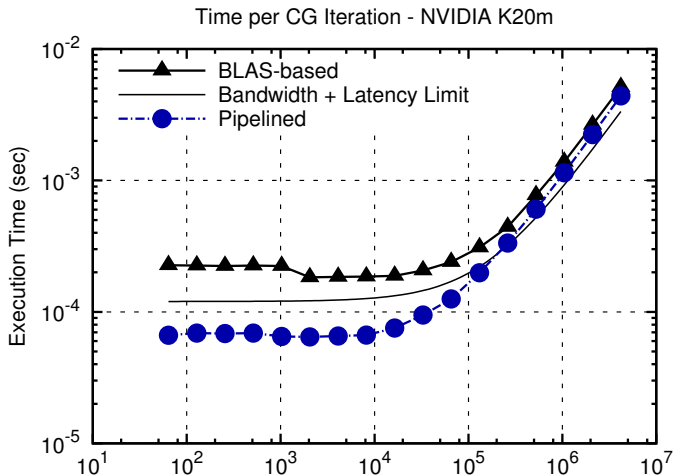
For  $i = 1$  until convergence

1.  $i = 1$ : Compute  $\alpha_0, \beta_0, Ap_0$
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



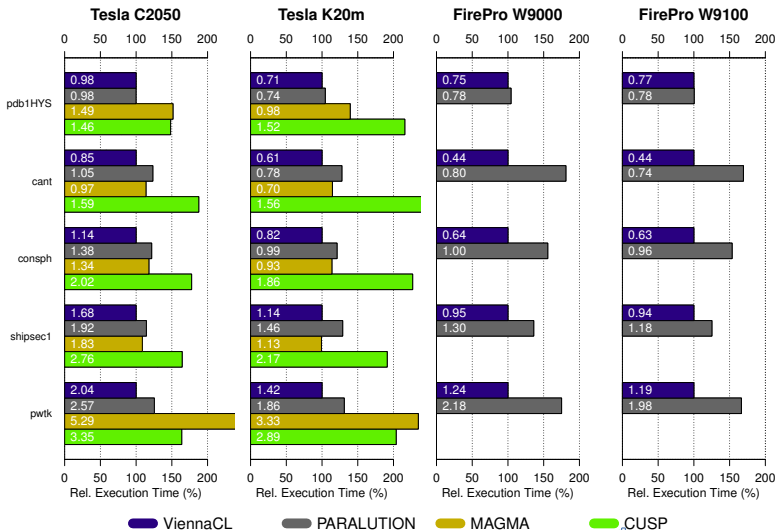
# Performance Modeling: Conjugate Gradients



(2D Finite Difference Discretization)



## Benefits of Pipelining also for Large Matrices



## Sparse Matrix Transposition

Compute  $B = A^T$

$A$  and  $B$  using sparse storage schemes

## Row-Wise Datastructures

```
std::vector<std::map<int, double> >
```

```
std::vector<boost::flat_map<int, double> >
```

Compressed Sparse Row



## Simplest Case: STL

```
for (int i=0; i<N; ++i)
  for (auto col = A[i].begin(); col != A[i].end(); ++col)
    B[col->index][i] = col->value
```

## More Work: CSR

Compute number of nonzeros per row in  $B$

Allocate data arrays for  $B$

Exclusive scan to find entry points per row

Populate  $B$

## Performance Modeling

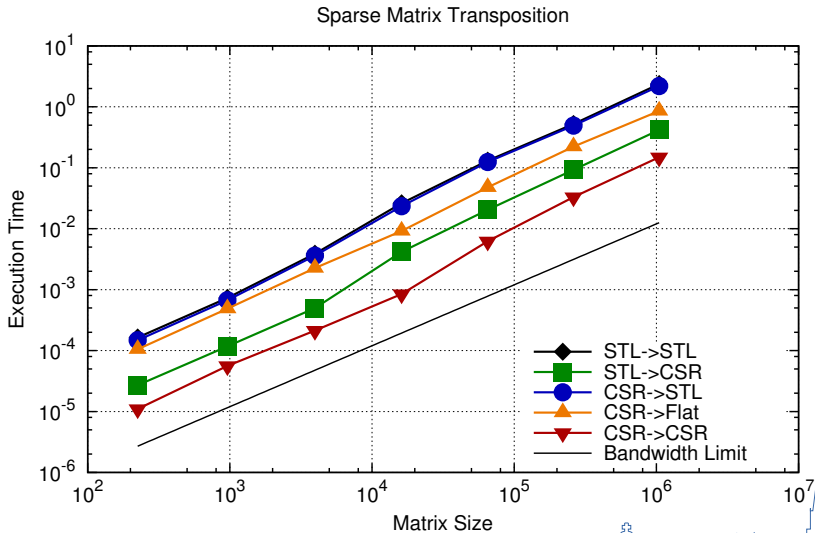
Read and write each nonzero entry at least once

Low arithmetic intensity: Bandwidth limited!

$$T(N) = 2 \times (4 + 8) \times \text{nnz}(N) / \text{Bandwidth}$$



# Performance Modeling: Sparse Matrix Transpose



## Many-Core Architectures

GPUs: 100s and 1000s of threads

MIC: 61 cores (KNC), 72 cores (KNL)

2-3x enhancements (perf/Watt) for some applications

Amdahl's Law limits general applicability

## Recommendations

(Re-)Use software libraries

Profile and model before you optimize

Pay attention to memory transfers

