An Introduction to OpenCL for Scientific Computing

Florian Rudolf[1], Karl Rupp[1,2], Josef Weinbub[1]

http://karlrupp.net/

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

Workshop Programming of Heterogeneous Systems in Physics

July 14th, 2014

Introduction

Positions

- PhD student at TU Wien (2009-2011)
- Postdoc at Argonne Natl. Lab. (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
- ...

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

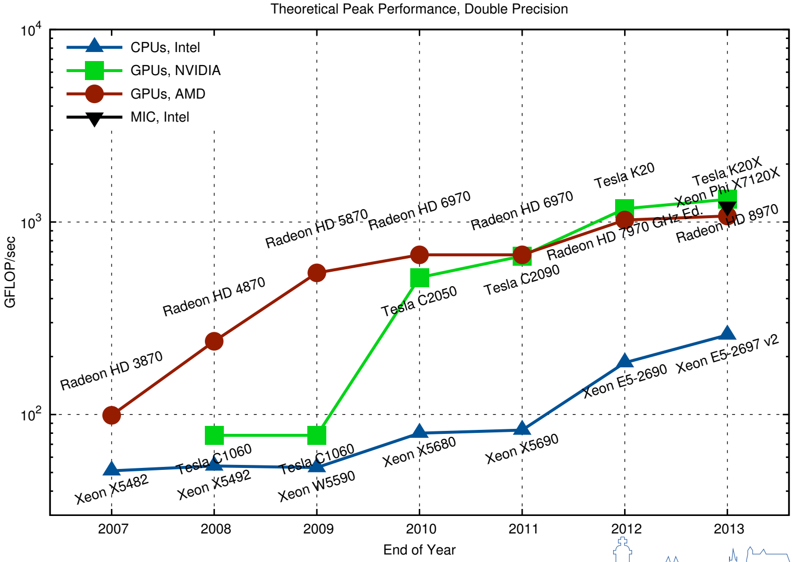
Recent advances in computing have led to an explosion in the amount of data being generated. Processing the ever-growing data in a timely manner has made throughput computing an important aspect for emerging applications. Our analysis of a set of important throughput computing kernels shows that there 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

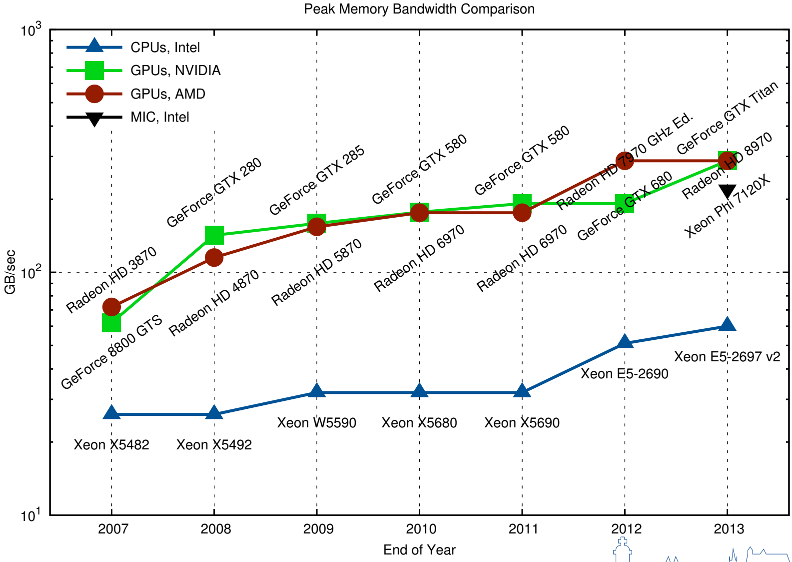
The past decade has seen a huge increase in digital content as more documents are being created in digital form than ever before. Moreover, the web has become the medium of choice for storing and delivering information such as stock market data, personal records, and news. Soon, the amount of digital 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

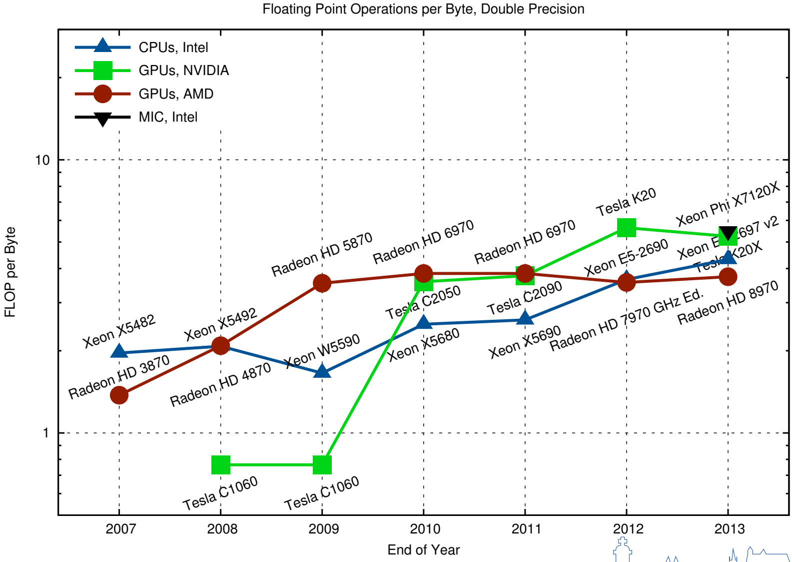
100x Speed-Up?



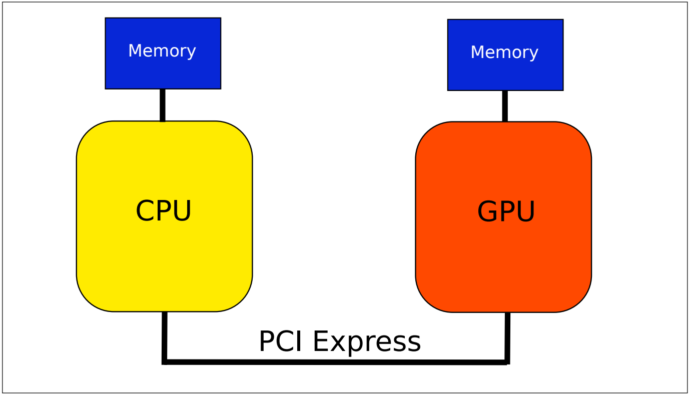
100x Speed-Up?



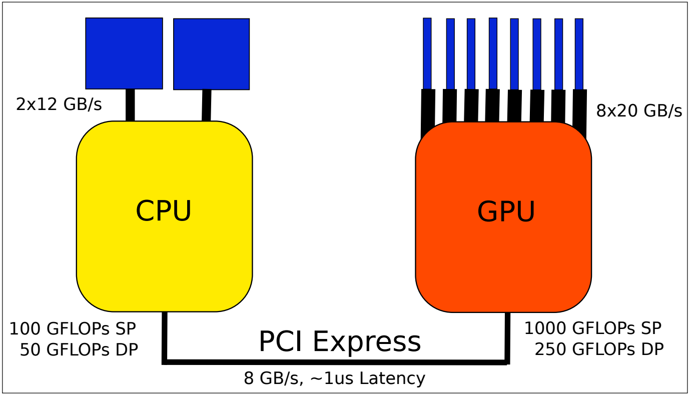
100x Speed-Up?



Computing Architecture Schematic



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

Table of Contents

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.)

Advantages

- Not restricted to a single vendor: Intel, NVIDIA, AMD, ...
- Just a shared C-library
- Does not rely on compiler magic

About OpenCL

Advantages

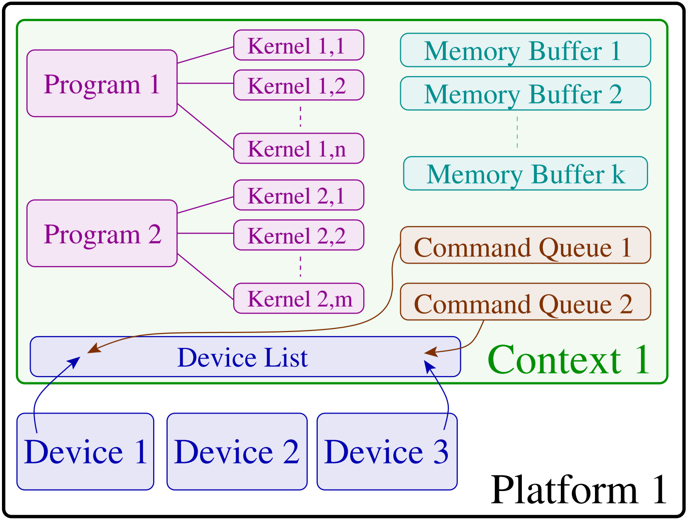
- Not restricted to a single vendor: Intel, NVIDIA, AMD, ...
- Just a shared C-library
- Does not rely on compiler magic

Disadvantages

- Not restricted to a single vendor
- Boilerplate code required
- Portable performance?

Table of Contents

OpenCL Platform Model



OpenCL Platform Model

https://github.com/karlrupp/phsp2014

OpenCL Host API

```c
// Setup contet and queue
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, NULL, 0, NULL);

// Create memory buffers
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries
      , NULL, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      sizeof(float)*2*num_entries, srcA, NULL);

// Create OpenCL program and kernels
program = clCreateProgramWithSource(context, 1, &kernel_src, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

kernel = clCreateKernel(program, "my_kernel", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL);

// Run an OpenCL kernel:
global_work_size[0] = 128;
 local_work_size[0] = 64;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size,
      0, NULL, NULL);
```

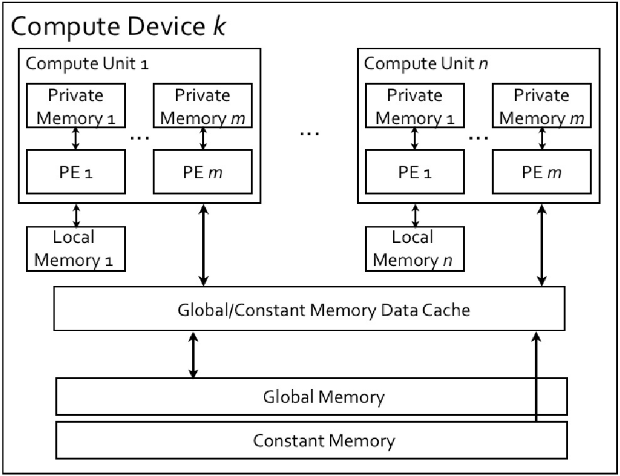
Issues

“Where is the error?”

Manage OpenCL handles

Table of Contents

Primer: OpenCL Device Execution Model



http://developer.amd.com/documentation/articles/PublishingImages/opencl_figure5.jpg

Sample Operation: Inplace Vector Addition

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} += \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

OpenCL Kernel

```
__kernel void inplace_add(
          __global const float * vec1,
          __global const float * vec2,
          unsigned int size)
{
  for (unsigned int i  = get_global_id(0);
                    i  < size;
                    i += get_global_size(0))
    x[i] += y[i];
}
```

OpenCL Kernel Language

Just Like C

Datatypes: **`float`**, **`double`**, **`long`**, ... , `float2`, `float4`, ...

Keywords: **`for`**, **`if`**, **`switch`**, ...

...

Thread Management

ID: `get_local_id(dim)`, `get_global_id(dim)`

Count: `get_local_size(dim)`, `get_global_size(dim)`

Sync: `barrier(flags)`, `mem_fence(flags)`

Memory Qualifiers

`__global`: Global memory (e.g. GPU RAM)

`__constant`: Constant global memory (vendor-specific!)

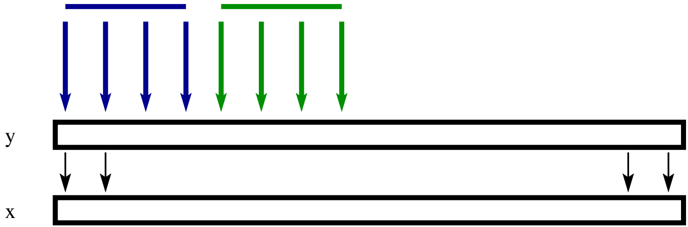
`__local`: Shared memory (per workgroup)

Vector Assignment (Copy) Kernel

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

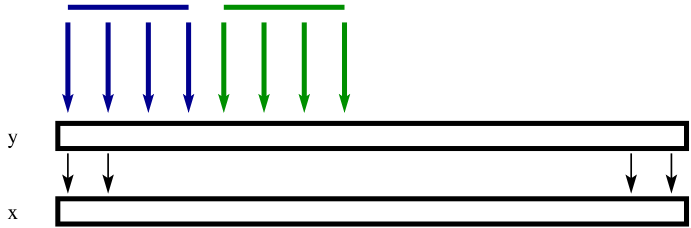
Vector Assignment (Copy) Kernel

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



Vector Assignment (Copy) Kernel

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

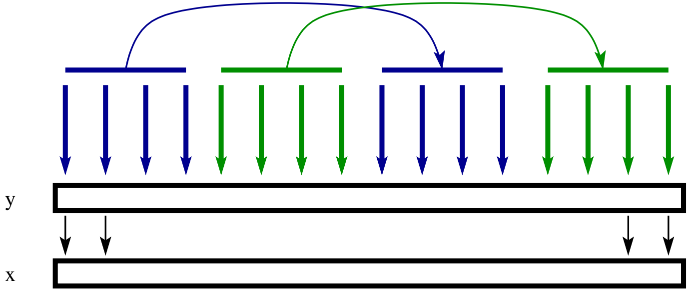


Parameters ($\gg 1000$ variations possible)

- Local work size, global work size
- Vector types (float1, float2, ... , float16)
- Thread increment type

Vector Assignment (Copy) Kernel

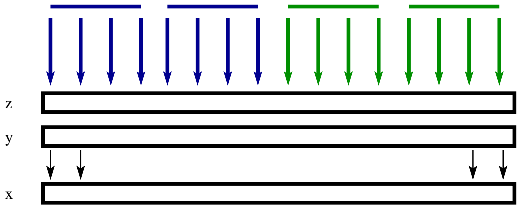
$x \Leftarrow y$ for (large) vectors x, y



Parameters ($\gg 1000$ variations possible)

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

No Synchronization: Vector Addition



Synchronization: Dot Product

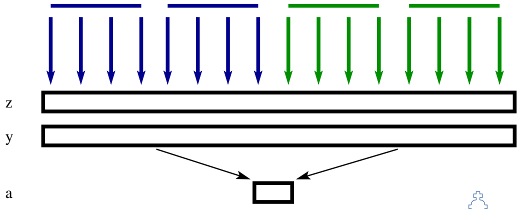


Table of Contents

Comparison with CUDA and OpenACC

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);
  ...
}
```

Almost no additional code required

Vendor-lock

Relies on `nvcc` being available (plus version conflicts...)

Comparison with CUDA and OpenACC

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);
}
```

Additional boilerplate code required (low-level API)

Broad hardware support (separate SDKs)

Second-best programming model for each vendor

Comparison with CUDA and OpenACC

OpenACC

```
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);
}
```

- Simple OpenMP-type pragma annotations
- Compiler support?
- Insufficient control over memory transfers?

Thread Explosion Problem

```
void func(double *A) {
  #pragma omp parallel for
  for(int i=0; i<1337; i++) A[i] = 42.0;
}

void func2(double *A) {
  #pragma omp parallel for
  for(int i=0; i<1337; i++) func(A);
}

int main() {
  double A[1337];
  func(A);  // okay
  func2(A); // boom...
}
```

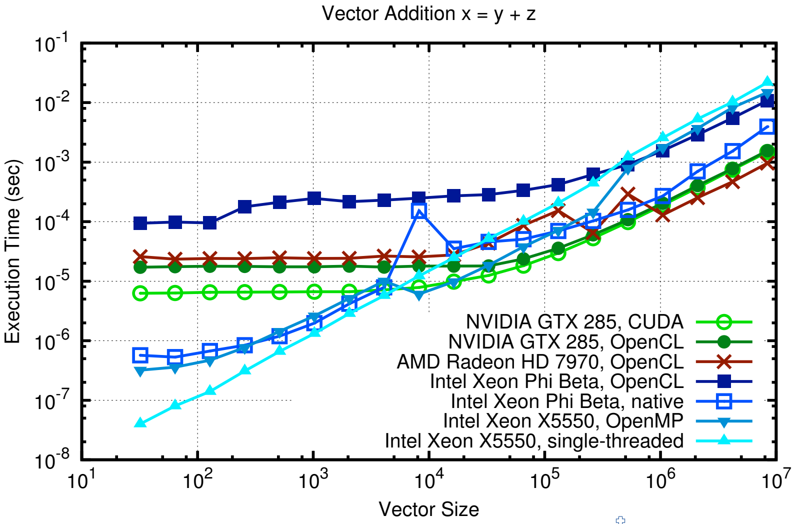
Usually not as obvious as here

Problem when OpenMP'ing MPI code

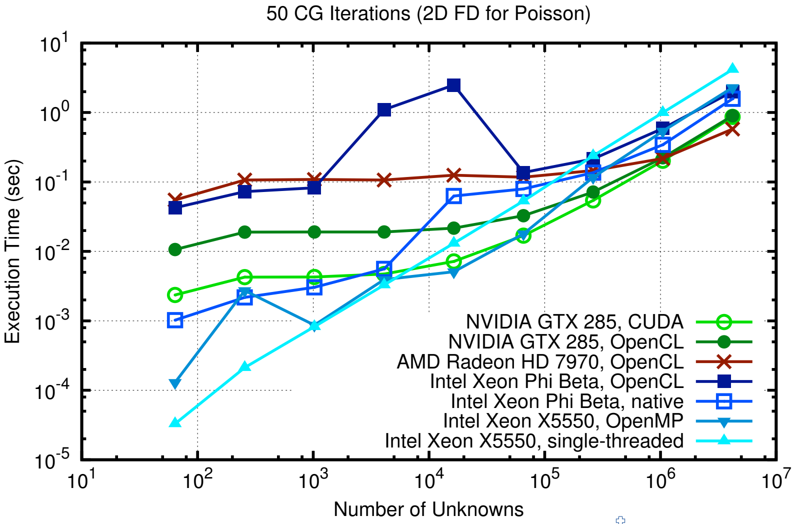
Headache for library implementors

Table of Contents

Benchmarks


Vector Addition x = y + z
Execution Time (sec)
Vector Size
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
NVIDIA GTX 285, CUDA
NVIDIA GTX 285, OpenCL
AMD Radeon HD 7970, OpenCL
Intel Xeon Phi Beta, OpenCL
Intel Xeon Phi Beta, native
Intel Xeon X5550, OpenMP
Intel Xeon X5550, single-threaded

Benchmarks



50 CG Iterations (2D FD for Poisson)
Execution Time (sec)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
Number of Unknowns
NVIDIA GTX 285, CUDA
NVIDIA GTX 285, OpenCL
AMD Radeon HD 7970, OpenCL
Intel Xeon Phi Beta, OpenCL
Intel Xeon Phi Beta, native
Intel Xeon X5550, OpenMP
Intel Xeon X5550, single-threaded

Portable Performance

Scope for Portability Study

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

Key Question (Memory-Bandwidth-Limited Kernels)

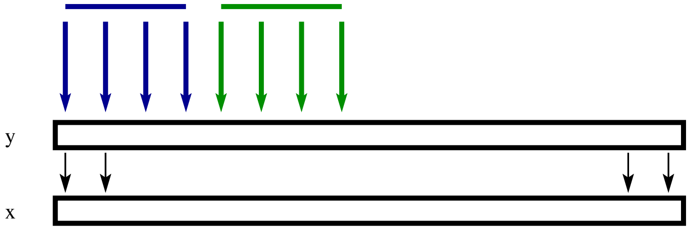
Good performance of complicated kernels
by optimizing the simplest kernel?

Vector Assignment (Copy) Kernel

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

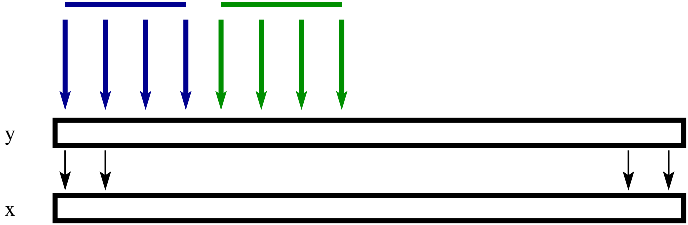
Vector Assignment (Copy) Kernel

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



Vector Assignment (Copy) Kernel

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



Parameters (1900 variations)

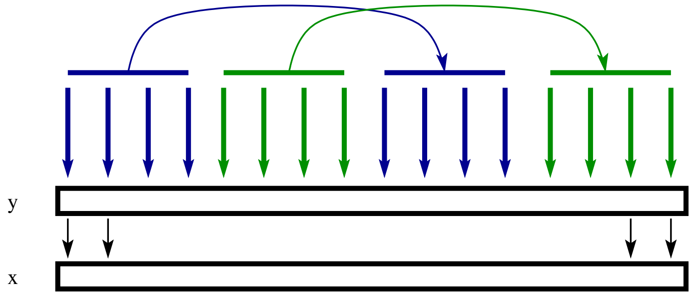
Local work size, global work size

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

Thread increment type

Vector Assignment (Copy) Kernel

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

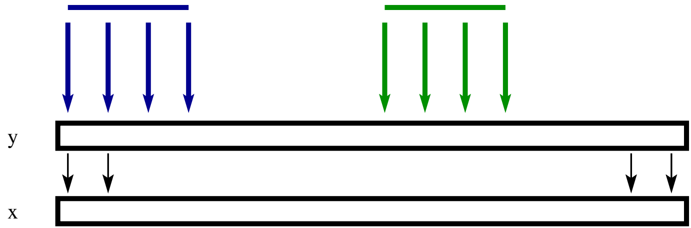


Parameters (1900 variations)

```
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

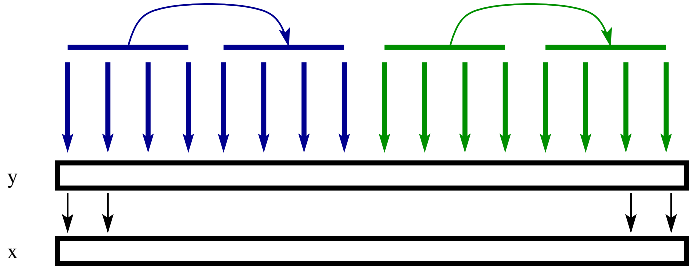


Parameters (1900 variations)

```
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



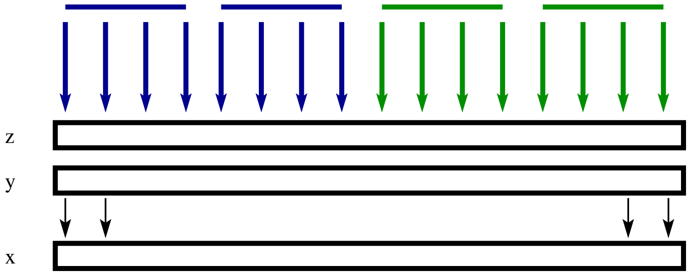
Parameters (1900 variations)

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

Operations

Vector copy, vector addition, inner product

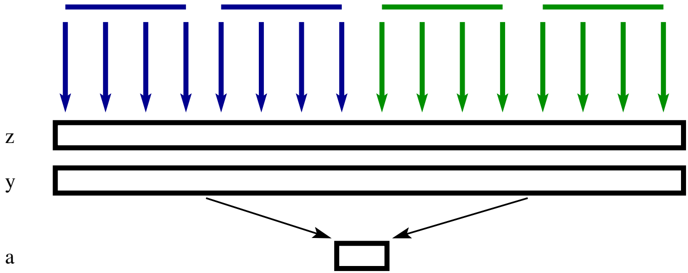
Matrix-vector product



Operations

Vector copy, vector addition, inner product

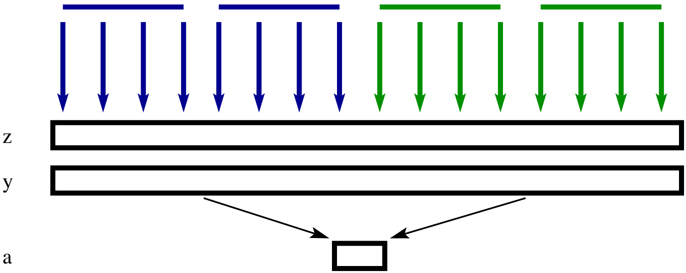
Matrix-vector product



Operations

Vector copy, vector addition, inner product

Matrix-vector product



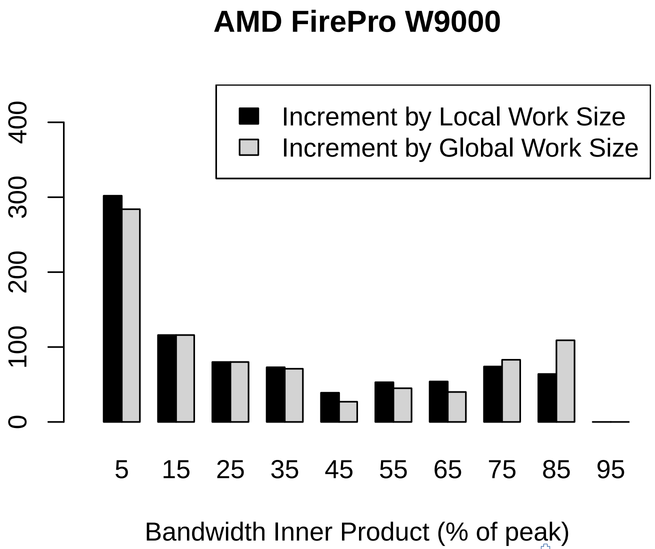
Devices

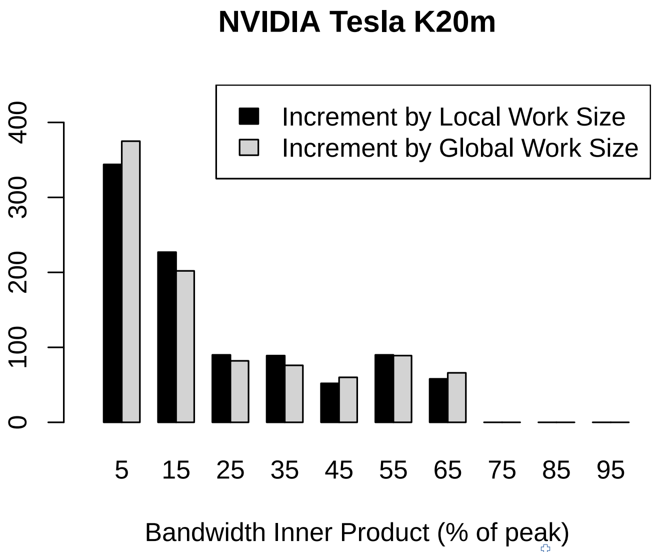
AMD: Radeon HD 5850, FirePro W9000

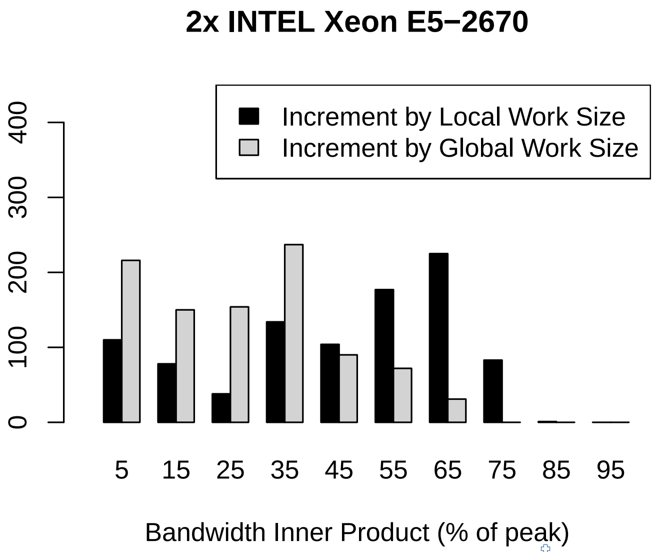
INTEL: Dual Socket Xeon E5-2670, Xeon Phi

NVIDIA: GTX 285, Tesla K20m

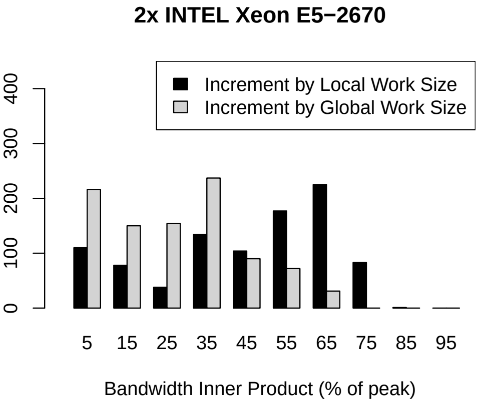
Histograms

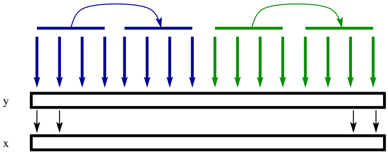

AMD FirePro W9000
Increment by Local Work Size
Increment by Global Work Size
0
100
200
300
400
5
15
25
35
45
55
65
75
85
95
Bandwidth Inner Product (% of peak)


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)


2x 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 peak)

Portable Performance

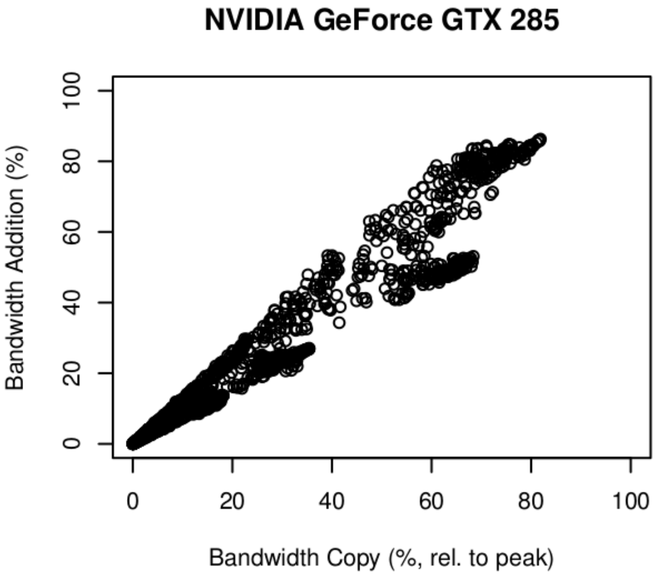




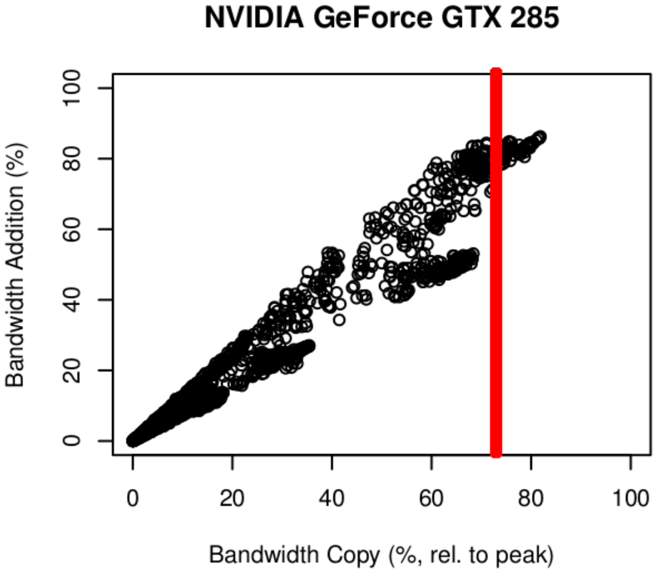
[Addition|Inner Product|Matrix-Vector] vs. Copy Kernel

Same Device

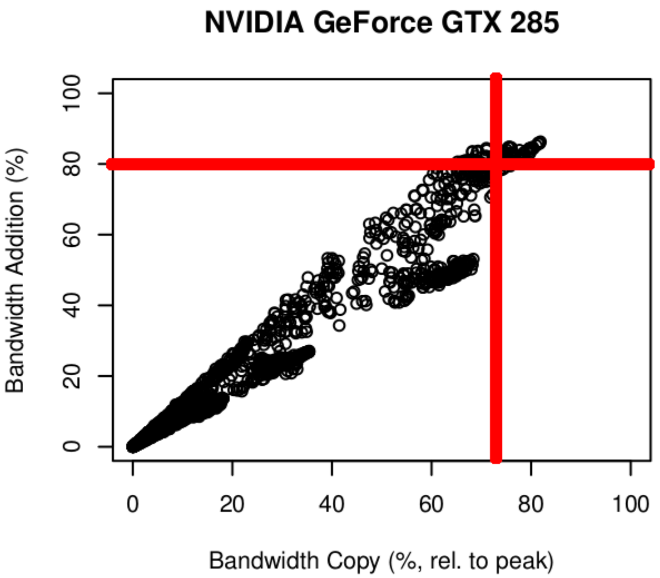
Portable Performance



Portable Performance

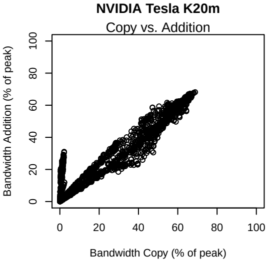


Portable Performance

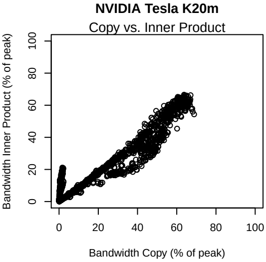


NVIDIA Tesla K20m

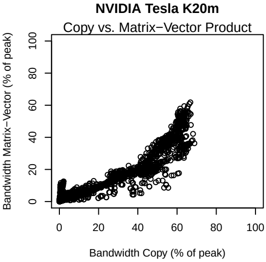
Addition



Inner Product

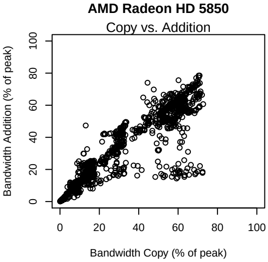


Mat-Vec Product

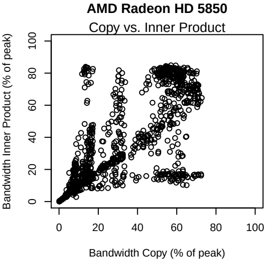


AMD Radeon HD 5850

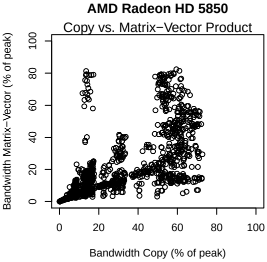
Addition



Inner Product

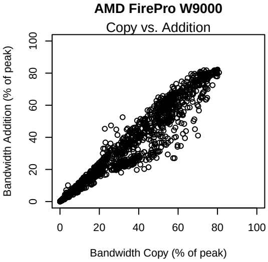


Mat-Vec Product

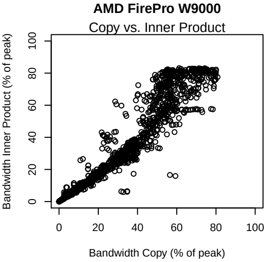


AMD FirePro W9000

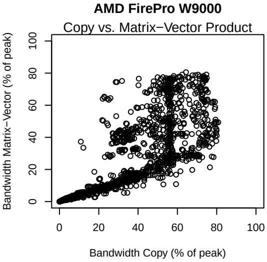
Addition



Inner Product



Mat-Vec Product

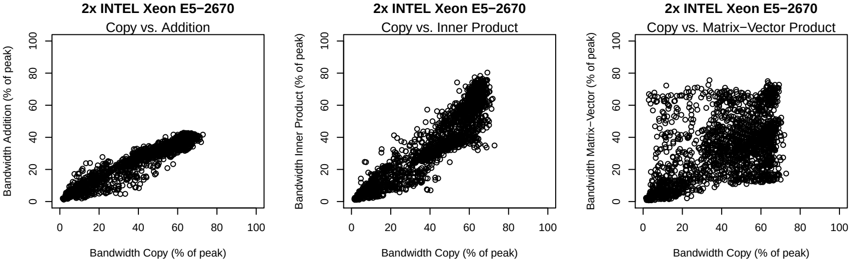


INTEL Dual Xeon E5-2670

Addition

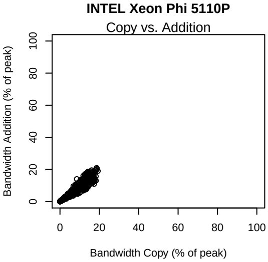
Inner Product

Mat-Vec Product

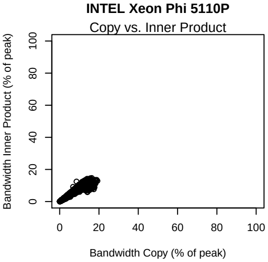


INTEL Xeon Phi

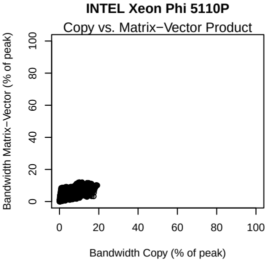
Addition



Inner Product



Mat-Vec Product



Portable Performance

Conclusio:

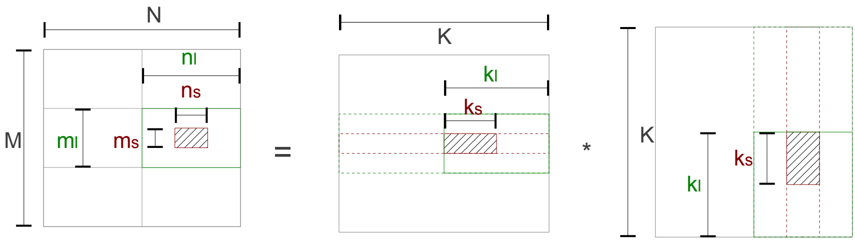
Focus on fastest configurations for copy-kernel sufficient

Good choice:

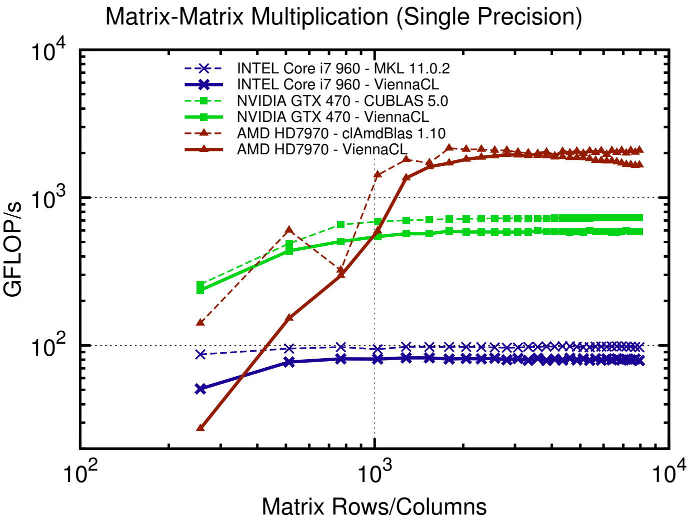
Local workgroup size of 128 with 128 workgroups

Matrix-Matrix Multiplication

- Compute-bound
- Block-decomposition to maximize cache utilization



Portable Performance



Ph. Tillet *et al.*: HotPar'13

Summary

OpenCL

Program CPUs and accelerators (GPUs, MIC, etc.) from different vendors

Clean integration into existing projects

Not a silver bullet

Recommendations

Program with data locality and data movement in mind

Compilers cannot fix bad programming

There is no free lunch

Convenient Use of OpenCL

Use libraries built on top of OpenCL

Tue, July 15, 9:00am:

Lessons Learned in Developing the Linear Algebra Library ViennaCL