An Overview of GPU-accelerated Routines and Implementation Techniques in ViennaCL

Karl Rupp[1,2]

rupp@iue.tuwien.ac.at

https://karlrupp.net/

https://github.com/karlrupp/slides/

@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

Lugano, September 16th, 2015

Introduction

Positions

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

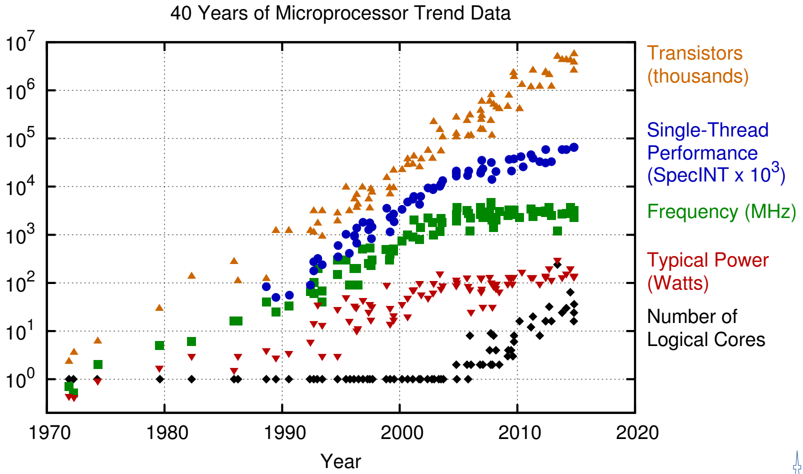
Research Interests

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

Software Development

- PETSc
- ViennaCL
- ViennaSHE
- ...

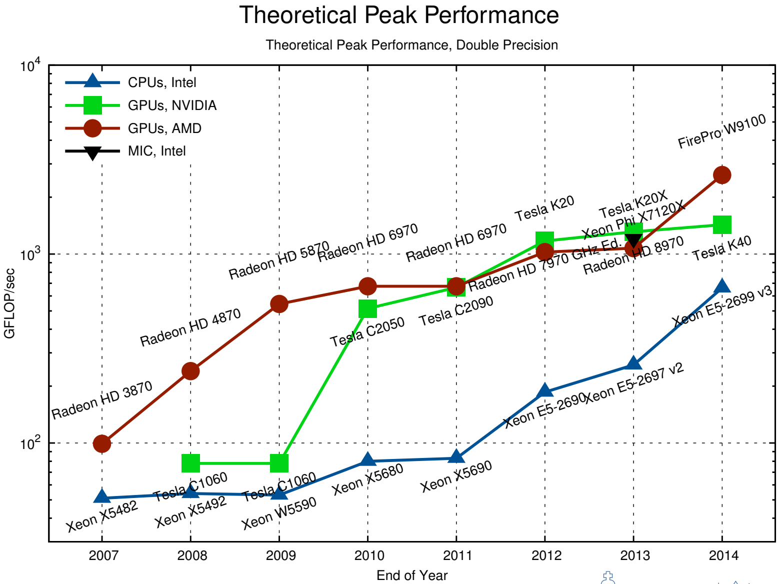
Introduction



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

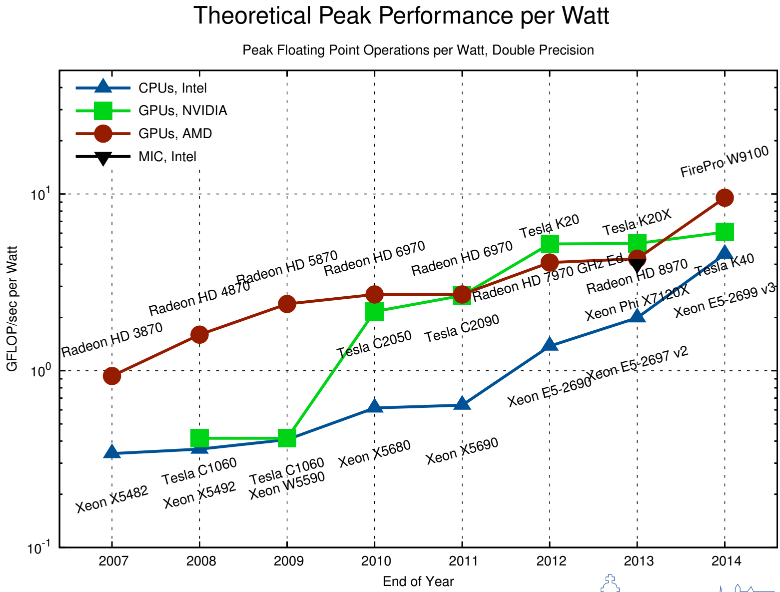
https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Introduction



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

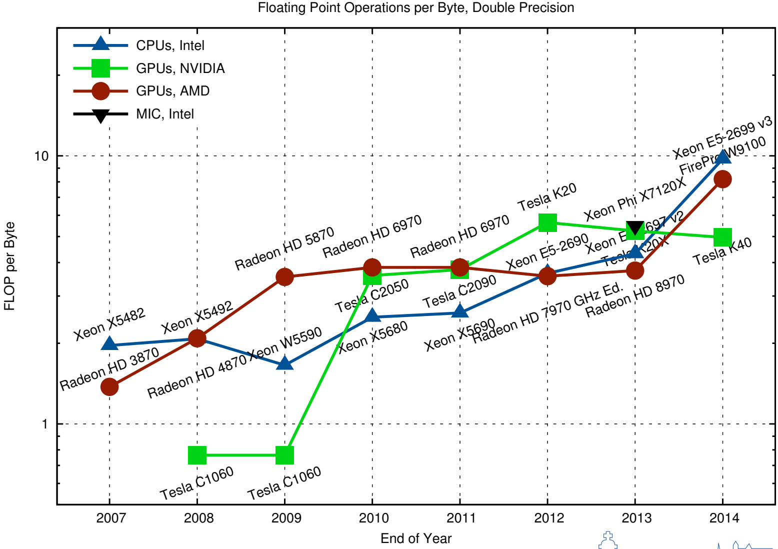
Introduction



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/

Consider Existing CPU Code (Boost.uBLAS)

```
using namespace boost::numeric::ublas;


matrix<double> A(1000, 1000);
vector<double> x(1000), y(1000);

/* Fill A, x, y here */

double val = inner_prod(x, y);
y += 2.0 * x;
A += val * outer_prod(x, y);

x = solve(A, y, upper_tag()); // Upper tri. solver

std::cout << "  2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

High-level code with syntactic sugar

Previous Code Snippet Rewritten with ViennaCL

```
using namespace viennacl;
using namespace viennacl::linalg;

matrix<double> A(1000, 1000);
vector<double> x(1000), y(1000);

/* Fill A, x, y here */

double val = inner_prod(x, y);
y += 2.0 * x;
A += val * outer_prod(x, y);

x = solve(A, y, upper_tag()); // Upper tri. solver

std::cout << "  2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

High-level code with syntactic sugar

ViennaCL in Addition Provides Iterative Solvers

```
using namespace viennacl;
using namespace viennacl::linalg;

compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), y(1000);

/* Fill A, x, y here */

x = solve(A, y, cg_tag());       // Conjugate Gradients
x = solve(A, y, bicgstab_tag()); // BiCGStab solver
x = solve(A, y, gmres_tag());    // GMRES solver
```

No Iterative Solvers Available in Boost.uBLAS...

Thanks to Interface Compatibility

```
using namespace boost::numeric::ublas;
using namespace viennacl::linalg;

compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), y(1000);

/* Fill A, x, y here */

x = solve(A, y, cg_tag());        // Conjugate Gradients
x = solve(A, y, bicgstab_tag()); // BiCGStab solver
x = solve(A, y, gmres_tag());    // GMRES solver
```

Code Reuse Beyond GPU Borders

Armadillo `http://arma.sourceforge.net/`

Eigen `http://eigen.tuxfamily.org/`

MTL 4 `http://www.mtl4.org/`

About ViennaCL

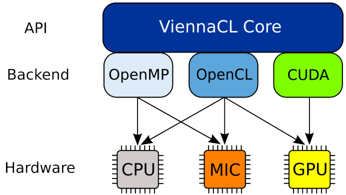
About

High-level linear algebra C++ library

OpenMP, OpenCL, and CUDA backends

Header-only

Multi-platform



Dissemination

Free Open-Source MIT (X11) License

http://viennacl.sourceforge.net/

50-100 downloads per week

Design Rules

Reasonable default values

Compatible to Boost.uBLAS whenever possible

In doubt: clean design over performance

About ViennaCL

Basic Types

- scalar
- vector
- matrix
- compressed_matrix, coordinate_matrix, (sliced_)ell_matrix, hyb_matrix

Data Initialization

Using viennacl::copy()

```
     std::vector<double>      std_x(100);
   ublas::vector<double>    ublas_x(100);
viennacl::vector<double>      vcl_x(100);


for (size_t i=0; i<100; ++i){
    std_x[i] = rand();
  ublas_x[i] = rand();
    vcl_x[i] = rand();  //possible, inefficient
}
```

About ViennaCL

Basic Types

- scalar
- vector
- matrix
- compressed_matrix, coordinate_matrix, (sliced_)ell_matrix, hyb_matrix

Data Initialization

- Using viennacl::copy()

```
     std::vector<double>      std_x(100);
   ublas::vector<double>    ublas_x(100);
viennacl::vector<double>      vcl_x(100);

/* setup of std_x and ublas_x omitted */

viennacl::copy(std_x.begin(), std_x.end(),
               vcl_x.begin());   //to GPU
viennacl::copy(vcl_x.begin(), vcl_x.end(),
               ublas_x.begin()); //to CPU
```

About ViennaCL

Basic Types

scalar

vector

matrix

compressed_matrix, coordinate_matrix, (sliced_)ell_matrix, hyb_matrix

Data Initialization

Using viennacl::copy()

```
    std::vector<std::vector<double> >    std_A;
   ublas::matrix<double>                ublas_A;
viennacl::matrix<double>                  vcl_A;

/* setup of std_A and ublas_A omitted */

viennacl::copy(std_A,
               vcl_A);     // CPU to GPU
viennacl::copy(vcl_A,
               ublas_A);   // GPU to CPU
```

Vector Addition

```
x = y + z;
```

Naive Operator Overloading

```
vector<T> operator+(vector<T> & v, vector<T> & w);
```

$t \leftarrow y + z, x \leftarrow t$

Temporaries are extremely expensive!

Expression Templates

```
vector_expr<vector<T>, op_plus, vector<T> >
operator+(vector<T> & v, vector<T> & w) { ... }

vector::operator=(vector_expr<...> const & e) {
  viennacl::linalg::avbv(*this, 1,e.lhs(), 1,e.rhs());
}
```

Vector Addition

```
// x = y + z
void avbv(...) {
  switch (active_handle_id(x))
  {
    case MAIN_MEMORY:
      host_based::avbv(...);
      break;
    case OPENCL_MEMORY:
      opencl::avbv(...);
      break;
    case CUDA_MEMORY:
      cuda::avbv(...);
      break;
    default:
      raise_error();
  }
}
```

Memory buffers can switch memory domain at runtime

Generalizing Compute Kernels

```
  // x = y + z
  __kernel void avbv(
    double * x,

    double * y,

    double * z, uint size)
{
  i = get_global_id(0);
  for (size_t i=0; i<size; i += get_global_size(0))
    x[i] = y[i] + z[i];

}
```

Generalizing Compute Kernels

```
  // x = a * y + b * z
  __kernel void avbv(
    double * x,
    double a,
    double * y,
    double b,
    double * z, uint size)
{
  i = get_global_id(0);
  for (size_t i=0; i<size; i += get_global_size(0))
    x[i] = a * y[i] + b * z[i];

}
```

Generalizing Compute Kernels

```
  // x[4:8] = a * y[2:6] + b * z[3:7]
  __kernel void avbv(
    double * x, uint off_x,
    double a,
    double * y, uint off_y,
    double b,
    double * z, uint off_z, uint size)
{
  i = get_global_id(0);
  for (size_t i=0; i<size; i += get_global_size(0))
    x[off_x + i] = a * y[off_y + i] + b * z[off_z + i];

}
```

Generalizing Compute Kernels

```
// x[4:2:8] = a * y[2:2:6] + b * z[3:2:7]
__kernel void avbv(
  double * x, uint off_x, uint inc_x,
  double a,
  double * y, uint off_y, uint inc_y,
  double b,
  double * z, uint off_z, uint inc_z, uint size)
{
  i = get_global_id(0);
  for (size_t i=0; i<size; i += get_global_size(0))
    x[off_x + i * inc_x] =  a * y[off_y + i * inc_y]
                          + b * z[off_z + i * inc_z];
}
```

No penalty on GPUs because FLOPs are for free

ViennaCL Features (Excerpt)

Standard Functionality

- BLAS levels 1-3
- Sparse matrix times {vector, dense matrix, sparse matrix}
- Triangular solvers (dense and sparse)

Iterative Solvers

- Krylov solvers: CG, BiCGStab, GMRES
- Preconditioners: Jacobi, serial + parallel ILU0, ILUT, AMG, SPAI

Eigen Solvers

- Lanczos, power iteration
- QR method (experimental), bisection

Miscellaneous

- FFT, QR factorization, SVD, Non-negative matrix factorization
- Bandwidth reduction: Cuthill-McKee, Gibbs-Poole-Stockmeyer

Case Study: Optimizing Iterative Solvers

A Story in Three Parts

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. $r_{i+1} = r_i - \alpha_i A p_i$
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

-

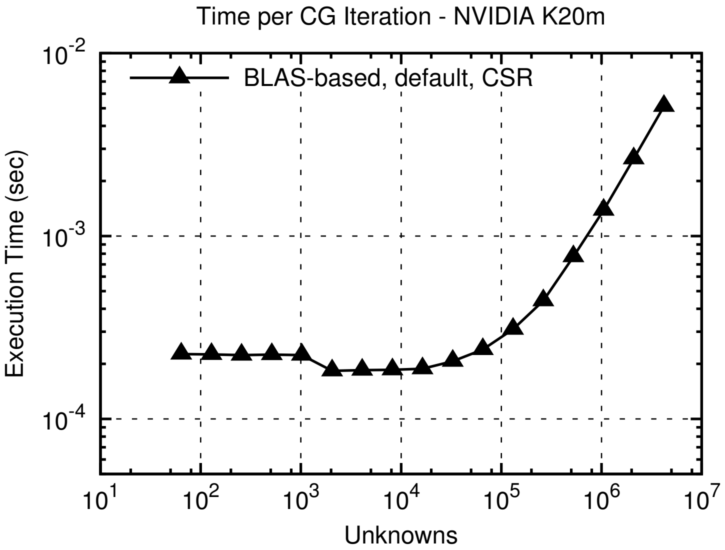
SpMV, AXPY

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

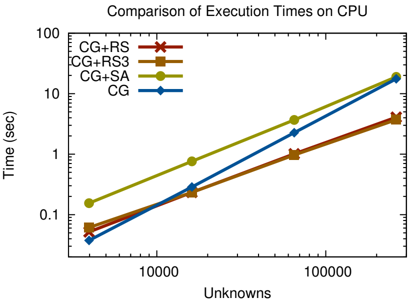


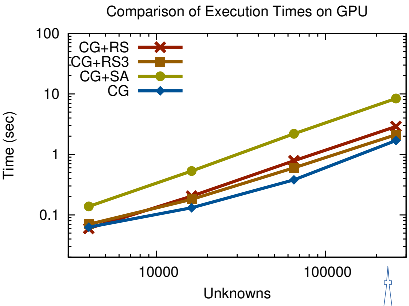
(2D Finite Difference Discretization)

Implications

Kernel launches expensive

Delicate balance for preconditioners





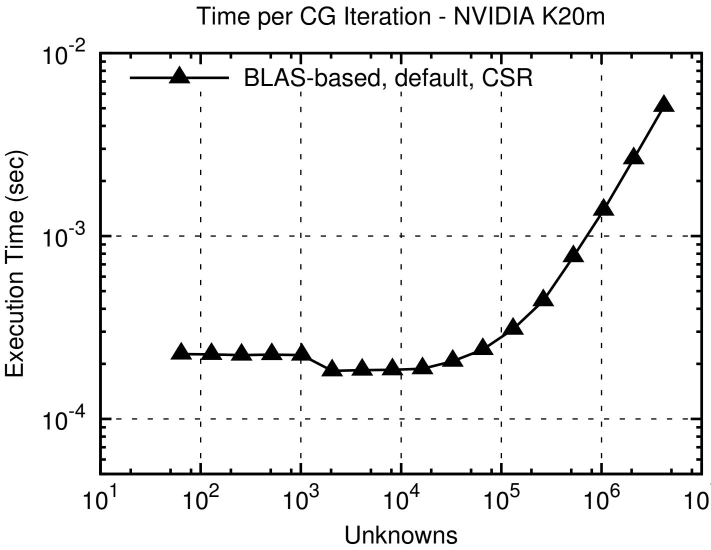
Conjugate Gradient Optimizations

Optimization 1

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

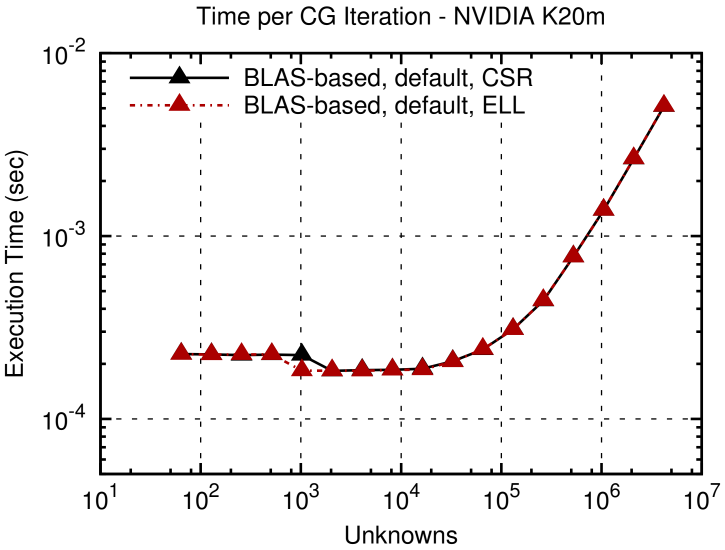
Cf.: N. Bell: Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proc. SC '09*

Conjugate Gradients



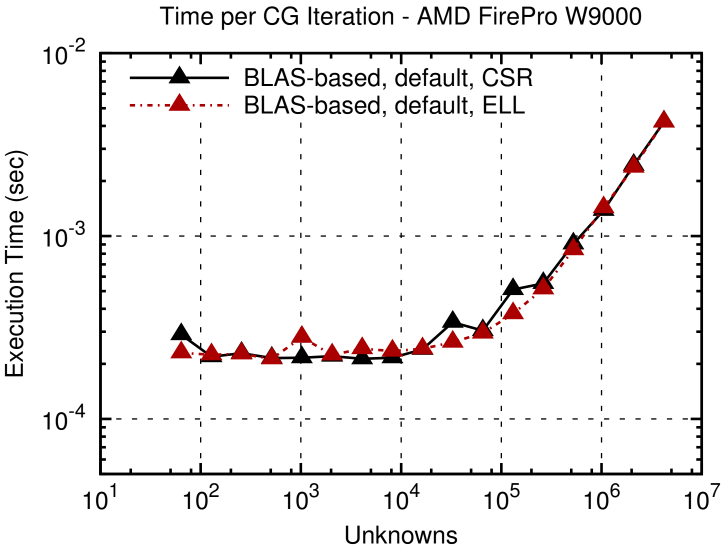
(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradient Optimizations

Optimization 2

Optimize kernel parameters for each operation

Benchmark Setting

Scope for Portability Study

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

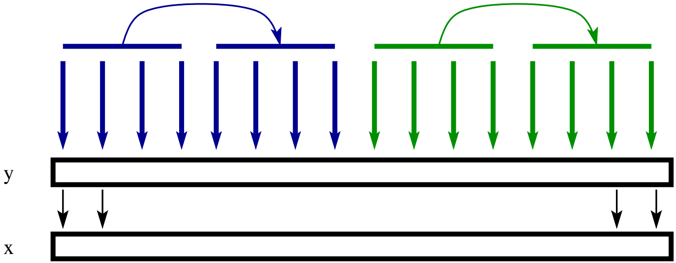
Key Question (Memory-Bandwidth-Limited Kernels)

Good performance of complicated kernels
by optimizing the simplest kernel?

Benchmark Setting

Vector Assignment (Copy) Kernel

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



Parameters (1900 variations)

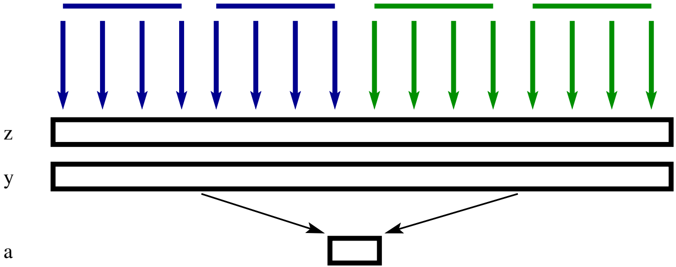
```
for (size_t i = group_start + get_local_id(0);
     i < group_end; i+= get_local_size(0))
  x[i] = y[i];
```

Benchmark Setting

Operations

Vector copy, vector addition, inner product

Matrix-vector product



Devices

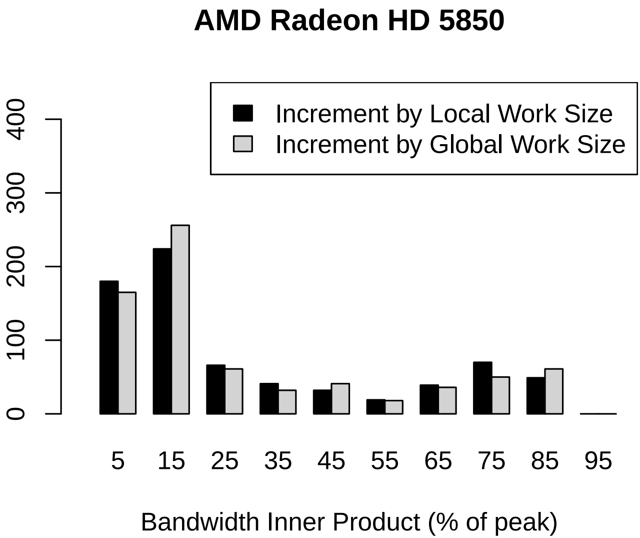
AMD: A10-5800 APU, HD 5850 GPU

INTEL: Dual Socket Xeon E5-2670, Xeon Phi

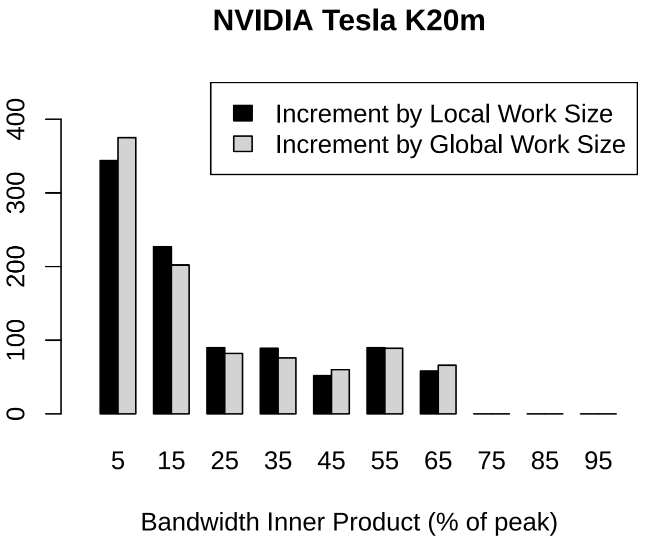
NVIDIA: GTX 285, Tesla K20m

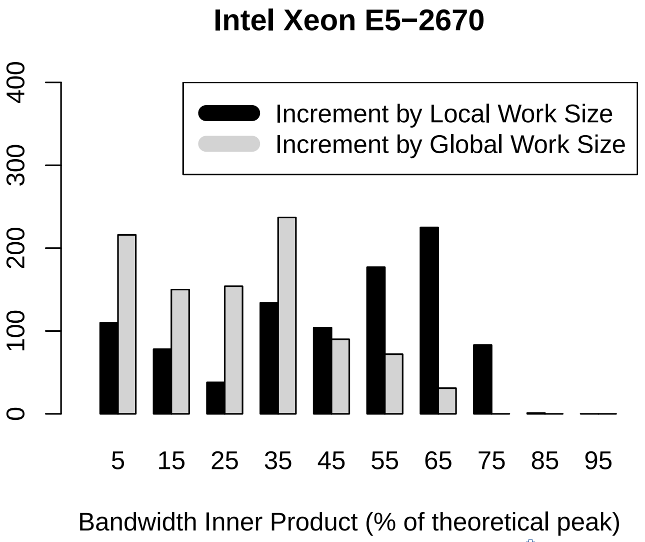
Histograms

Benchmark

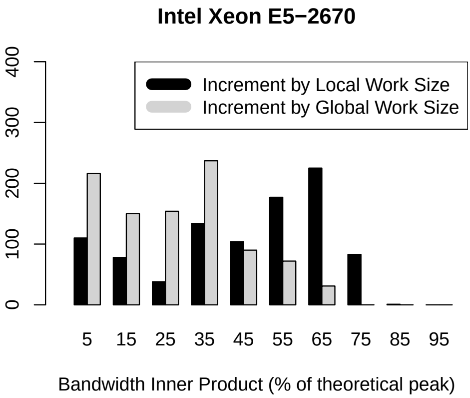


AMD Radeon HD 5850
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)

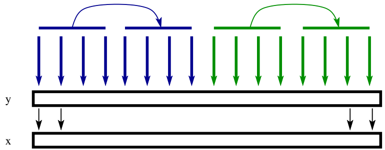

NVIDIA Tesla K20m
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)


Intel Xeon E5−2670
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)

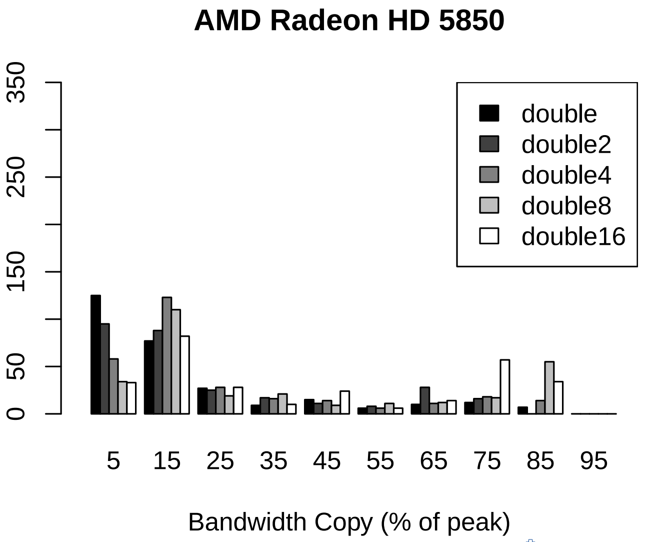
Benchmark

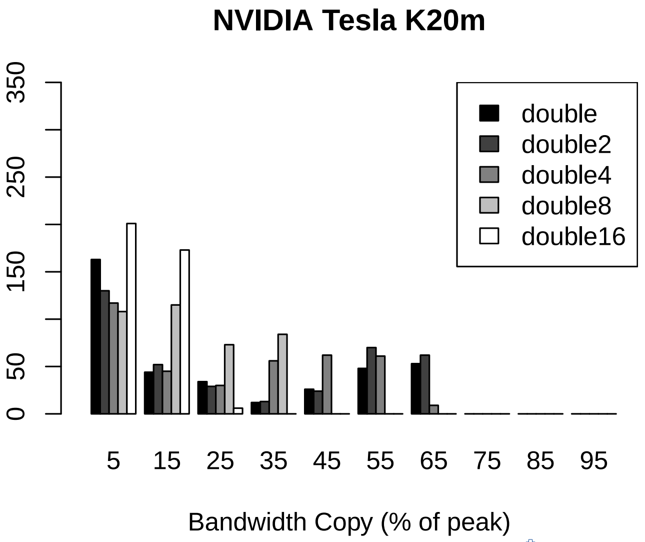


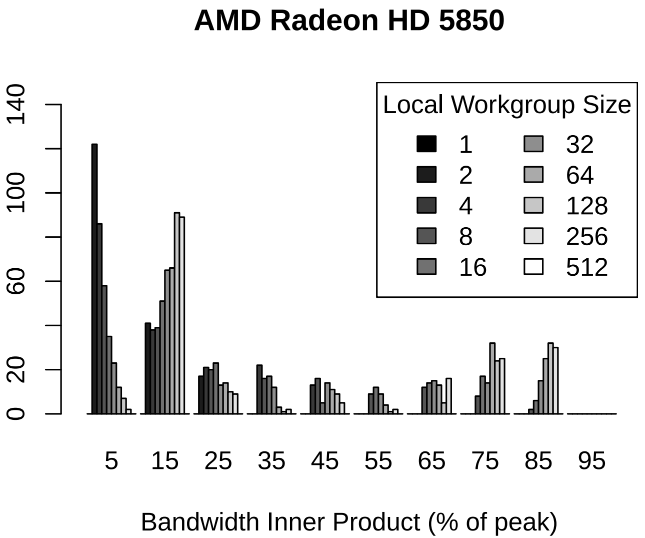
Intel Xeon E5−2670
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of theoretical peak)



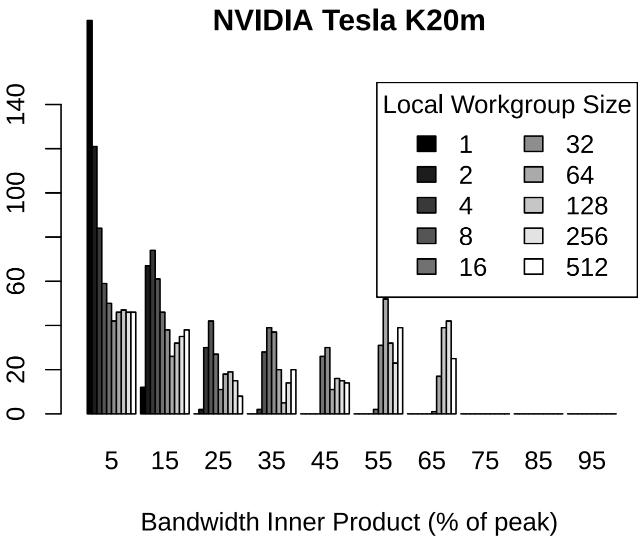
y
x


AMD Radeon HD 5850
350
250
150
50
0
double
double2
double4
double8
double16
5
15
25
35
45
55
65
75
85
95
Bandwidth Copy (% of peak)

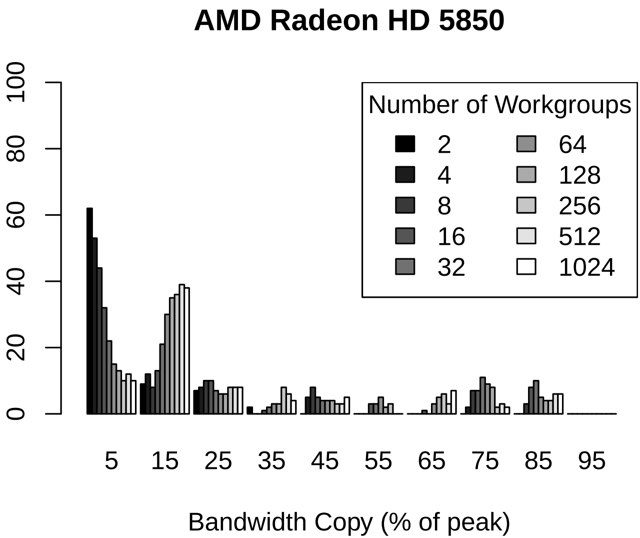

NVIDIA Tesla K20m
0
50
150
250
350
double
double2
double4
double8
double16
5
15
25
35
45
55
65
75
85
95
Bandwidth Copy (% of peak)


AMD Radeon HD 5850
Local Workgroup Size
1
2
4
8
16
32
64
128
256
512
0
20
60
100
140
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)

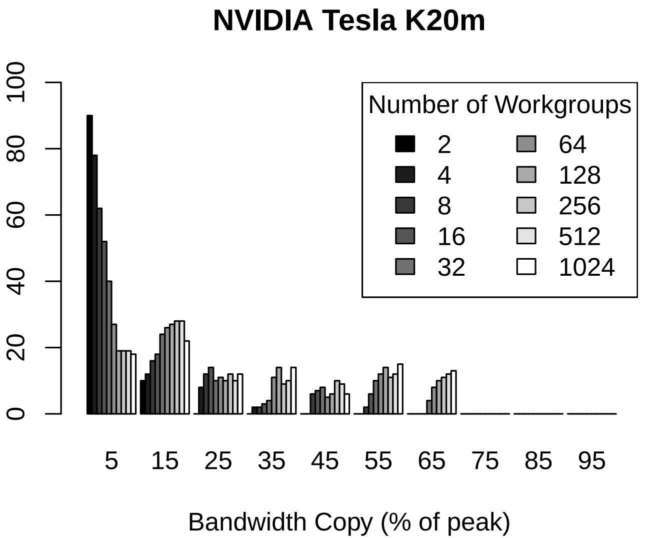
Benchmark



NVIDIA Tesla K20m
Local Workgroup Size
1
2
4
8
16
32
64
128
256
512
0
20
60
100
140
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)


AMD Radeon HD 5850
Number of Workgroups
2
4
8
16
32
64
128
256
512
1024
0
20
40
60
80
100
5
15
25
35
45
55
65
75
85
95
Bandwidth Copy (% of peak)

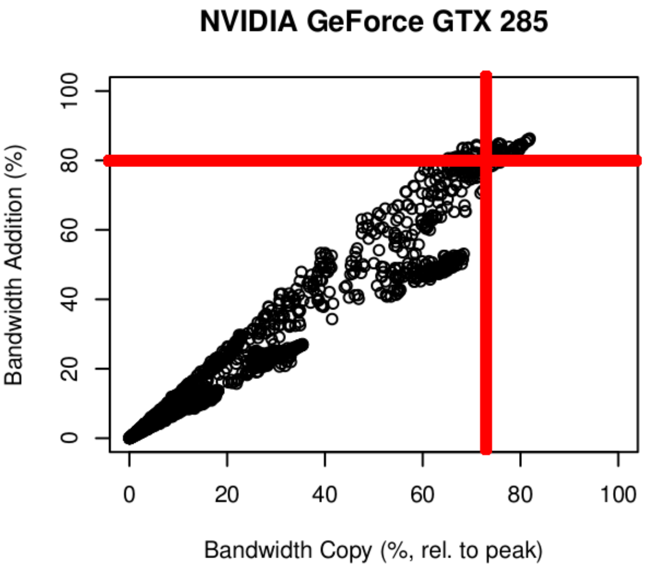
Benchmark


NVIDIA Tesla K20m
100
80
60
40
20
0
Number of Workgroups
2
4
8
16
32
64
128
256
512
1024
5
15
25
35
45
55
65
75
85
95
Bandwidth Copy (% of peak)

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

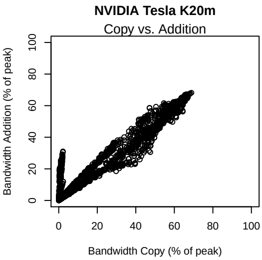
Same Device

Benchmark

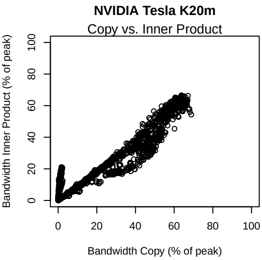

NVIDIA GeForce GTX 285
Bandwidth Addition (%)
0
20
40
60
80
100
0
20
40
60
80
100
Bandwidth Copy (%, rel. to peak)

NVIDIA Tesla K20m

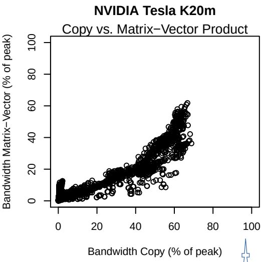
Addition



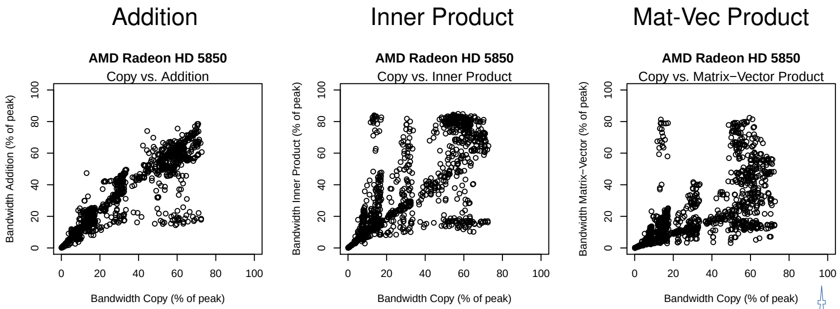
Inner Product



Mat-Vec Product

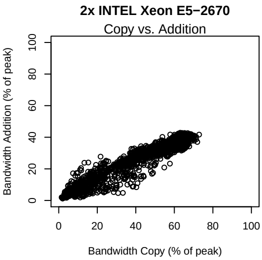


AMD Radeon HD 5850

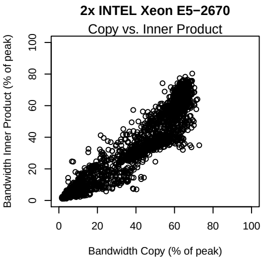


INTEL Dual Xeon E5-2670

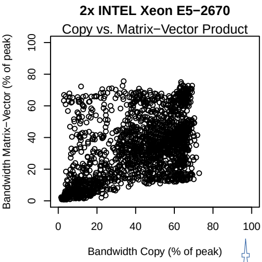
Addition



Inner Product

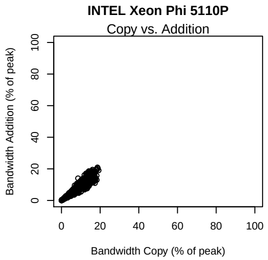


Mat-Vec Product

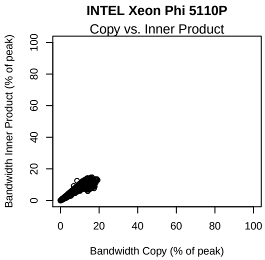


INTEL Xeon Phi

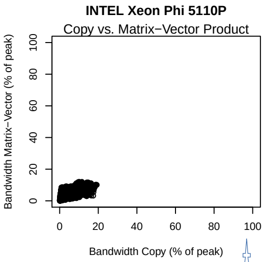
Addition



Inner Product



Mat-Vec Product

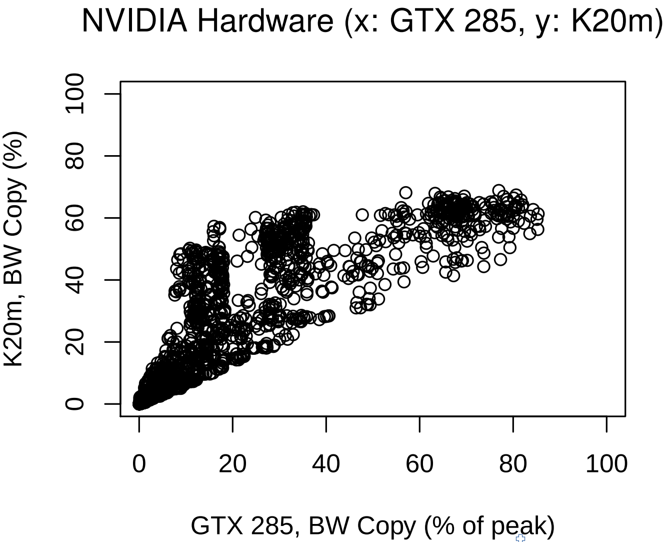


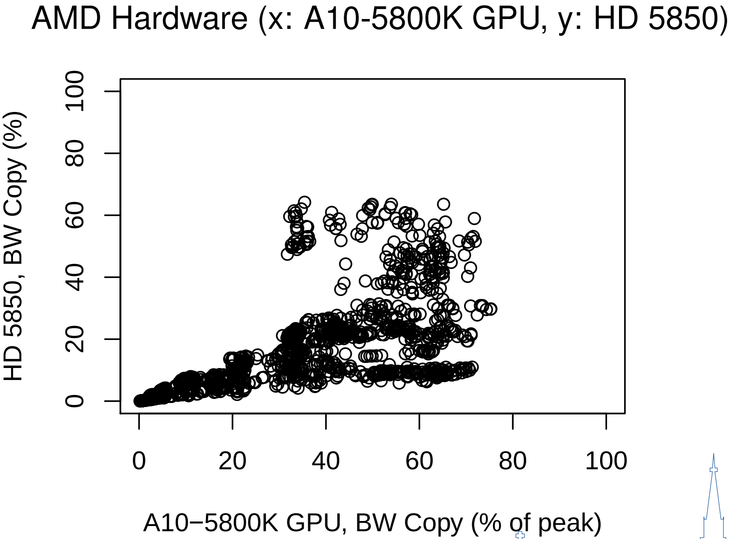
Conclusio:

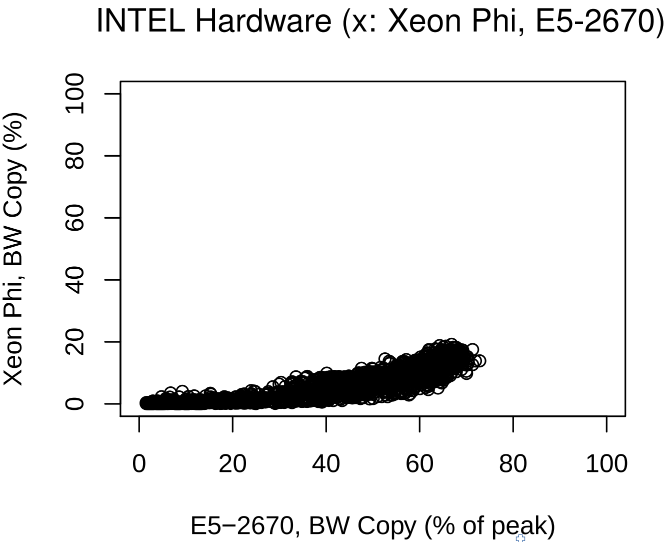
Focus on fastest configurations for copy-kernel sufficient

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

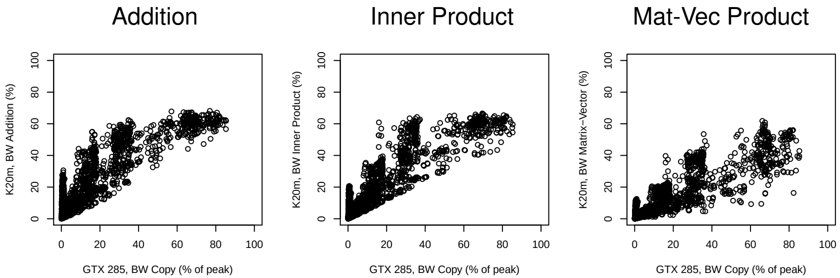
Different Device, Same Vendor


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)

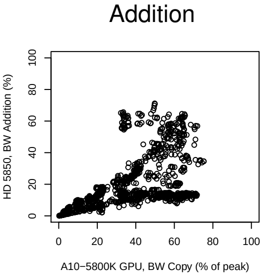

AMD Hardware (x: A10-5800K GPU, y: HD 5850)
HD 5850, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
A10−5800K GPU, BW Copy (% of peak)

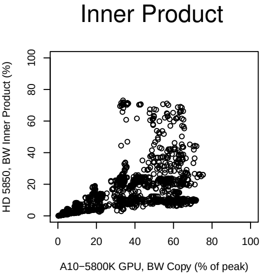

INTEL Hardware (x: Xeon Phi, E5-2670)
Xeon Phi, BW Copy (%)
0
20
40
60
80
100
0
20
40
60
80
100
E5−2670, BW Copy (% of peak)

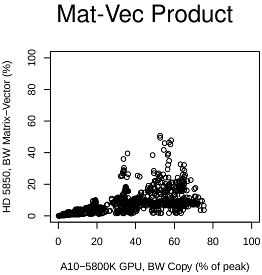
NVIDIA Hardware (x: GTX 285, y: K20m)



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



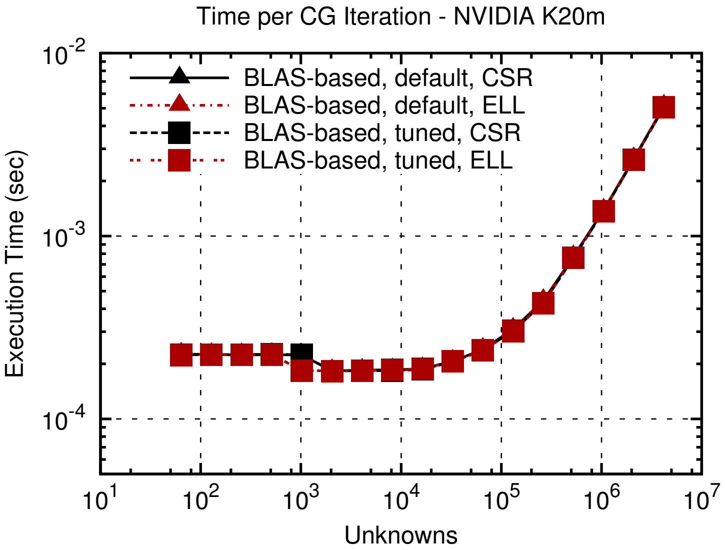




Conclusio:

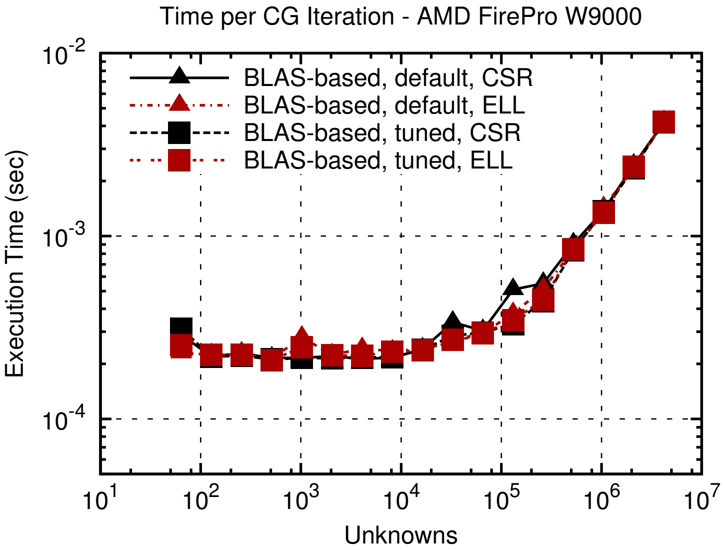
Certain Performance Portability per Vendor

$\rightarrow$ Iterative Solver Optimization



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradient Optimizations

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

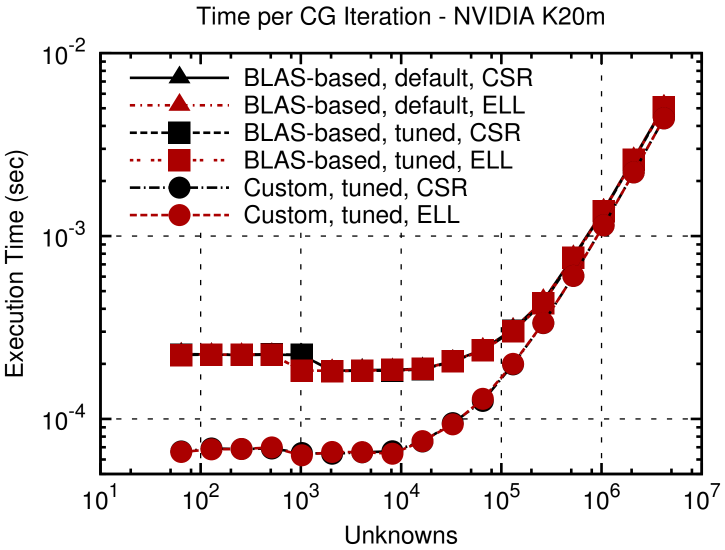
Pipelined CG

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. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

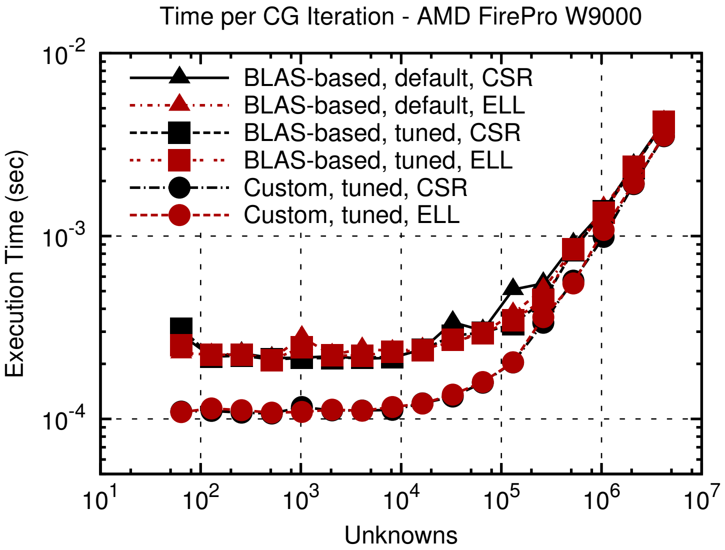
EndFor

Conjugate Gradients



(2D Finite Difference Discretization)

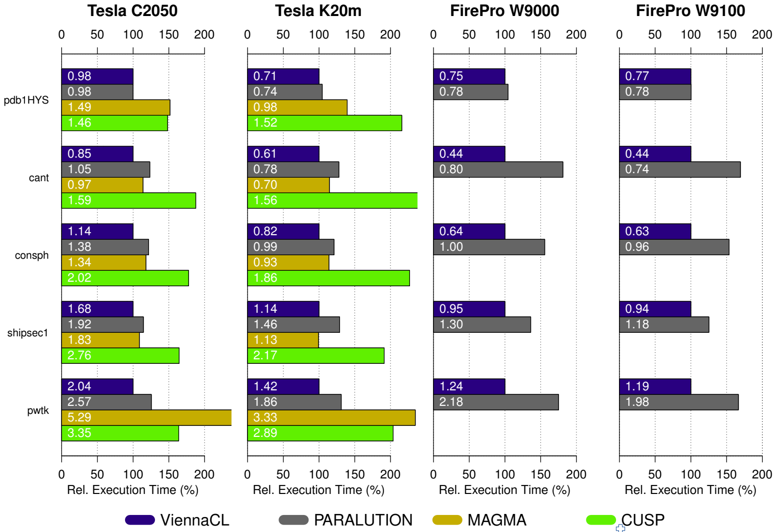
Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients

Benefits of Pipelining also for Large Matrices



Conclusion

Pick Proper Algorithms

- FLOPs are (almost always) for free
- Avoid unnecessary PCI-Express communication
- Expose fine-grained parallelism
- Pipelining and overlapping computations

Fuse Lightweight Kernels

- Reduced number of kernel launches
- Less PCI-Express traffic
- Case study: faster than BLAS-based implementations

Parameterize Kernels for Performance-Portability

- 128 work items, 128 work groups is a good starting point
- Vector datatypes (float4, etc.) often not necessary
- Let each workgroup operate on a contiguous piece of memory