Performance Tuning for GPUs - An Iterative Process

Karl Rupp[1,2]

rupp@iue.tuwien.ac.at

@karlrupp

with contributions from
Philippe Tillet[1], Florian Rudolf[1],
Josef Weinbub[1], Ansgar Jüngel[2], Tibor Grasser[1]
(based on stimuli from PETSc+ViennaCL users)

[1] Institute for Microelectronics, TU Wien, Austria
[2] Institute for Analysis and Scientific Computing, TU Wien, Austria

LANS Seminar at the Argonne National Laboratory, June 2nd, 2014

Outline

Introduction

Conjugate Gradient - Standard Formulation

Conjugate Gradient - Kernel Parameters

Parameter Study for Portable Performance

Conjugate Gradient - Pipelining

GMRES Optimizations

Conclusion

Introduction

Positions

- PhD student at TU Wien (2009-2011)
- Postdoc at ANL (09/2012-09/2013)
- Postdoc at TU Wien (01/2012-09/2012, 09/2013-current)

Research Interests

- Semiconductor device simulation
- Numerical solution of PDEs
- Parallel computing

Software Development

- PETSc
- ViennaCL
- ViennaSHE
- ...

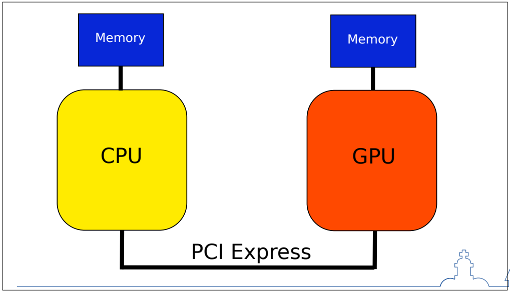
Introduction

Iterative Solvers

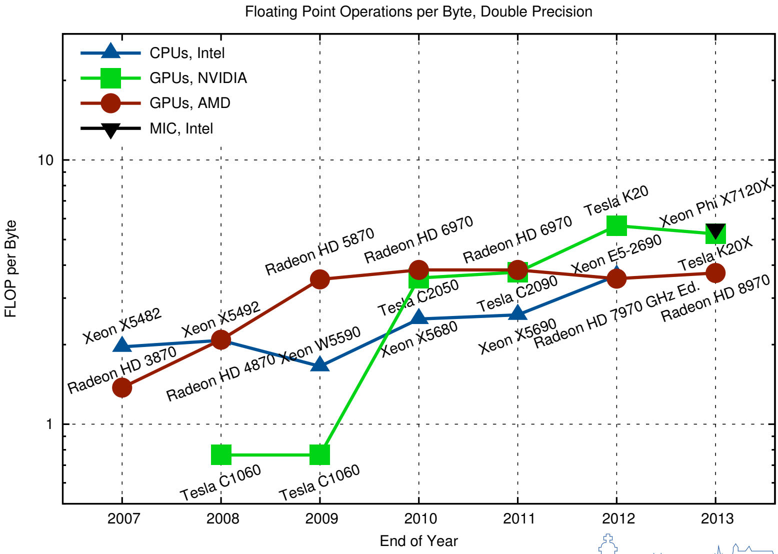
- Matrix-vector products and vector operations only
- Expose more fine-grained parallelism
- Preconditioners often desirable

Accelerators (CUDA, OpenCL)

- Graphics processing units (GPUs)
- Intel Xeon Phi



Introduction



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

-

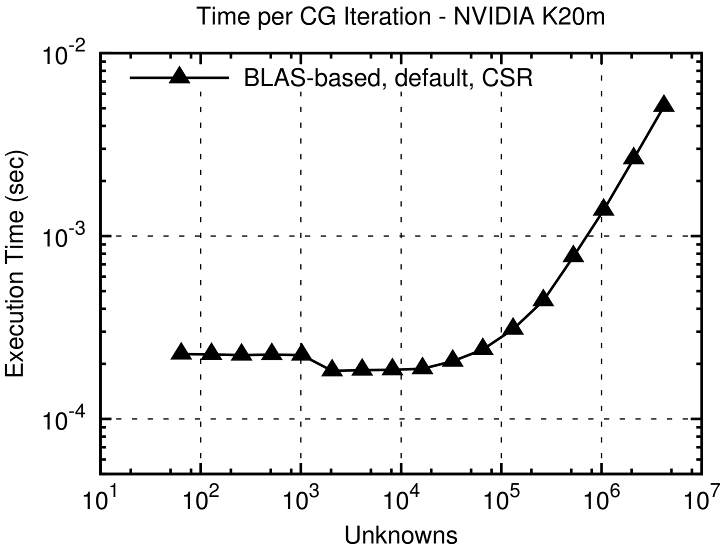
SpMV, AXPY

For $i = 0$ until convergence

1. SpMV $\leftarrow$ 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

Conjugate Gradients

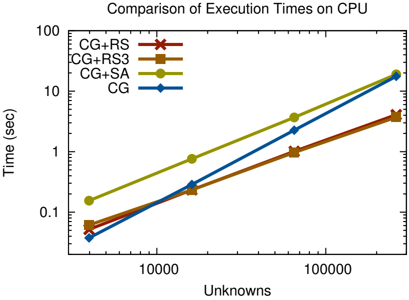


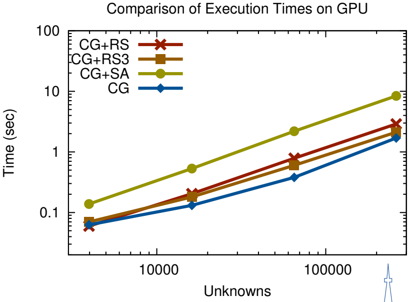
(2D Finite Difference Discretization)

Implications

Kernel launches expensive

Delicate balance for preconditioners





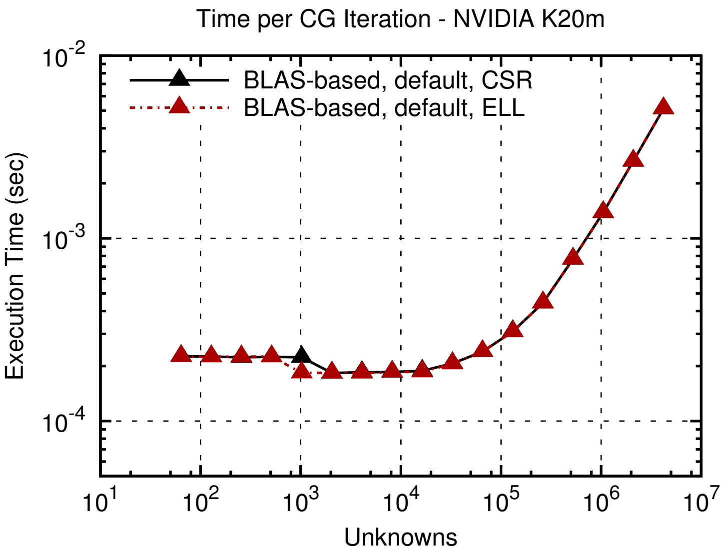
Conjugate Gradient Optimizations

Optimization 1

- Get best performance out of SpMV
- Compare different sparse matrix types

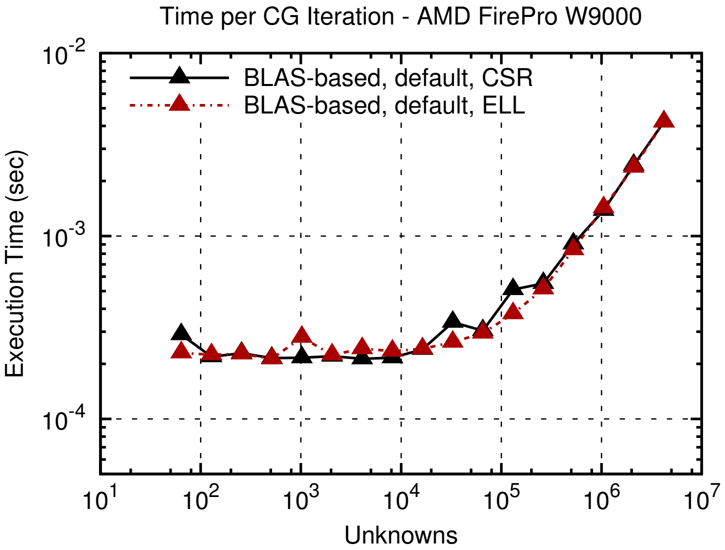
Cf.: N. Bell: Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proc. SC '09*

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradient Optimizations

Optimization 2

Optimize kernel parameters for each operation

Outline

Introduction

Conjugate Gradient - Standard Formulation

Conjugate Gradient - Kernel Parameters

Parameter Study for Portable Performance

Conjugate Gradient - Pipelining

GMRES Optimizations

Conclusion

Benchmark Setting

Scope for OpenCL-based Portability Study

- Vector and matrix-vector operations (BLAS levels 1 and 2)
- Limited by memory bandwidth

Benchmark Setting

Scope for OpenCL-based Portability Study

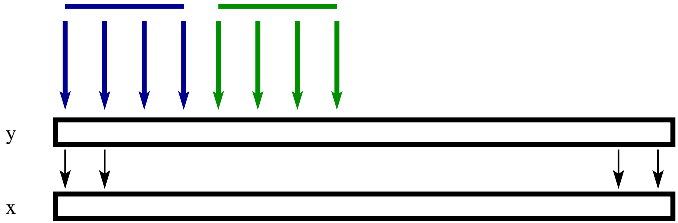
- Vector and matrix-vector operations (BLAS levels 1 and 2)
- Limited by memory bandwidth

Key Question (Memory-Bandwidth-Limited Kernels)

Good performance of complicated kernels
by optimizing the simplest kernel?

Vector Assignment (Copy) Kernel

$x \Leftarrow y$ for (large) vectors x, y



Parameters (1900 variations)

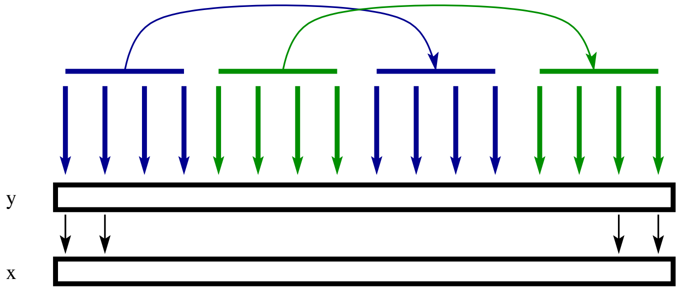
Local work size, global work size

Vector types (float1, float2, ... , float16)

Thread increment type

Vector Assignment (Copy) Kernel

$x \Leftarrow y$ for (large) vectors x, y

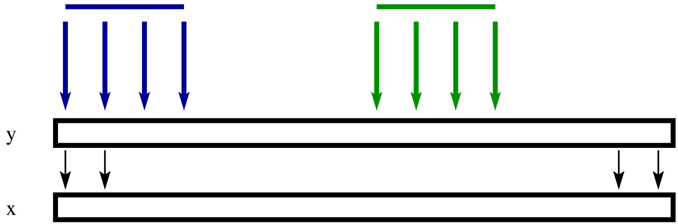


Parameters (1900 variations)

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

Vector Assignment (Copy) Kernel

$x \Leftarrow y$ for (large) vectors x, y

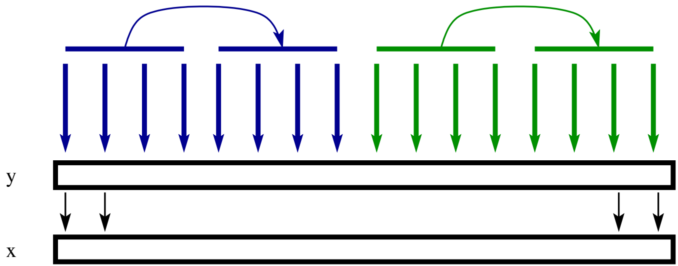


Parameters (1900 variations)

```
for (size_t i = group_start + get_local_id(0);
     i < group_end; i+= get_local_size(0))
  x[i] = y[i];
```

Vector Assignment (Copy) Kernel

$x \Leftarrow y$ for (large) vectors x, y



Parameters (1900 variations)

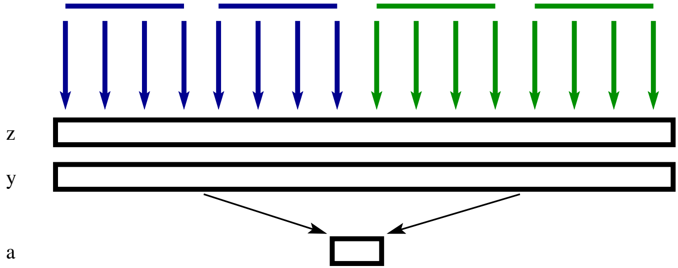
```
for (size_t i = group_start + get_local_id(0);
     i < group_end; i+= get_local_size(0))
  x[i] = y[i];
```

Benchmark Setting

Operations

Vector copy, vector addition, inner product

Matrix-vector product



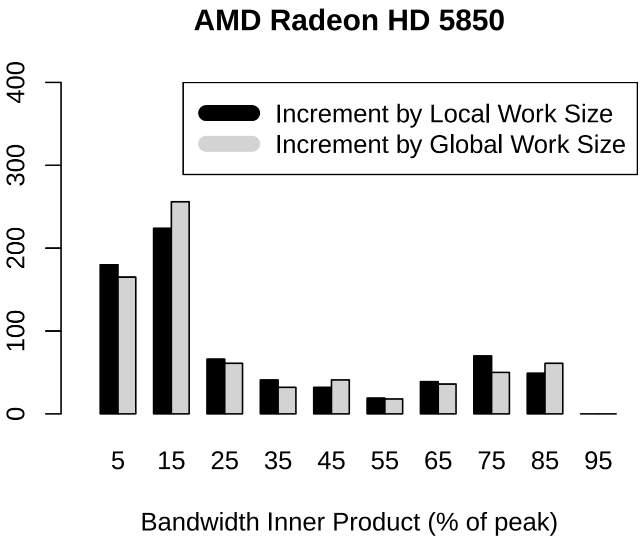
Devices

AMD: A10-5800 APU, HD 5850 GPU

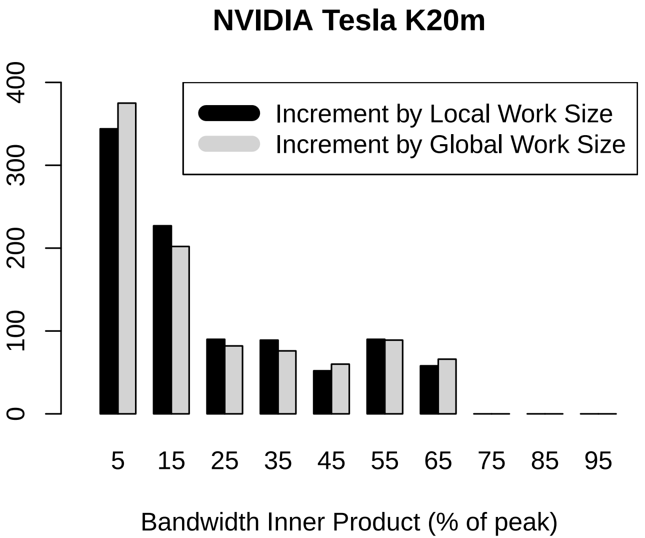
INTEL: Dual Socket Xeon E5-2670, Xeon Phi

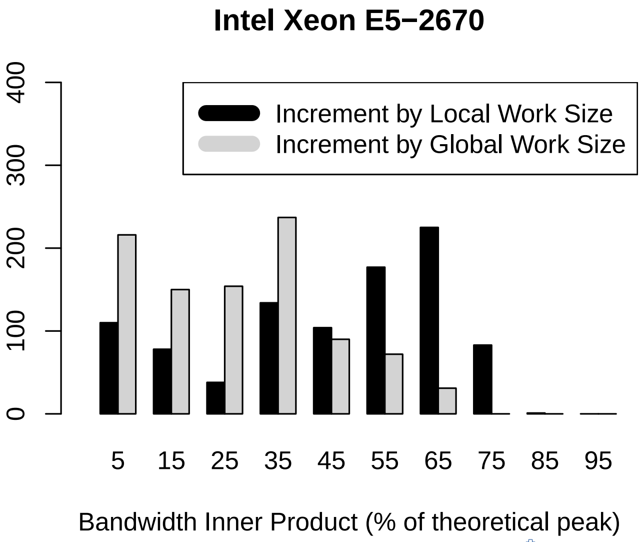
NVIDIA: GTX 285, Tesla K20m

Histograms

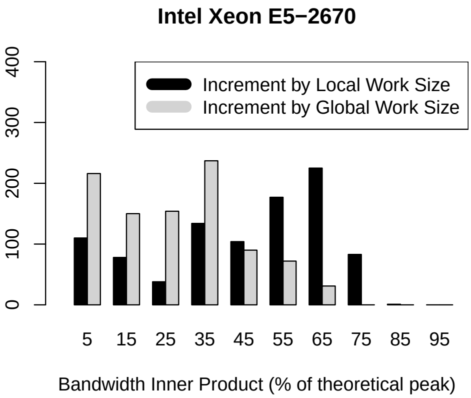

AMD Radeon HD 5850
400
300
200
100
0
Increment by Local Work Size
Increment by Global Work Size
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)

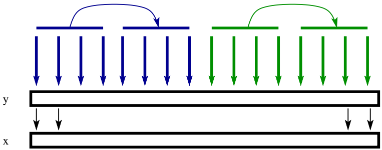
Benchmark


NVIDIA Tesla K20m
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)

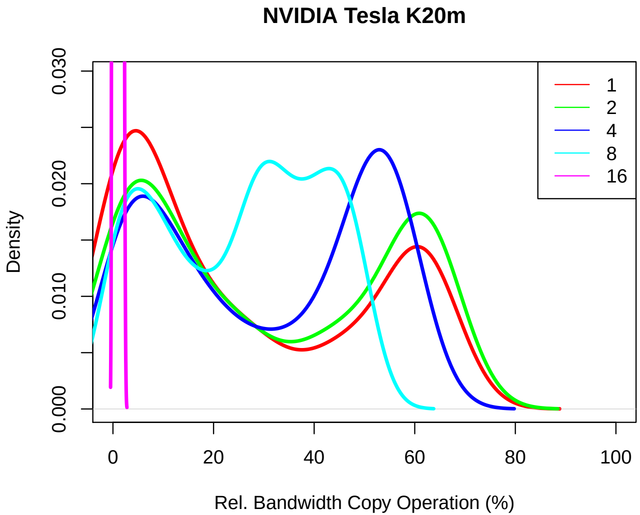

Intel Xeon E5−2670
0
100
200
300
400
Increment by Local Work Size
Increment by Global Work Size
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of theoretical peak)

Benchmark


Intel Xeon E5−2670
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of theoretical peak)

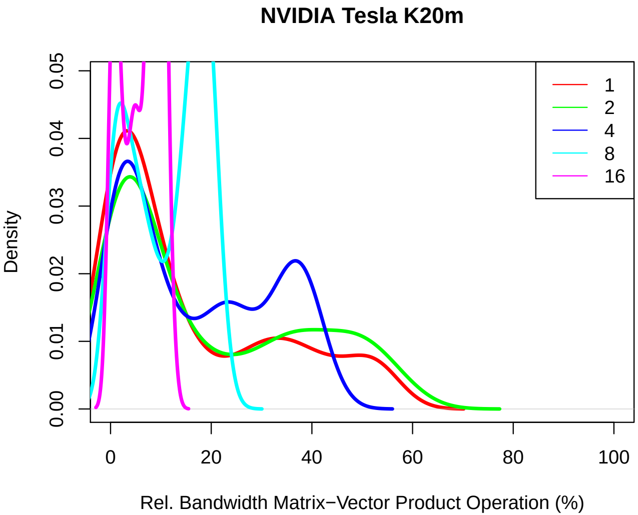

y
x

Benchmark



(comparison of vector types `double`, `double2`, `double4`, `double8`, `double16`)

Benchmark

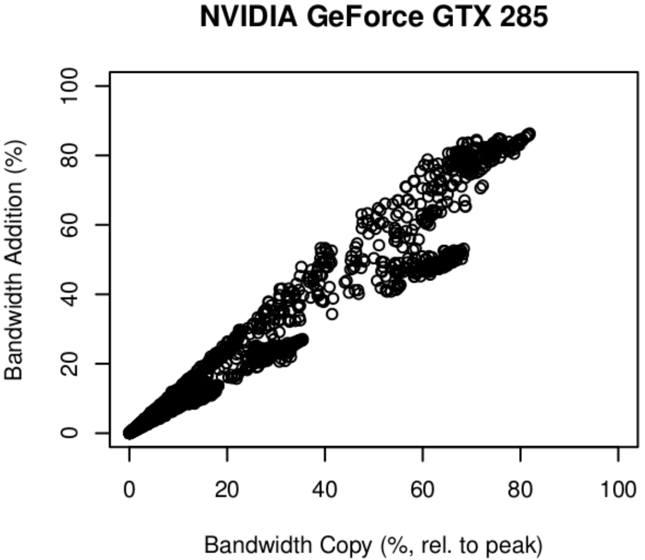


(comparison of vector types `double`, `double2`, `double4`, `double8`, `double16`)

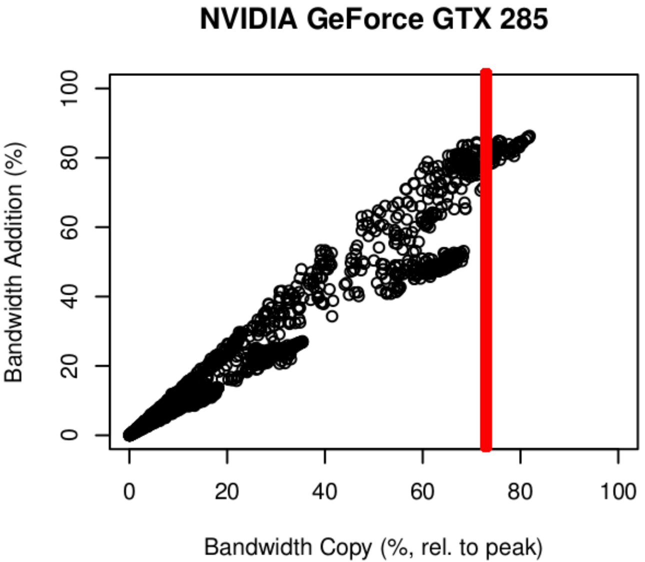
[Addition|Inner Product|Matrix-Vector] vs. Copy Kernel

Same Device

Benchmark

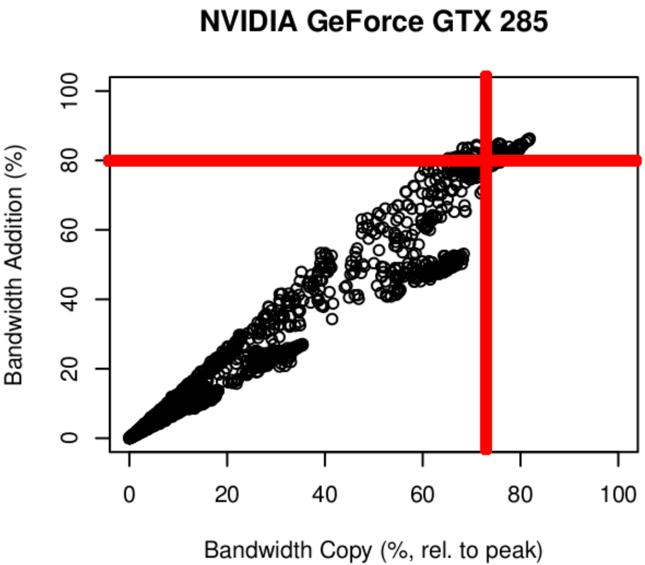

NVIDIA GeForce GTX 285
Bandwidth Addition (%)
0
20
40
60
80
100
Bandwidth Copy (%, rel. to peak)

Benchmark



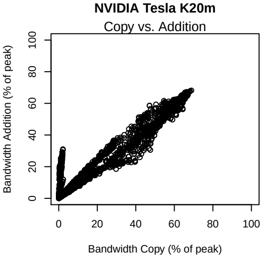
NVIDIA GeForce GTX 285
Bandwidth Addition (%)
0
20
40
60
80
100
Bandwidth Copy (%, rel. to peak)
0
20
40
60
80
100

Benchmark

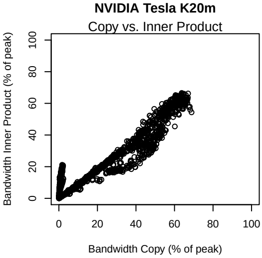

NVIDIA GeForce GTX 285
Bandwidth Addition (%)
0
20
40
60
80
100
Bandwidth Copy (%, rel. to peak)
0
20
40
60
80
100

NVIDIA Tesla K20m

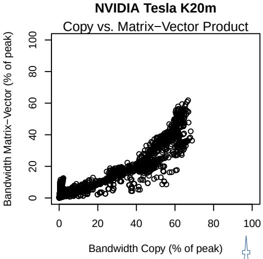
Addition



Inner Product

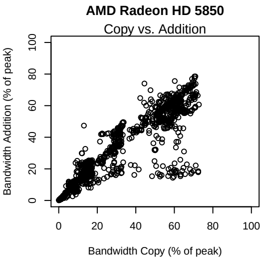


Mat-Vec Product

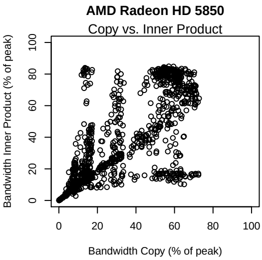


AMD Radeon HD 5850

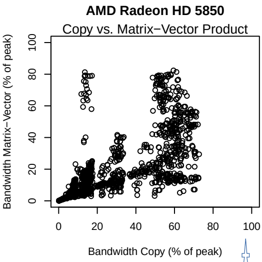
Addition



Inner Product

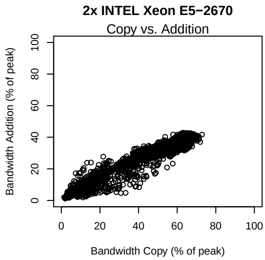


Mat-Vec Product

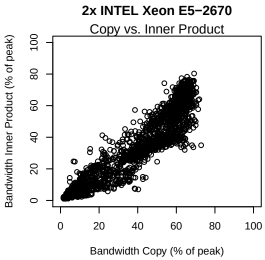


INTEL Dual Xeon E5-2670

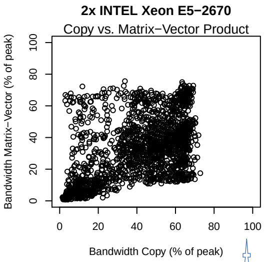
Addition



Inner Product

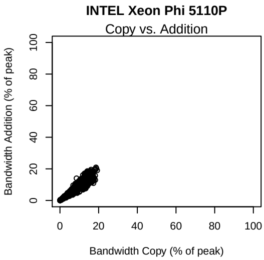


Mat-Vec Product

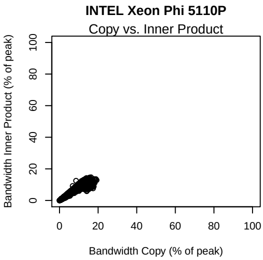


INTEL Xeon Phi

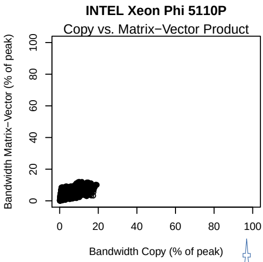
Addition



Inner Product



Mat-Vec Product



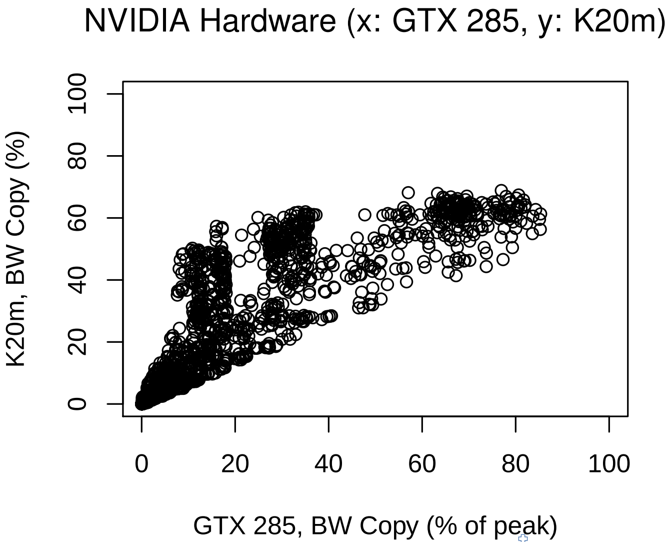
Benchmark

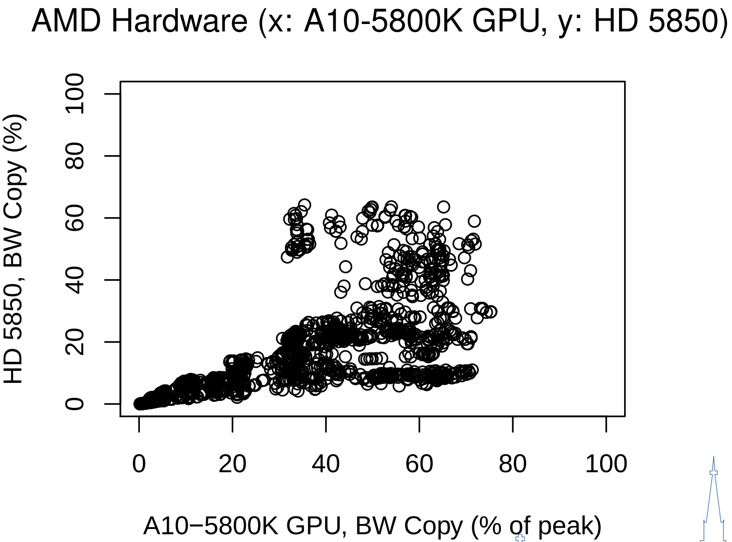
Conclusio:

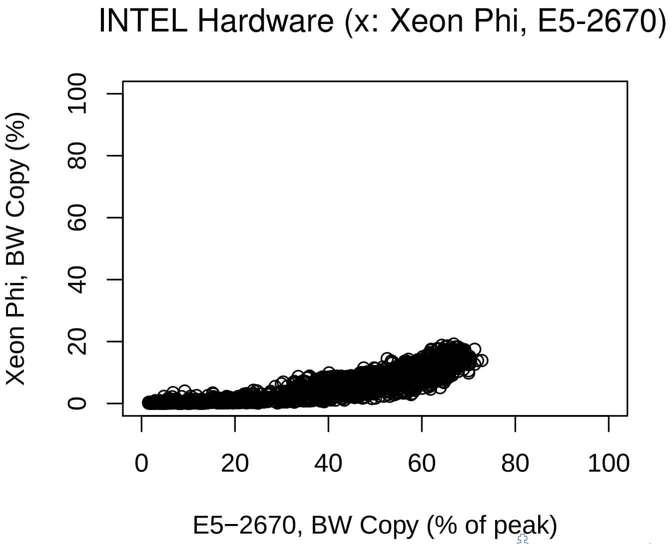
Focus on fastest configurations for copy-kernel sufficient

[Copy|Addition|Inner Product|Matrix-Vector] vs. Copy Kernel

Different Device, Same Vendor

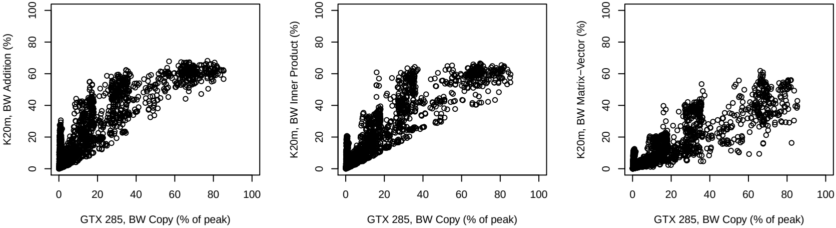

NVIDIA Hardware (x: GTX 285, y: K20m)
K20m, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
GTX 285, BW Copy (% of peak)


AMD Hardware (x: A10-5800K GPU, y: HD 5850)
HD 5850, BW Copy (%)
0
20
40
60
80
100
A10−5800K GPU, BW Copy (% of peak)

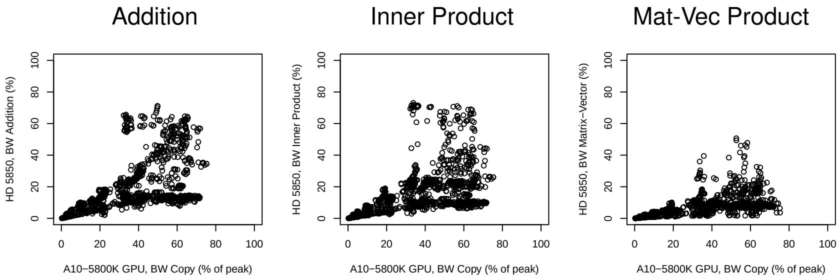

INTEL Hardware (x: Xeon Phi, E5-2670)
Xeon Phi, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
E5−2670, BW Copy (% of peak)

NVIDIA Hardware (x: GTX 285, y: K20m)

Addition | Inner Product | Mat-Vec Product



AMD Hardware (x: A10-5800K GPU, y: HD 5850)

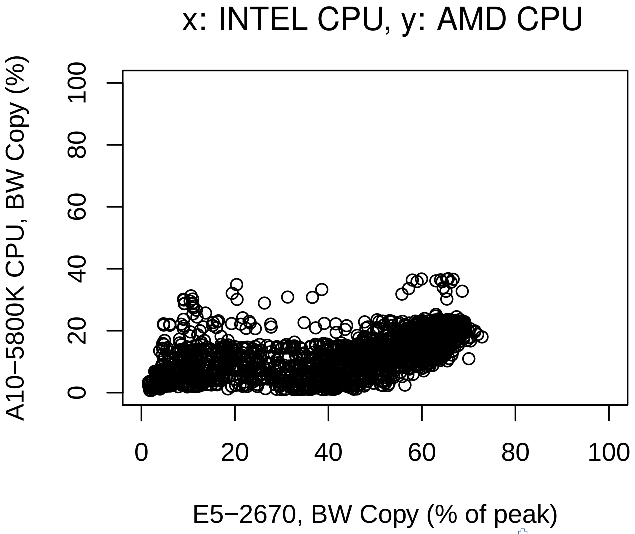


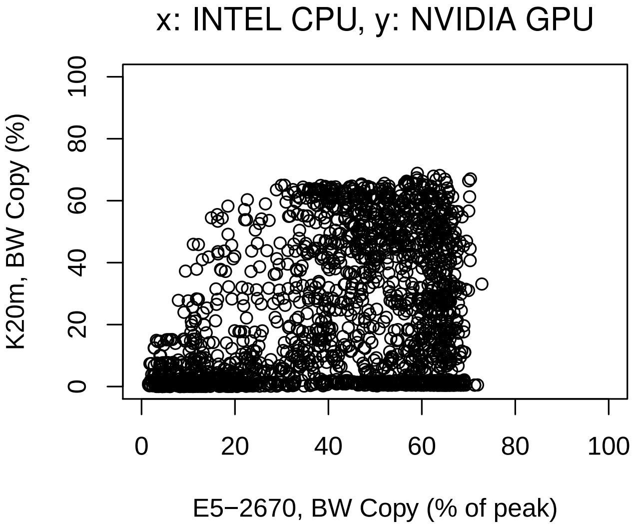
Conclusio:

Certain Performance Portability per Vendor

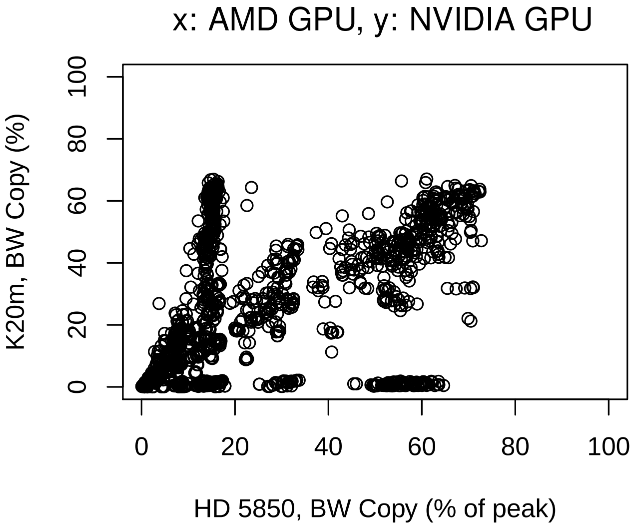
[Copy|Addition|Inner Product|Matrix-Vector] vs. Copy Kernel

Different Device, Different Vendor


x: INTEL CPU, y: AMD CPU
A10−5800K CPU, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
E5−2670, BW Copy (% of peak)


x: INTEL CPU, y: NVIDIA GPU
K20m, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
E5−2670, BW Copy (% of peak)

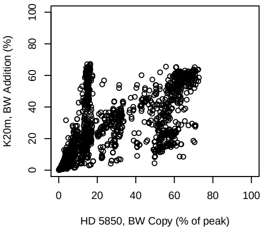
Benchmark


x: AMD GPU, y: NVIDIA GPU
K20m, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
HD 5850, BW Copy (% of peak)

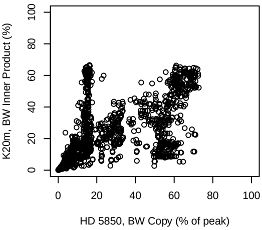
Benchmark

x: AMD HD 5850, y: NVIDIA K20m

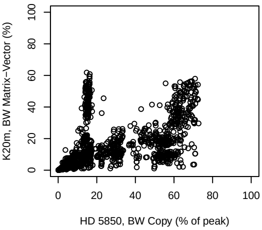
Addition



Inner Product



Mat-Vec Product



Conclusio:

Fast Configurations Across Vendors Exist

Outline

Introduction

Conjugate Gradient - Standard Formulation

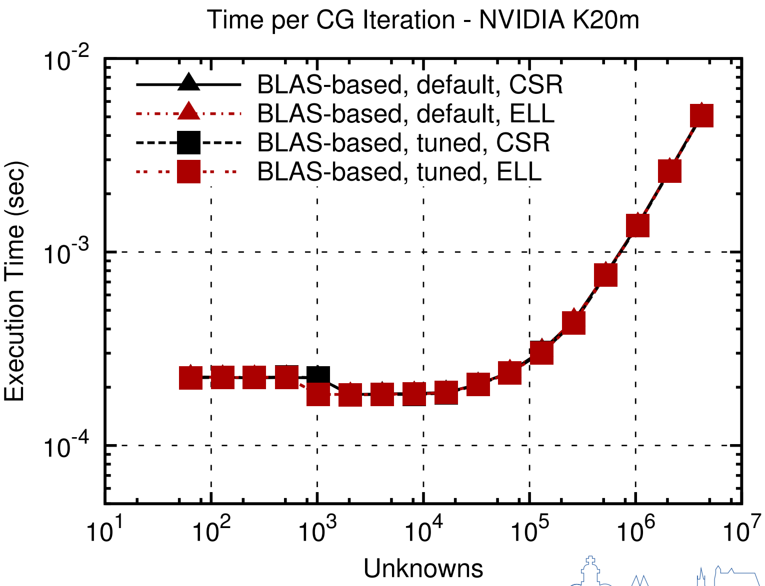
Conjugate Gradient - Kernel Parameters

Parameter Study for Portable Performance

Conjugate Gradient - Pipelining

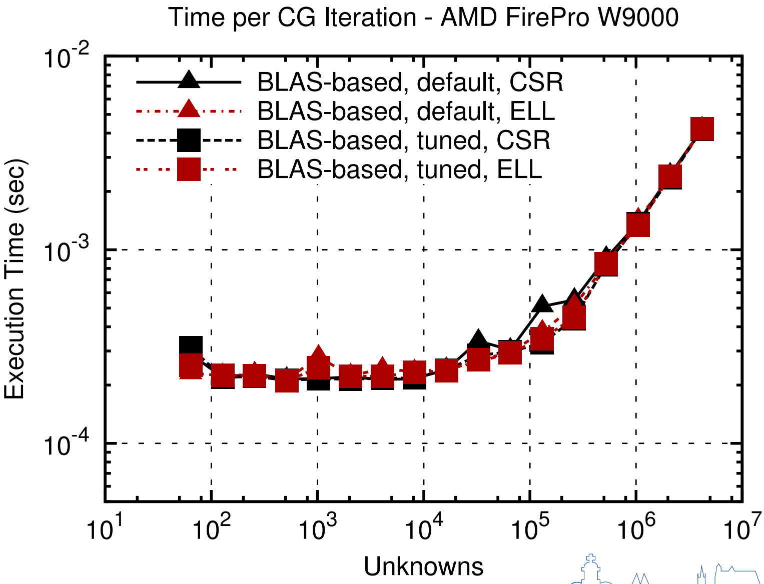
GMRES Optimizations

Conclusion



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Optimization 3: Rearrange the algorithm

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

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. $p_{i+1} = r_{i+1} + \beta_i p_i$

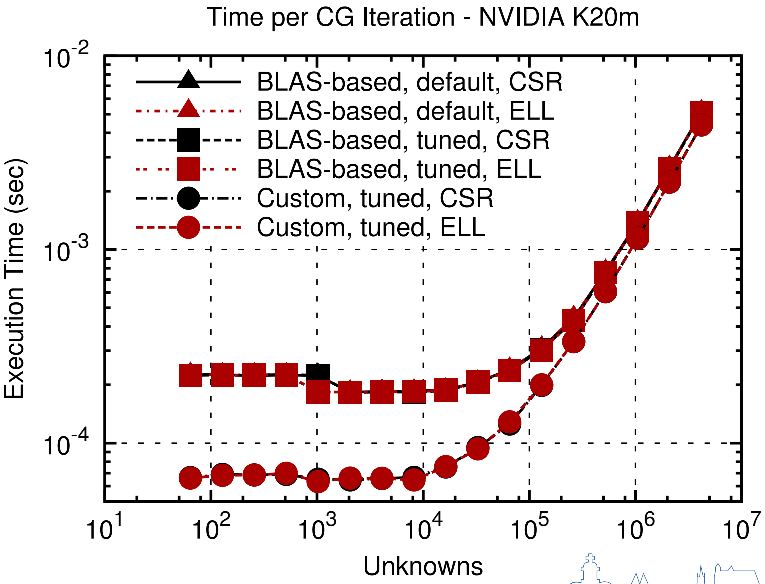
EndFor

Pipelined CG

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

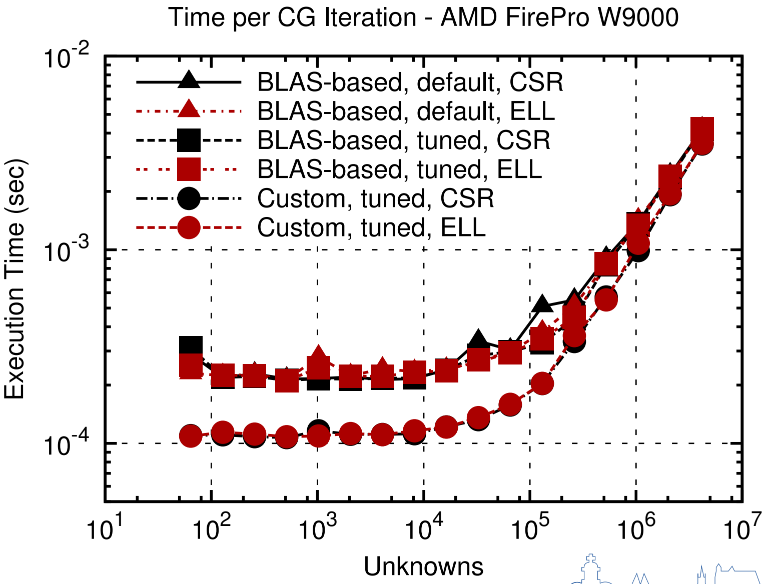
1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i,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



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

GMRES Optimization

Generalized Minimum Residual (GMRES) Method

Krylov space $\mathrm{span}\{r, Ar, A^2r, \ldots, A^{N-1}r\}$

Orthogonal basis $\{v_1, v_2, \ldots, v_N\}$

Gram-Schmidt Method revisited

Given: orthonormal basis $\{v_1, v_2, \ldots, v_N\}$, augment by w

$w \leftarrow w - \sum_{i=1}^{N} \langle w, v_i \rangle v_i$

$w \leftarrow w / \|w\|$

Add w to basis

Multiple inner products $\langle w, v_i \rangle$

Performance critical (global reductions)

Reuse of w desirable

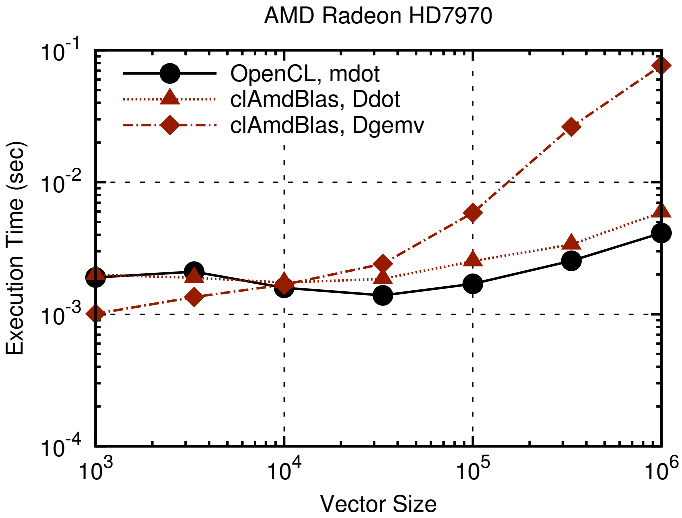
Custom routine *mdot*

- Process $\alpha_i = \langle w, v_i \rangle$ in batches
- Batch sizes 1, 2, 3, 4, 8
- Batch size 8: Only 12.5% overhead vs. arbitrary batch sizes

Code sketch (Batch size 4)

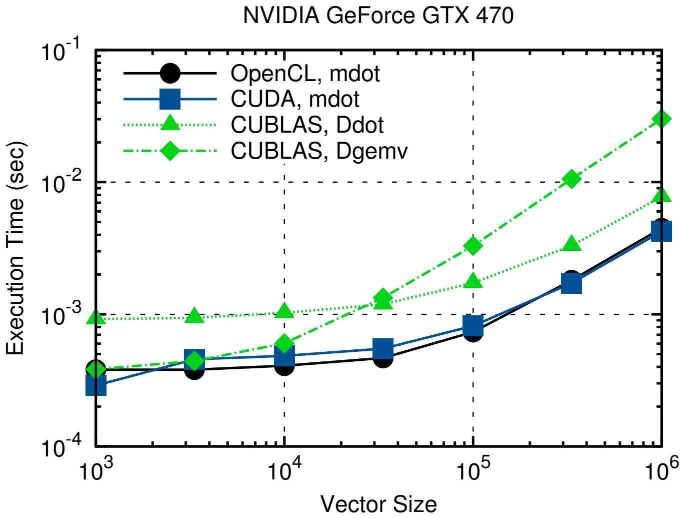
```
for (size_t i=thread_id; i<M; i += threads_per_group)
{
  double val_w = w[i];
  alpha_1 += val_w * v1[i];
  alpha_2 += val_w * v2[i];
  alpha_3 += val_w * v3[i];
  alpha_4 += val_w * v4[i];
}
```

Benchmarks



Fixed number of 50 vectors

Benchmarks



Fixed number of 50 vectors

Summary and Conclusion

Conjugate Gradient Method

- Careful choice of sparse matrix format
- Tune kernels to target device
- Minimize reads from global memory (kernel fusion, pipelining)

Generalized Minimum Residual Method (GMRES)

- Minimizes reads from global memory (mdot kernel)
- Up to twice the performance of ’naive’ implementations

Implications

- Tune primarily with memory transfers in mind
- Prefer regular memory access patterns
- Use *appropriate* vector data types