EPGPU: Expressive Programming for GPGPU

Dr. Orion Sky Lawlor

lawlor@alaska.edu

U. Alaska Fairbanks

2011-05-31

http://www.cs.uaf.edu/sw/EPGPU

Obligatory Introductory Quote

**"He who controls the past,
controls the future."**

George Orwell, *1984*

In Parallel Programming…

**"He who controls the writes,
controls performance."**

~~**"He who controls the past,
controls the future."**~~

~~**George Orwell, *1984***~~

Orion Lawlor, 2011

Talk Outline

- Embedding OpenCL inside C++
- FILL kernels and parallelism
  - Who controls the writes?
- Application Performance
- Conclusions

EPGPU:
Embedding OpenCL in C++

Why Bother?

- Parallel hardware is here
  - Needs good parallel software
- Why OpenCL?
  - Just as fast as CUDA on GPU
  - Same *binary* works on ATI, NVIDIA, x86 SMP, cellphone, …
- Why C++?
  - Similar syntax with OpenCL
  - Macros, templates, operators, …

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);

// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                        ...                 }

// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                                NULL, &err);
CL_CHECK_ERROR(err);

float f=1.2345;
err = clEnqueueWriteBuffer(theQueue, devP, CL_TRUE,
                                0, sizeof(float), &f, 0, NULL, NULL);
CL_CHECK_ERROR(err);


// Create kernel
theKrnl = clCreateKernel(theProg, "writeArr", &err);
CL_CHECK_ERROR(err);

// Call the kernel
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devP);
CL_CHECK_ERROR(err);

float addThis=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(float), &addThis);
CL_CHECK_ERROR(err);

size_t localsz = 32;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1, NULL,
                                &globalsz, &localsz, 0,
                                NULL, NULL);
CL_CHECK_ERROR(err);

// Read back the results
for (int i=0;i<n;i+=4) {
                err = clEnqueueReadBuffer(theQueue, devP, CL_TRUE,
            i*sizeof(float), sizeof(float), &f, 0, NULL, NULL);
                CL_CHECK_ERROR(err);

                std::cout<<"arr["<<i<<"]="<<f<<"\n";
}

// Cleanup
clReleaseMemObject(devP);
clReleaseKernel(theKrnl);
clReleaseProgram(theProg);
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);
```

Auto-init on first use

```
// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                        ...                 }

// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                                NULL, &err);
CL_CHECK_ERROR(err);

float f=1.2345;
err = clEnqueueWriteBuffer(theQueue, devP, CL_TRUE,
                                0, sizeof(float), &f, 0, NULL, NULL);
CL_CHECK_ERROR(err);


// Create kernel
theKrnl = clCreateKernel(theProg, "writeArr", &err);
CL_CHECK_ERROR(err);

// Call the kernel
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devP);
CL_CHECK_ERROR(err);

float addThis=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(float), &addThis);
CL_CHECK_ERROR(err);

size_t localsz = 32;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1, NULL,
                                &globalsz, &localsz, 0,
                                NULL, NULL);
CL_CHECK_ERROR(err);

// Read back the results
for (int i=0;i<n;i+=4) {
                err = clEnqueueReadBuffer(theQueue, devP, CL_TRUE,
        i*sizeof(float), sizeof(float), &f, 0, NULL, NULL);
                CL_CHECK_ERROR(err);

                std::cout<<"arr["<<i<<"]="<<f<<"\n";
}

// Cleanup
clReleaseMemObject(devP);
clReleaseKernel(theKrnl);
clReleaseProgram(theProg);
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);
```

```
// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                    ...                }
```

```
// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                                NULL, &err);
CL_CHECK_ERROR(err);
```

GPU Buffer Allocation:
- Big performance hit
- Automagic buffer reuse (>2x faster)

```
size_t localsz = 32;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1, NULL,
                                &globalsz, &localsz, 0,
                                NULL, NULL);
CL_CHECK_ERROR(err);

// Read back the results
for (int i=0;i<n;i+=4) {
                err = clEnqueueReadBuffer(theQueue, devP, CL_TRUE,
            i*sizeof(float), sizeof(float), &f, 0, NULL, NULL);
                CL_CHECK_ERROR(err);

                std::cout<<"arr["<<i<<"]="<<f<<"\n";
}

// Cleanup
clReleaseMemObject(devP);
clReleaseKernel(theKrnl);
clReleaseProgram(theProg);
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);

// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                    ...                 }

// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                                NULL, &err);
CL_CHECK_ERROR(err);

float f=1.2345;
err = clEnqueueWriteBuffer(theQueue, devP, CL_TRUE,
                                0, sizeof(float), &f, 0, NULL, NULL);
CL_CHECK_ERROR(err);


// Create kernel
theKrnl = clCreateKernel(theProg, "writeArr", &err);
CL_CHECK_ERROR(err);

// Call the kernel
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devP);
CL_CHECK_ERROR(err);

float addThis=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(float), &addThis);
CL_CHECK_ERROR(err);

size_t localsz = 32;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1, NULL,
                                &globalsz, &localsz, 0,
                                NULL, NULL);
CL_CHECK_ERROR(err);

// Read b
for (int i=0
        i

}

// Cleanu
clRelease
clRelease
clRelease
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

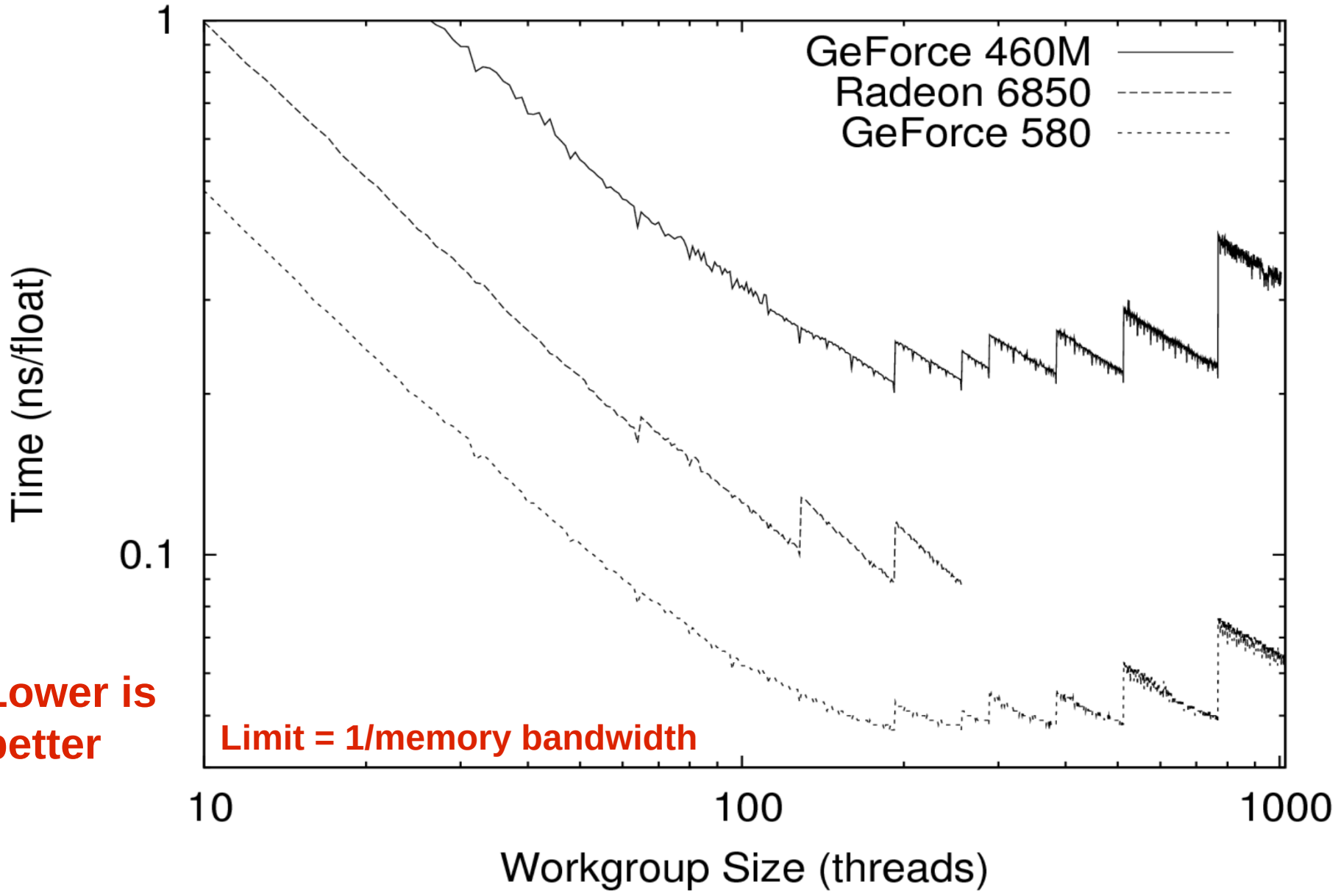
Workgroup Size:
- **Many constraints**
- **Performance critical**

Workgroup Size Determination

```
size_t localsz = 256;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1,
NULL, &globalsz, &localsz, 0, NULL, NULL);
```

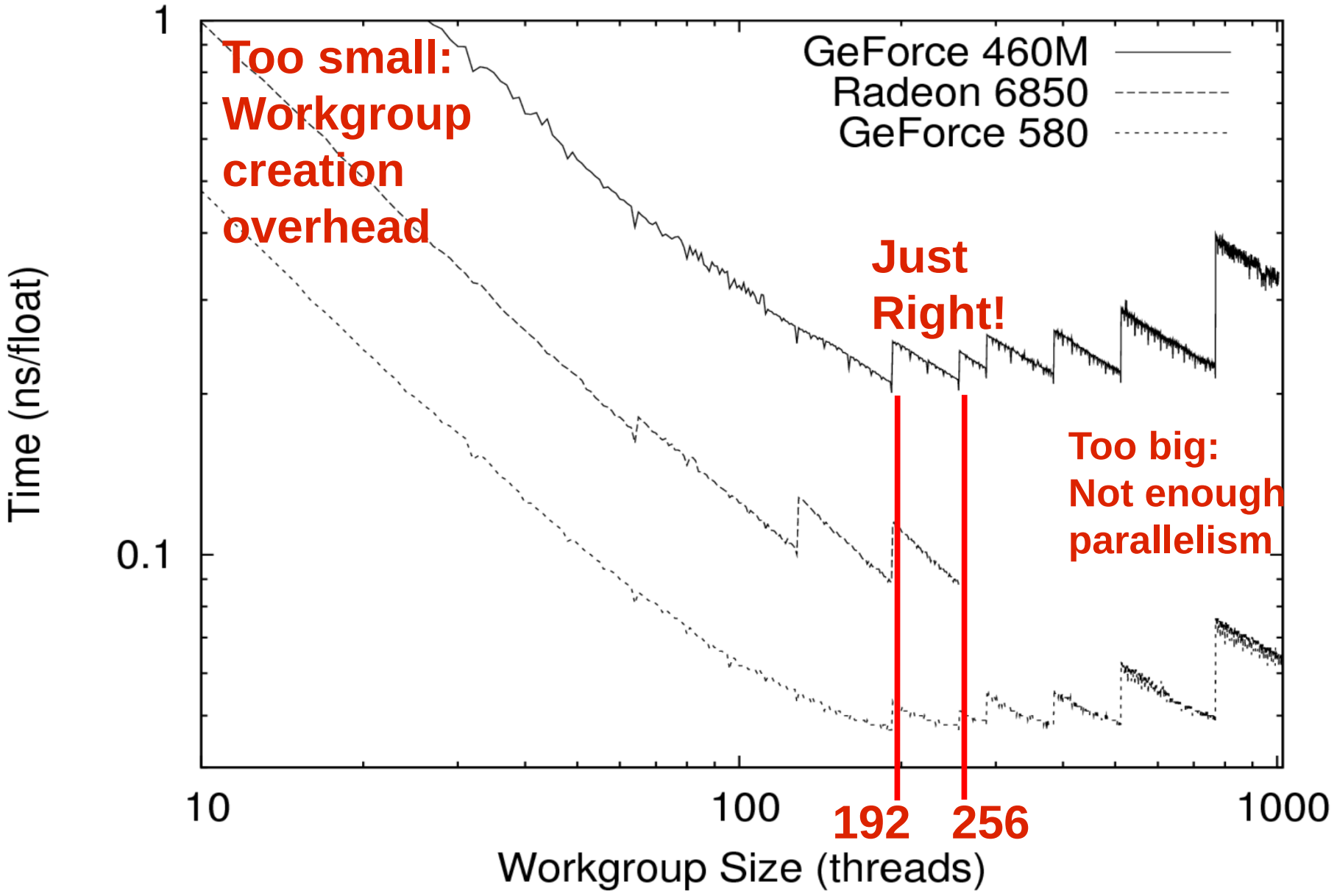
- **Workgroup size MUST be less than CL_KERNEL_WORK_GROUP_SIZE and CL_DEVICE_MAX_WORK_GROUP_SIZE**
  - **Yet still be big enough (performance)**
  - **And be a multiple of global size (err)**
- **In theory: constrained autotuner**
- **In practice: hardcoded constant**

Workgroup Size vs Time



Lower is better

Workgroup Size vs Time



Solution: Remove Constraints

- Generate OpenCL code "if (i<n) …"
  - Adds one highly coherent branch
  - Automatically added to your kernel
- Round up global size to be a multiple of an efficient workgroup size
  - Obey hardware constraints
  - Correct answer for any global size

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);

// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                        ...                 }

// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                                NULL, &err);
CL_CHECK_ERROR(err);

float f=1.2345;
err = clEnqueueWriteBuffer(theQueue, devP, CL_TRUE,
                                0, sizeof(float), &f, 0, NULL, NULL);
CL_CHECK_ERROR(err);


// Create kernel
theKrnl = clCreateKernel(theProg, "writeArr", &err);
CL_CHECK_ERROR(err);

// Call the kernel
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devP);
CL_CHECK_ERROR(err);

float addThis=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(float), &addThis);
CL_CHECK_ERROR(err);

size_t localsz = 32;
size_t globalsz = n;
err = clEnqueueNDRangeKernel(theQueue, theKrnl, 1, NULL,
                                &globalsz, &localsz, 0,
                                NULL, NULL);
CL_CHECK_ERROR(err);

// Read back the results
for (int i=0;i<n;i+=4) {
                err = clEnqueueReadBuffer(theQueue, devP, CL_TRUE,
            i*sizeof(float), sizeof(float), &f, 0, NULL, NULL);
                CL_CHECK_ERROR(err);

                std::cout<<"arr["<<i<<"]="<<f<<"\n";
}

// Cleanup
clReleaseMemObject(devP);
clReleaseKernel(theKrnl);
clReleaseProgram(theProg);
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

GPU code: quoted string

Motivation: Inline Kernels

```
static const char *theSource="/* Simple OpenCL: */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"    int i=get_global_id(0);\n"
"    arr[i]+=v;\n"
"}\n";
```

- OpenCL code goes in as a string
  - More futureproof than PTX
  - Allows metaprogramming
- Hardcoded strings are tedious
  - Must quote every line
- Strings from files: I/O paths, CPU/GPU disconnect

Solution: Stringify with Macro

```
#define QUOTE_OPENCL(code)   #code
static const char *theSource=QUOTE_OPENCL(
__kernel void writeArr(__global float *arr,float v) {
int i=get_global_id(0);
arr[i]+=v;
});
```

- Use preprocessor to make string
  - Feels a bit like C#/Java/C++0x
- Supports multi-line expressions
  - But bare commas need varargs
- Allows OpenCL and C++ to be intermixed naturally

Motivation for Expressive OpenCL

```
/**
  "simple" OpenCL example program
  Adapted from the SHOC 1.0.3 OpenCL FFT caller code.

Dr. Orion Sky Lawlor, lawlor@alaska.edu, 2011-05-29 (Public Domain)
*/
#include <iostream>
#include <stdio.h>
#include <assert.h>
#include "CL/cl.h"
#define CL_CHECK_ERROR(err) do{if (err) {printf("FATAL ERROR %d at " __FILE__
":%d\n",err,__LINE__); exit(1); } } while(0)

cl_device_id theDev;
cl_context theCtx;
cl_command_queue theQueue;
cl_kernel theKrnl;
cl_program theProg;

static const char *theSource="/* Lots more code here! */\n"
"__kernel void writeArr(__global float *arr,float v) {\n"
"                int i=get_global_id(0);\n"
"                arr[i]+=v;\n"
"}\n";


int main()
{
cl_int err;

// Set up OpenCL
// Get the platform
enum {MAX_PLAT=8, MAX_DEVS=8};
cl_platform_id platforms[MAX_PLAT];
cl_uint num_platforms=MAX_PLAT;
err= clGetPlatformIDs(MAX_PLAT,platforms,&num_platforms);
CL_CHECK_ERROR(err);
cl_platform_id cpPlatform=platforms[0];

//Get the devices
cl_device_id cdDevices[MAX_DEVS];
err=clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, MAX_DEVS, cdDevices, NULL);
theDev=cdDevices[0];
CL_CHECK_ERROR(err);

// now get the context
theCtx = clCreateContext(NULL, 1, &theDev, NULL, NULL, &err);
CL_CHECK_ERROR(err);

// get a queue
theQueue = clCreateCommandQueue(theCtx, theDev, CL_QUEUE_PROFILING_ENABLE,
                                &err);
CL_CHECK_ERROR(err);
```

```
// Create the program...
theProg = clCreateProgramWithSource(theCtx, 1, &theSource, NULL, &err);
CL_CHECK_ERROR(err);

// ...and build it
const char * args = " -cl-mad-enable  -cl-fast-relaxed-math ";

err = clBuildProgram(theProg, 0, NULL, args, NULL, NULL);
if (err != CL_SUCCESS) {                    ...                }

// Set up input memory
int n=64; int bytes=n*sizeof(float);
cl_mem devP = clCreateBuffer(theCtx, CL_MEM_READ_WRITE, bytes,
                             NULL, &err);
CL_CHECK_ERROR(err);

float f=1.2345;
err = clEnqueueWriteBuffer(theQueue, devP, CL_TRUE,
                           0, sizeof(float), &f, 0, NULL, NULL);
CL_CHECK_ERROR(err);


// Create kernel
theKrnl = clCreateKernel(theProg, "writeArr", &err);
CL_CHECK_ERROR(err);

// Call the kernel
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devP);
CL_CHECK_ERROR(err);

float addThis=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(float), &addThis);
CL_CHECK_ERROR(err);

size_t loca
size_t glol
err = clEn

CL_CHEC

// Read ba
for (int i=0

        i*

}

// Cleanup
clReleaseMemObject(devP);
clReleaseKernel(theKrnl);
clReleaseProgram(theProg);
clReleaseCommandQueue(theQueue);
clReleaseContext(theCtx);

return 0;
}
```

Kernel Arguments:

- Too much code
- Not typesafe
- Silent failure modes

OpenCL Kernel Argument Passing

```
// OpenCL kernel arguments: (__global float *ptr,int value)
cl_mem devPtr=...;
err=clSetKernelArg(theKrnl, 0, sizeof(cl_mem), &devPtr);
int val=1000;
err=clSetKernelArg(theKrnl, 1, sizeof(int), &val);
```

- One function call per argument
  - Discourages use of GPU
- All parameters must match, or runtime err
  - Pass too many? Runtime error.
  - Pass double to cl_mem? Crash!
  - Pass int to float? Wrong answer!
- C'mon! Do it at compile time!

Solution: Template Arguments

```
// OpenCL kernel arguments: (__global float *ptr,int value)
//  C++ template: gpu_kernel<void (__global<float*>,int)>
template <typename T0,typename T1>
class gpu_kernel<void (T0,T1)> {public: ...
void operator()(T0 A0,T1 A1) {
    checkErr(clSetKernelArg(k, 0, sizeof(T0), &A0));
    checkErr(clSetKernelArg(k, 1, sizeof(T1), &A1));
}
};
```

- Specialized templated function object
  - "weird C++ magic" (cf Boost, Thrust)
- Instantiate template from our kernel macro
  - So same arguments in OpenCL & C++
- Compile-time argument promotion and

FILL Kernels and Expressive Programming

Problem: Shared Memory Access

- Multithreaded code is hard
- It's easy to have multiple threads overwrite each others' results
  - Array indexing malfunctions
  - 2D or 3D arrays: row, column?
  - Glitchy, various HW/SW/config
- "Where does this data go?" is useless cognitive burden
  - Again, solve it *once*

Solution: FILL kernel

GPU_FILLKERNEL(float, addf, (float v), { result+=v; })

- **result ≡ <u>your</u> array value, at <u>your</u> index**
  - **Read and written by EPGPU automatically**
  - **Essentially new language keyword**
  - **Automatically does array indexing**
  - **Surprisingly easier for user**
- **Lots of new potential for library**

Example: FILL kernel

```
GPU_FILLKERNEL(float, addf, (float v), { result+=v; } )
```

Input and return type

Kernel name
(in OpenCL and C++)

Arguments

User code

```
// Call from C++ using operator=
    myArray=addf(2.34);

// Plain C++ CPU-side equivalent:
   for (int i=0;i<len;i++) addf(&myArray[i],2.34);
```

Example: FILL kernel (Generated)

GPU_FILLKERNEL(float, addf, (float v), { result+=v; })

```
// Generated OpenCL:
__kernel void addf(int length,__global float *array, float v)
{
int i=get_global_id(0);
if (i<length) {
    const int result_index=i;
    float result=array[result_index];
    { result+=v; }
    array[result_index]=result;
}
}
```

Extra arguments

Bounds check (local vs global)

Also has 2D indexing

Related Work

- Thrust: like STL for GPU
  - But CUDA is NVIDIA-only
- Intel ArBB: SIMD from kernel
  - Based on RapidMind
  - When will we see GPU support?
- Many other parallel languages
- My “GPGPU” library
  - Based on GLSL: nice; but limited!

Performance Examples

Example: EPGPU Hello World

```
#include "epgpu.h"
#include <iostream>

/* OpenCL code: return value = array index plus a constant*/
GPU_FILLKERNEL(float, do_work, (float k), { result = i+k; } )

/* C++ code: allocate, run, and print */
int main() {
int n=1000;
gpu_array<float> arr(n); /* make storage on GPU */
arr=do_work(10000.3); /* run code on GPU */
for (int i=0;i<n;i+=100) { /* read the result */
    std::cout<<"arr["<<i<<"] = "<<arr[i]<<"\n";
}
}
```

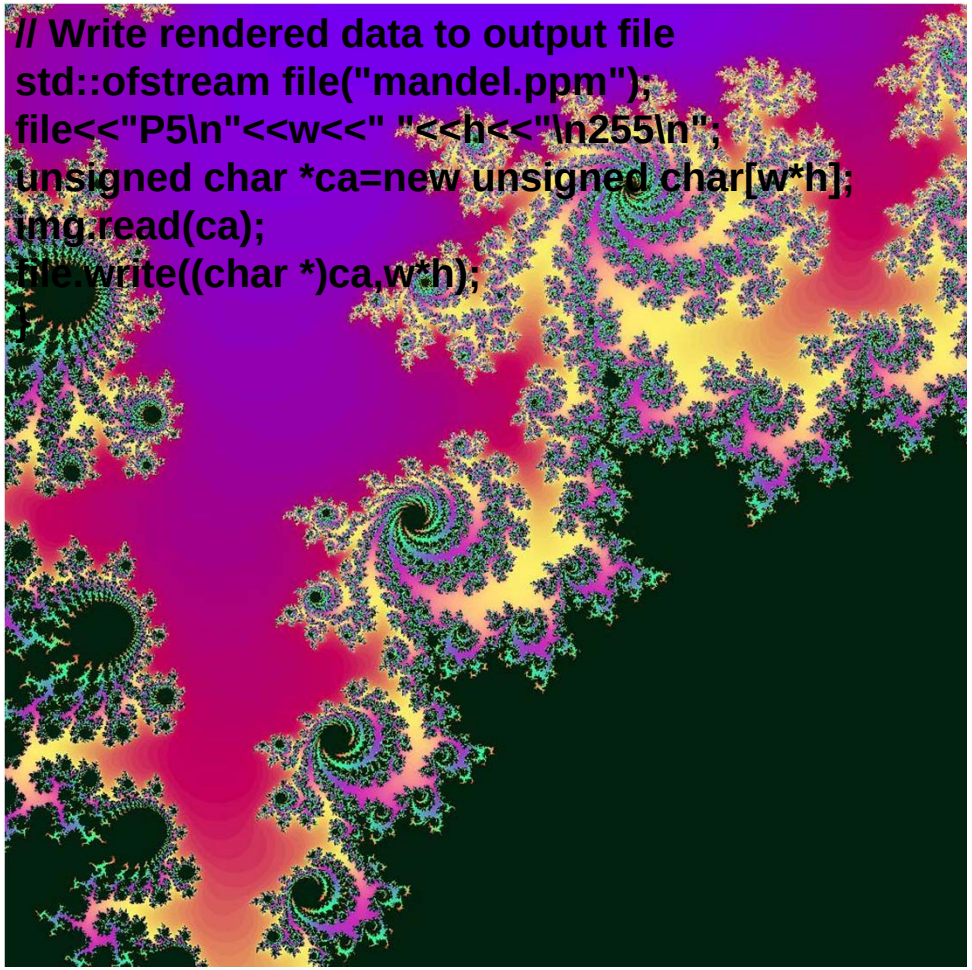
Example: EPGPU Mandelbrot

```
#include "epgpu.h"
#include <fstream>

/* OpenCL code */
GPU_FILLKERNEL_2D(unsigned char,
   mandelbrot, (float sz,float xoff,float yoff),
   {
/* Create complex numbers c and z */
float2 c=(float2)(i*sz+xoff,(h-1-j)*sz+yoff);
float2 z=c;
/* Run the mandelbrot iteration */
int count;
enum { max_count=250};
for (count=0;count<max_count;count++)
{
        if ((z.x*z.x+z.y*z.y)>4.0f) break;
        z=(float2)(
                z.x*z.x-z.y*z.y + c.x,
                2.0f*z.x*z.y + c.y
        );
}
/* Return the output pixel color */
result=count;
   }
)
```

```
/* C++ main function */
int main() {
int w=1024, h=1024;
gpu_array2d<unsigned char> img(w,h);
img=mandelbrot(0.00001,0.317,0.414);
```

```
// Write rendered data to output file
std::ofstream file("mandel.ppm");
file<<"P5\n"<<w<<" "<<h<<"\n255\n";
unsigned char *ca=new unsigned char[w*h];
img.read(ca);
file.write((char *)ca,w*h);
}
```



Example: EPGPU Stencil

```
... headers, initial conditions ...
/* Do one neighborhood averaging pass over src array. */
GPU_FILLKERNEL_2D(float,
stencil_sweep,(__global<float *> src),
int n=i+w*j; // 2D to 1D indexing
if (i>0 && i<w-1 && j>0 && j<h-1) { // Interior
        result = (src[n-1]+src[n+1]+
                        src[n-w]+src[n+w])*0.25;
} else { // Boundary--copy old value
        result = src[n];
}
)

int main() {
int w=1024, h=1024;
gpu_array2d<float> stencil_src(w,h);
stencil_src=stencil_initial(0.01,6.0,2.4,3.0); // an EPGPU FILLkernel (not shown)
for (int time=0;time<1000;time++) {
        gpu_array2d<float> stencil_dst(w,h); // cheap, due to buffer reuse inside the library
        stencil_dst=stencil_sweep(stencil_src); // EPGPU kernel above
        std::swap(stencil_dst,stencil_src); // ping-pongs the buffers
}

... do something with the resulting image ...
```

Example Performance

	CPU	6850	280	460M	580
poly3	2GB/s	92GB/s	115GB/s	38GB/s	83GB/s
mbrot	10GF	156GF	192GF	92GF	372GF
stencil	2GB/s	77GB/s	86GB/s	49GB/s	223GB/s
naiveT	1GB/s	3GB/s	3GB/s	14GB/s	63GB/s
localT	0.2GB/s	19GB/s	29GB/s	24GB/s	93GB/s

- **EPGPU seems to be performance competitive with hand-coded OpenCL & CUDA**
- **Most GPU applications are memory bound (gigabytes, not gigaflops)**
- **Fermi cards (460M, 580) are much more lenient for irregular memory access patterns**

The Future

GPU/CPU Convergence

- GPU, per socket:
  - SIMD: 16-32 way (“warps”)
  - SMT: 2-128 way (register limited)
  - SMP: 4-36 way (“SMs”)
- CPUs will get there, soon!
  - SIMD: 8 way AVX (or 64-way SWAR)
  - SMT: 2 way Intel; 4 way IBM
  - SMP: 6-8 way/socket already
    - Intel has shown 48 way many-core chips
- Biggest difference: CPU has

The Future: Memory Bandwidth

- Today: 1TF/s, but only 0.1TB/s
- Don't communicate, recompute
  - multistep stencil methods
  - FILL lets compiler reorder writes
- 64-bit -> 32-bit -> 16-bit -> 8?
  - Spend flops scaling the data
  - Split solution + residual storage
    - Most flops use fewer bits, in residual
  - Fight roundoff with stochastic rounding
    - Add noise to improve precision

Conclusions

- C++ is dead. Long live C++!
- CPU and GPU on collision course
  - SIMD+SMT+SMP+network
- Software is the bottleneck
  - Exciting time to build software!
- EPGPU model
  - Mix C++ and OpenCL easily
  - Simplify programmer's life
  - Add flexibility for runtime system
  - Open Source: please use & extend!