Embedding OpenCL in C++ for Expressive GPU Programming

Orion Sky Lawlor lawlor@alaska.edu

ABSTRACT

We present a high performance GPU programming language, based on OpenCL, that is embedded in C++. Our embedding provides shared data structures, typesafe kernel invocation, and the ability to more naturally interleave CPU and GPU functions, similar to CUDA but with the portability of OpenCL. For expressivity, our language provides the FILL abstraction that releases control over data writes to the runtime system, which both improves expressivity and eliminates the chance of memory race conditions. We benchmark our new language EPGPU on NVIDIA and AMD hardware for several small examples.

1. INTRODUCTION

Modern hardware systems present enormous parallelism—at the instruction level (superscalar), vector level (SIMD), thread level (SMT), core level (SMP), and across the network—yet modern software such as C++ or C# is primarily serial. Building usable abstractions for parallel software has been identified as a key challenge for 21st century computer science [17]. In particular, the modern Graphics Processing Unit (GPU) combines 16–48 way parallel SIMD execution, thousands of SMT threads, and 4-32 SMP cores to execute thousands of floating point instructions per clock cycle. To take advantage of this GPU hardware parallelism, application software must provide approximately million-fold parallelism while carefully managing memory accesses.

In this work, we explore several simple methods to express massive parallelism by embedding OpenCL in C++. We call the corresponding library expressive programming for GPU: EPGPU. In addition to C++ embedding, our other key enabling technology is encoding detailed knowledge about the hardware performance model into the runtime system. As shown in Figure 1, our system makes it straightforward to mix C++ and OpenCL.

1.1 Prior Work

OpenGL's GL Shading Language (GLSL or GLslang), introduced in 2003, is a C++-like language for computing the colors of screen pixels based on texture lookups and arbitrary amounts of C++-like arithmetic. GLSL supports branching, looping, function calls, a restricted form of user-defined classes, and a surprisingly rich standard library, including access to a full set of texture lookup functions including mipmaps. Since many games use GLSL, it is comparatively reliable, high performance, and widely supported by OpenGL drivers. The biggest language limitation of GLSL is that each shader execution can only write to the pixel currently being rendered; other writes are not allowed, which means some applications such as bitonic sort cannot be easily written in GLSL. However, mipmapping support means that some applications such as multigrid are actually easier to write in GLSL than in more recent languages, as we discuss in the appendix.

Microsoft's DirectX 9 High Level Shading Language (HLSL) and NVIDIA's Cg are contemporary with and quite similar to GLSL. Brook [1] is a stream programming language using HLSL as a backend. Shallows [7] is a more recent GPL GPGPU library using OpenGL's GLSL, but the wrapper only slightly reduces the complexity of doing GPU work, and client code must still make a number of calls to the underlying glut library.

NVIDIA's CUDA, introduced in 2007, has become the *de facto* standard programming model for expressing GPU parallelism. The performance potential of CUDA is incredible, and the fact it is a compiled language allows excellent CPU-GPU integration. However, CUDA only runs on NVIDIA GPUs, and its plain C style pointers-and-memcpy interface is somewhat error prone.

Thrust [5] is an excellent C++ template wrapper around CUDA's memory and kernel interfaces, combining the best of high performance and high productivity. Our work shares many of these goals, but is based on OpenCL.

OpenCL [6] is a recent cross-platform library for heterogeneous computing, including both GPU and CPU SIMD/SMP execution. High performance implementations exist for x86 SSE CPUs, AMD GPUs, NVIDIA GPUs, the Cell Broadband Engine, and even upcoming cell phone GPUs. Like GLSL, OpenCL kernels are typically uploaded to the runtime system as strings and compiled at runtime. Like CUDA, access to memory is via bare pointers, which provide no protection against buffer overruns or memory race conditions.

Other recent work has built a GPU backend for Haskell array codes [3], and several groups are working on Matlab GPU backends such GPULib [16]. These vector-style interfaces are easy to use, but the performance is not always as high as a kernel-style interface.

Intel has released several parallel programming toolkits, most recently the Array Building Blocks (ArBB) [15], a parallel language embedded within C++ using template-lambda-calculus style techniques. ArBB is an evolution of RapidMind, itself the commercial evolution of the composeable shader language Sh [13]. ArBB is capable of scaling to industrial strength problems, and has a solid implementation for CPU SIMD instructions such as SSE and AVX, but does not yet support a GPU backend.

The runtime system and library nature of this work is similar in spirit to our own previous work on the CUDA message passing library cudaMPI [9], an automatic parallelizing powerwall library MPIglut [12], parallel unstructured mesh library ParFUM [11], and Adaptive MPI [8]. A similar runtime system is the Charm++ Accelerator Interface [18], a runtime system which automatically overlaps computation and communication using asynchronous kernel executions.

2. GPU PROBLEMS & SOLUTIONS

In this section, we explore the software challenges posed by modern GPU hardware, and begin to explore how those challenges can be addressed.

2.1 PCI Bus and GPU Driver Latency

The GPU's arithmetic rate, measured in trillions of floating point operations per second, is often limited by the memory and I/O bandwidth. The GPU is distant from the CPU, and so it takes time to coordinate GPU and CPU activities across the PCI-Express bus. For example, as shown in Table 1, on an NVIDIA GeForce GTX 580[1] the OpenCL kernel shown in Figure 1 costs about 5 microseconds end to end latency to set up, but once started the kernel can read, modify, and update one 32-bit floating point number every 0.05 nanoseconds. Put another way, we can only issue 200,000 kernels per second, modify 20 billion memory floats per second, and calculate trillions of floats per second.

The memory/arithmetic performance gap is well known, but this factor of 100,000 difference between kernel latency and bandwidth can also dominate the execution time for GPU applications. For this reason, we must make it easy for users to maximize the size and minimize the total number of OpenCL kernel calls.

[1]Intel Core i5 2400 3.1GHz, 8GB RAM, Linux 2.6.38, g++ 4.4.3, CUDA 3.2

```
GPU_FILLKERNEL(float,
 poly3gpu, (float a,float b,float c,float d),
 { // OpenCL code
   if (result>=0) {
     result=((a*result+b)*result+c)*result+d;
   }
 }
)
void poly3(gpu_array<float> &arr,float x)
{ // C++ code
   arr=poly3gpu(-0.01,x,0.01,0.2);
}
```

Figure 1: GPU polynomial evaluation in EPGPU. Each entry in the array is read, the polynomial is evaluated with that value, and the result is written back to GPU memory.

NVIDIA GeForce GTX 580	
OpenCL Function	**Performance Model**
Kernel	5 us + 0.05 ns/float
Host Read	46 us + 0.66 ns/float
Host Write	25 us + 0.68 ns/float
Write, Kernel, Read	95 us + 1.38 ns/float
Allocate	100 us + 0.05 ns/float
Allocate, Write, Kernel, Read	329 us + 1.43 ns/float
AMD Radeon 6850	
OpenCL Function	**Performance Model**
Kernel	8 us + 0.09 ns/float
Host Read	316 us + 1.39 ns/float
Host Write	333 us + 2.25 ns/float
Write, Kernel, Read	1659 us + 3.62 ns/float
Allocate	364 us + 2.00 ns/float
Allocate, Write, Kernel, Read	2598 us + 5.23 ns/float

Table 1: Measured performance for basic OpenCL operations on various hardware. Latencies are in microseconds per operation; memory rates in nanoseconds per 32-bit float.

The well known bandwidth cost of reading and writing data back over the PCI-Express bus to the host CPU are shown in Table 1. These figures correspond to a few gigabytes per second of bandwidth. However, note the enormous latency of memory transfers—if there are several small pieces of data to transfer, rather than make several small transfers it is more efficient to call even several kernels to consolidate memory on the GPU before copying one large piece. Unlike in CUDA, in OpenCL there currently appears to be no latency or bandwidth improvement from using mapped buffers, with or without CL_MEM_ALLOC_HOST_PTR.

The substantial costs of allocating new GPU memory are shown in Table 1. Note that merely calling clCreateBuffer takes less than a microsecond, but any use of the new memory triggers the actual allocation. In addition to the per-float cost of allocation, which is at least as expensive as writing memory, allocation incurs a startup latency that is twenty times higher than a kernel's already high latency. Clearly, minimizing memory allocations is key for acceptable performance.

The combination of memory allocation and host copy cost is particularly acute. Allocating new GPU storage, writing CPU data into that storage, running a kernel, and reading the result back to the CPU can be over 50 times more expensive than running a GPU kernel on data left on the GPU.

One way to address the cost of transferring data is to leave the data on the GPU for all processing except I/O. In complex applications, this becomes difficult due to the number and variety of accesses to application data. Hence one of our key efforts is to simplify the process of defining and calling new GPU kernels. See Section 3 for details on how EPGPU interleaves C++ and OpenCL.

We can address the high cost of GPU memory allocation by simply re-using allocated GPU memory buffers. This is typically done manually, at the application level, but we feel it is more beneficial when performed by the runtime system. Unlike application-level code, the runtime system can reuse buffers between separate parallel components, resulting in better "performance compositionality" in addition to simpler application code.

We implemented buffer reuse in the usual way, by checking the buffer pool for existing space of compatible size in our gpu_buffer class constructor before EPGPU allocates a new buffer, and the destructor merely returns used buffers back to the pool. To avoid keeping too much memory in the pool, we keep a short leash on the number of buffers stored for reuse.

The performance improvement from buffer reuse is dramatic, as shown in Table 2. This measures the time to declare a new GPU data array and run a kernel on it. With current AMD drivers the improvement in both latency and bandwidth is over twentyfold! On NVIDIA hardware the improvement in bandwidth is only a factor of two, but the latency improvement is still enormous.

AMD Radeon 6850	
Without buffer reuse	282 us + 2.14 ns/float
With buffer reuse	10 us + 0.09 ns/float
NVIDIA GeForce GTX 580	
Without buffer reuse	85 us + 0.10 ns/float
With buffer reuse	3 us + 0.05 ns/float

Table 2: Performance improvement for buffer reuse.

2.2 Choosing a Workgroup Size

GPU hardware allocates threads in blocks, known in OpenCL as a "work group." Delivered performance as a function of workgroup size is shown in Table 3 for powers of two, and plotted in Figure 2 for every integer workgroup size. For kernels to take advantage of the hardware's parallelism, a work group needs to contain hundreds of threads. But larger workgroups are not always better; GPU software must respect the limits of clGetDeviceInfo(CL_DEVICE_MAX_WORK_GROUP_SIZE) and clGetKernelWorkGroupInfo(CL_KERNEL_WORK_GROUP_SIZE), which depends on the kernel's register usage. Using the largest possible workgroup may limit cross-workgroup parallelism, impacting performance.

These curves are shown for the kernel in Figure 1, but are similar for any short kernel. Longer kernels can give good performance at smaller workgroup sizes. The sawtooth pattern is due to the hardware's branch granularity (warp size), which is a lower bound on the useful workgroup size.

Ideally, the OpenCL driver would do a good job of automatically determining the workgroup size, but unfortunately in many cases the automatically determined workgroup is of size one. For example, if the size of kernel's domain is a large prime number, a terrible-performing single thread workgroup is the only way to divide the domain into an integral number of workgroups less than the hardware's limit.

To handle domain sizes that are not an even multiple of our chosen workgroup size, we insert a domain size check into each generated kernel, similar to "if (i<length)". These branches are highly coherent, only firing on the last workgroup, and hence do not appear to impact the code's performance. In any case, the benefit from using large workgroups drastically outweighs the cost of an additional branch.

On current hardware, for simple kernels the optimum workgroup size seems to be 256 threads, unless a lower limit is imposed by the hardware. This is the value our runtime system automatically targets for each kernel, unless overridden by the user. Our default seems to work well for most memory-bound kernels.

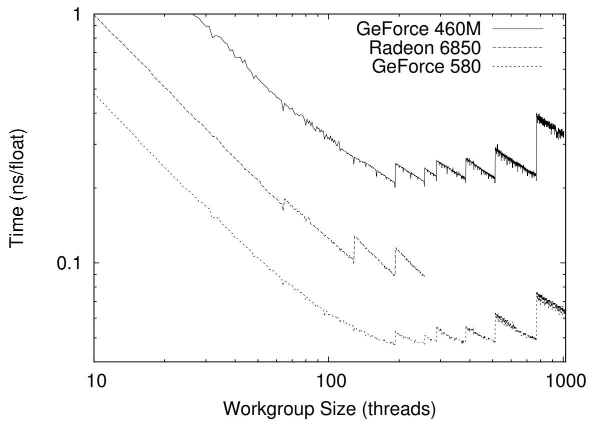

Figure 2: Timing as a function of workgroup size. Log-log plot. The Radeon card cannot handle workgroups over 256.

For multidimensional domains, such as 2D rectangles, our runtime system automatically divides the domain into square workgroups with a total size not exceeding our target workgroup size.

2.3 Memory Race Conditions

General purpose GPU languages like CUDA and OpenCL provide any kernel the ability to read and write from any location in GPU RAM. This flexibility allows a variety of interesting new applications to be written, but at the price of enabling the memory race conditions well known to shared memory programmers. Every part of every program must either ensure that multiple kernels never write to the same location in memory, or else use expensive atomic operations to synchronize the multiple writes. The "single writer" principle is easy to apply in theory, but in large complex programs it is difficult to guarantee.

Many kernels do not need the flexibility to write to any location in memory, but instead there is one unique destination for every kernel work-item's data. We call this a "FILL" type kernel, since it completely fills its memory. A FILL kernel is called on a destination region, and is free to read any memory except that region. Each FILL work-item does its own local computation and then hands the result off to the runtime system to be written to the destination memory region.

AMD Radeon 6850							
size	1	..	64	128	256	512	1024
ns/float	9.706	..	0.164	0.100	0.088	*n/a*	*n/a*
NVIDIA GeForce GTX 580							
size	1	...	64	128	256	512	1024
ns/float	4.500	...	0.081	0.053	0.047	0.048	0.060

Table 3: Effect of workgroup size on GPU performance. Small workgroups take many nanoseconds per float, while large workgroups compute many floats per nanosecond! Performance improves nearly linearly with workgroup sizes up to 64. ***n/a*** **means exceeds the maximum workgroup size.**

A typical read-modify-write 1D kernel, such as shown in Figure 1, works on a value "result" at array index "i"; similarly a 2D kernel works on array index "i" (column) and "j" (row). The EPGPU runtime system automatically generates these OpenCL variables via the following support code.

```
// OpenCL generated by the runtime system
int i=get_global_id(0);
if (i<result_length) { // bounds check
   float result=result_array[i]; // read

   // <- User code goes here == modify step

   result_array[i]=result; // write
}
```

For the user, the advantage of a FILL kernel is the lack of flexibility—user code cannot write past the end of an array or cause a memory race condition, because the runtime system does the memory writes. This is surprisingly liberating both for novice programmers, who are not yet comfortable with indexing, as well as experienced programmers, who know their pitfalls all too well.

For the runtime system, the fact that the FILL kernel is "well behaved" means a variety of interesting transformations become possible. For example, the array storage could be indexed using a space-filling curve for locality. A metakernel could be constructed that calls several versions of the FILL kernel and computes the variance of their results, for ensemble simulations or parameter sensitivity analysis. Along distributed memory boundaries, the runtime system could copy results directly into message buffers to be sent to other processors.

3. EMBEDDING OPENCL IN C++

Because OpenCL is implemented as a library, not a compiler, it is more difficult to provide a variety of basic features. But C++ provides an extremely rich set of tools for building domain-specific languages embedded inside C++, including powerful preprocessor macros, and partial specialization of user-defined templates. This section describes how we use these C++ features in EPGPU.

3.1 Mixing OpenCL with C++ Source

OpenCL source code is uploaded to the driver as a string, so the source code must either be stored in a separate file, or quoted as a string. Because adding quotes around every line is tedious and error prone, OpenCL code is normally relegated to a separate file. But because OpenCL code must always be called by C++, and in complex applications the call sequence is intricate, we find it is more readable to bring the OpenCL and C++ closer together.

```
// C++ code
cl_program p = GPU_SOURCECODE( //now OpenCL
   __kernel void fill_array
         (__global int *arr,int value)
   {
      int i=get_global_id(0);
      arr[i]=value;
   }
);
```

One method to achieve this syntax is using the C preprocessor feature called "stringification," which can quote a macro parameter at compile time. Our macro "#define GPU_SOURCECODE(code) gpu_compile(#code)" expands to include a quoted copy of everything inside the macro's parenthesis, which can be arbitrarily long, even including semicolons and extending across multiple lines.

This is surprisingly useful and reliable, with one minor exception: the preprocessor uses commas to separate macro arguments, so a compile error results from the use of commas outside parenthesis, such as in "int i,j;". The workaround is simply to separate the declarations into "int i; int j;", a slight restriction in the language.

3.2 Functors for Type Checking

Modern GPU programming revolves around kernels, task-parallel blocks of threads that share a common set of arguments. In our language, we represent OpenCL kernels in C++ using classes with an overloaded function-call operator; these function objects or "functors" provide a number of features.

For example, OpenCL kernel arguments are only checked for size at runtime, and never checked for data type. This means that in bare OpenCL, passing a C++ integer into an OpenCL floating point parameter, or passing a cl_mem handle representing an array of floats into a kernel expecting an array of int2, silently results in incorrect output at runtime. In a small program, this is relatively simple to find and correct; but in large programs with hundreds of kernels, it can be very difficult to track down the source of the error at runtime.

However, it is possible to use C++'s template partial specialization to both check OpenCL kernel arguments at compile time, and coerce compatible types into the correct data type. The fundamental trick here is to use templates to specialize our kernel functor object, and extract the template parameters to accept the appropriate arguments and upload them as OpenCL kernel arguments. Here is the two-argument version, which accepts CPU-side parameters from C++, coerces them to the correct datatype, and passes them to the GPU via OpenCL. The return type of our operator() is a runnable kernel object, as described in the next section.

```
template <class type0,class type1>
struct gpuKernel<void (type0,type1)> {
  ...
  runKernel &operator()(type0 v0,type1 v1)
  {
    clSetKernelArg(k,0,sizeof(v0),&v0);
    clSetKernelArg(k,1,sizeof(v1),&v1);
    return *this;
  }
};
```

Given the kernel's argument list, the compiler can pick and instantiate the appropriate specialization automatically, but we must somehow get the argument list into C++ from OpenCL. Forcing the user to manually duplicate the argument list in both C++ and OpenCL would inevitably lead to the sorts of mismatches we are trying to eliminate. However, it is possible to use a C++ macro to send the same argument list to both a C++ template specialization as well as an OpenCL code string, as shown below. The result of the following macro is the creation of a C++ function object with the same name as the OpenCL kernel, and including compile-time argument type checking.

```
#define GPU_KERNEL(kname,args,code)    \
class kname##_type :                   \
   public gpuKernel<void args> {        \
      static const char *getCode(void) {  \
         return "__kernel void "#kname#args\
           "{" #code "}";                  \
   }                                    \
} kname;
```

This macro-generated function object can then be used to define an OpenCL kernel in a reasonably natural fashion, aside from a few extra commas to separate the name, argument list, and function body.

```
// OpenCL: Define the kernel "doAtan"
GPU_KERNEL(
   doAtan,(__global<int *> d,int v), {
      int i=get_global_id(0);
      d[i]=atan2pi(v,i);
   }
);

// C++: Run "doAtan" for every index in arr
    arr=doAtan(a,17);
```

Unlike in OpenCL, in C++ "__global" is not a keyword. We work around this mismatch by making a C++ template "__global" as a typesafe wrapper around a bare OpenCL memory pointer "cl_mem". The "gpu_array" types decay into this when passed as arguments.

3.3 Running Kernels on Arrays

Now that we have function arguments, the only thing remaining to execute a kernel is the kernel's domain of execution (global work size), and for FILL kernels we need the destination array.

In EPGPU, we determine the domain of execution using an overloaded assignment statement "array=kernel(args);". For FILL kernels, the same assignment statement sets the given array as the destination for the results computed by the kernel, which are passed into OpenCL via generated kernel arguments not visible to the user.

For running a kernel on a novel domain, such as a subset of an existing grid, users can call the kernel's "run" method directly, supplying the dimensions the kernel should execute over.

4. PERFORMANCE EXAMPLES

We have extensively benchmarked the performance of our EPGPU library for several small applications on a variety of hardware, as summarized in Table4. The hardware is: a six-core AMD Phenom II 3.2GHz CPU; an $150 AMD Radeon 6850 desktop GPU; an older $200 NVIDIA GeForce GTX 280 desktop GPU; a laptop NVIDIA GeForce GTX 460M GPU; and a $500 NVIDIA GeForce GTX 580 desktop GPU. With sufficient attention to detail in the library, the same EPGPU binary runs perfectly on all these machines, which confirms OpenCL's binary interoperability. Bandwidths are measured using GPU manufacturers' standard: total global memory bytes read or written.

Broadly, the 6850 and 280 have good arithmetic rates, but memory accesses must be coalesced to be fast. By contrast, both the 460M and 580 are "Fermi" parts, with onboard coherent global memory caches, and so give better performance for non-coalesced memory access patterns. This difference is especially evident for *stencil*, and the matrix transposes *naiveT* and *localT* discussed in Section 4.3.

	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

Table 4: EPGPU example application performance on various hardware. Measurements are in gigabytes per second of memory bandwidth for memory-bound applications, and gigaflops for compute-bound applications. All applications benchmarked at 1024x1024 array resolution.

Figure 1 shows a trivial FILL kernel used to evaluate a cubic polynomial. This kernel runs at memory read-modify-write rate, about 80GB/s on a GTX 580.

4.1 Mandelbrot Example

Figure 3 shows the complete source code for a simple GPU Mandelbrot set renderer. First, note that the main function can immediately begin work: EPGPU automatically initializes the OpenCL library and compiles the kernel. The width (w) and height (h) parameters are truly arbitrary, and performance is still good even if they are large prime numbers.

OpenCL does not support complex numbers by default, and does not provide operator overloading, so the complex arithmetic is emulated using "float2" vectors. Figure 4 shows the colorized output. This is a complex area in the set, where high iteration count points are interleaved with low iteration count points, so branch granularity is a significant limiting factor. Aside from branches, the code is arithmetic bound, computing an average of 1,593 flops per pixel, and achieves 372 gigaflops/second on a GTX 580.

4.2 Stencil Example

Figure 5 shows a simple 2D five-point stencil on a regular grid. Compared to this simple example, real applications such as computational fluid dynamics do dramatically more arithmetic per grid cell, which improves the overall computation to memory ratio, but this trivial version maximally stresses the runtime system. This kernel benefits from global memory cache, running at about 220GB/s on the GTX 580. On Fermi cards, the code's performance seems to be competitive with a much more complex hand-optimized CUDA version exploiting __shared__ memory; as well as versions using texture memory (OpenCL images).

In main, the GPU 2D array "D" is created and destroyed at every iteration of the time loop, but this is actually efficient due to the buffer reuse described in Section 2.1. In stencil_sweep, the AMD OpenCL compiler actually seems to use a slower divide instruction when the average is written more naturally as sum/4.0 instead of sum*0.25; but either version runs at the same speed with the NVIDIA compiler. For further details on 3D stencil tuning on the GPU, see Micikevicius and Paulius [14], and also see a number of detailed optimizations and cross-platform performance comparisons in a recent survey [4].

```
#include "epgpu.h" /* our library */
#include <fstream>

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;
)

int main() {
   int w=1024, h=1024;
   gpu_array2d<unsigned char> img(w,h);
   img=mandelbrot(0.00001,0.317,0.414);

   // Write 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);
}
```

Figure 3: Mandelbrot rendering example.

4.3 Matrix Transpose

Figure 6 shows two versions of matrix transpose. The naive version is a trivial FILL kernel, but this results in scattered reads that perform especially poorly on older cards: about 3GB/s on the Radeon 6850 or GTX 280. To avoid the global memory scatter, the optimized version first does a set of contiguous global reads into local memory, switches indexing to essentially do a local transpose, and then writes back to global memory in the transposed order. This optimized version yields nearly a tenfold speedup on older hardware, and is still quite helpful even on the newest cards.

However, the number of lines of code needed is about fivefold more. Randal Schwartz popularized the quote "Make easy things easy, and make hard things possible," but languages and runtime systems can make hard things easy as well. Specifically, it should be possible to build abstractions to factor out the hardware knowledge expressed in local_transpose for use in other similar applications, future work we explore in the next section.

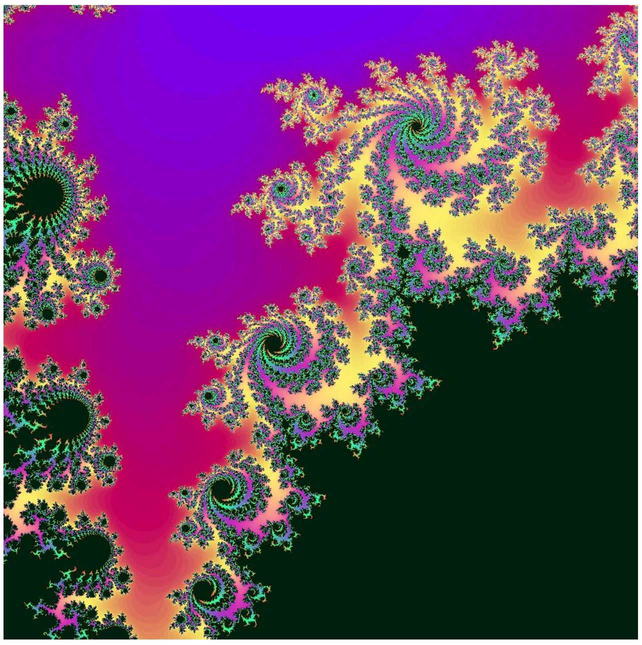

Figure 4: Colorized output from Mandelbrot renderer.

5. CONCLUSIONS AND FUTURE WORK

We have presented the design of a new GPU parallel language built from OpenCL and embedded inside C++. The language is designed to maximize programmer productivity by minimizing the distance between definition and use, reducing duplication of declarations and data structures, and eliminating opportunities for error.

This effort is just beginning, and an enormous amount of work remains to be done. The notion of well behaved FILL kernels can be exploited in a variety of interesting ways by runtime and library writers. Our language is under rapid development, supporting new features and more regular syntax for existing features. A more compiler-like preprocessing of the generated OpenCL code would allow a cleaner syntax for accessing 2D arrays, and allow us to insert runtime bounds checking for debugging. A key enabling technology for our future work is kernel data dependency analysis, which is dramatically simplified for kernels using our FILL kernel interface. Another idea we have not begun to exploit is the possibility of generating a C++ CPU-side version of each FILL kernel, which the runtime system could choose to use on datasets too small to benefit from the higher bandwidth of the GPU.

We have not addressed distributed-memory parallelism; although explicit memory-passing work along similar lines to cudaMPI [9] would be a natural addition, a higher-productivity solution would provide a globally shared view of GPU memory. An automated performance tuning component would be useful to optimize kernel sizes and work parameters. A more ambitious task would be to lazily evaluate GPU kernels, reducing kernel startup overhead by transparently combining multiple kernel calls.

We invite the reader to download, [10] experiment with, extend, and contribute to EPGPU!

```
/* Set up initial conditions */
GPU_FILLKERNEL_2D(float,
 stencil_setup,(float sz,float cx,float cy),
   float x=i*sz-cx; float y=j*sz-cy;
   if (sqrt(x*x+y*y)<=3.0) { // inside circle
      result=1.0;
   } else { // outside the circle
      result=0.0;
   }
)

/* Do neighborhood averages */
GPU_FILLKERNEL_2D(float,
 stencil_sweep,(__global<float *> src),
   int n=i+w*j; // our 1D array index
   if (i>0 && i<w-1 && j>0 && j<h-1)
   { // Interior--average four neighbors
      result = (src[n-1]+src[n+1]+
                src[n-w]+src[n+w])*0.25;
   } else { // Boundary--copy old value
      result = src[n];
   }
)

/* C++ driver and I/O code */
int main(void) {
   int w=1024, h=1024;
   gpu_array2d<float> S(w,h);
   S=stencil_setup(0.01,2.0,2.4);
   for (int time=0;time<1000;time++) {
      gpu_array2d<float> D(w,h);
      D=stencil_sweep(S);
      std::swap(D,S);
   }

   // Write data to output file
   std::ofstream of("stencil_image.ppm",
                    std::ios_base::binary);
   of<<"P5\n"<<w<<" "<<h<<"\n255\n";
   float *fa=new float[w*h];
   S.read(fa);
   char *ca=new char[w*h];
   for (int j=0;j<h;j++)
      for (int i=0;i<w;i++)
         ca[i+j*w]=(unsigned char)(
                    255.0*fa[i+j*w]
                  );
   of.write(ca,w*h);
   delete[] fa; delete[] ca;
   return 0;
}
```

Figure 5: A complete 2D five-point stencil example.

Acknowledgements

This research was supported in part by the NASA Applied Information Systems Research and NSF GK-12 programs. We would like to thank Chris Hartman for productive C++ discussions. Finally, we thank the AMD and NVIDIA OpenCL driver teams for their progress since early 2010: due to your efforts, a single OpenCL binary can now reliably execute on NVIDIA, AMD, or x86.

```
/* Simple naive 2D matrix transpose.
    src must have dimensions h x w. */
GPU_FILLKERNEL_2D(float,
 naive_transpose,(__global<float *> src),
   result=src[j+i*h];
);
... from C++ ...
    gpu_array2d<float> src(h,w),dst(w,h);
    src=...;
    dst=naive_transpose(src);
...

/* __local optimized matrix transpose.
      dst[i+j*w] = src[j+i*h]; */
GPU_KERNEL_2D( local_transpose,
     (__global<float *> dst,
      __global<float *> src,int w,int h),
 {
   enum N=16; // size of local work groups
   int2 G=(int2)(N*get_group_id(0),
                 N*get_group_id(1));
   int2 L=(int2)(get_local_id(0),
                 get_local_id(1));
   __local float loc[N*N]; // transpose here

   int2 S=G+(int2)(L.y,L.x); // Read flipped
   if (S.x<w && S.y<h)
     loc[L.y+N*L.x]=src[S.y+S.x*h];

   barrier(CLK_LOCAL_MEM_FENCE);

   int2 D=G+L; // Write forwards
   if (D.x<w && D.y<h)
     dst[D.x+D.y*w]=loc[L.x+N*L.y];
 }
);
... from C++ ...
   local_transpose.override_local[0]=16;
   local_transpose.override_local[1]=16;
   local_transpose(dst,src,w,h).run(w,h);
...
```

Figure 6: Naive and optimized matrix transposes.

6. REFERENCES

[1] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH ’04, pages 777–786. ACM, 2004.

[2] Ben Carter. *The Game Asset Pipeline*. Charles River Media, 2004.

[3] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP ’11, pages 3–14. ACM, 2011.

[4] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore

architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[5] Jared Hoberock et al. Thrust C++ template library for CUDA, 2011. http://code.google.com/p/thrust/.

[6] The Khronos Group. OpenCL—the open standard for parallel programming of heterogeneous systems. 2009. http://www.khronos.org/opencl/.

[7] Jon Mikkelsen Hjelmervik, Johan S. Seland, and Trond R. Hagen. Shallows library, 2009. http://sourceforge.net/projects/shallows/.

[8] Chao Huang, Orion Sky Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[9] Orion Sky Lawlor. Message passing for GPGPU clusters: cudaMPI. In *IEEE Cluster PPAC Workshop*, 2009. http://www.cluster2009.org/paperw102.php.

[10] Orion Sky Lawlor. EPGPU library, 2011. http://www.cs.uaf.edu/sw/EPGPU.

[11] Orion Sky Lawlor, Sayantan Chakravorty, Terry L. Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant V. Kale. ParFUM: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering With Computers*, 22(3):215–235, 2006.

[12] Orion Sky Lawlor, Matthew Page, and Jon Genetti. MPIglut: Powerwall programming made easier. *Journal of WSCG*, pages 130–137, 2008.

[13] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23:787–795, August 2004.

[14] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, New York, NY, USA, 2009. ACM.

[15] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *International Symposium on Code Generation and Optimization, Chamonix, France*, 2011.

[16] Brian E. Granger Peter Messmer, Paul J. Mullowney. GPULib: GPU computing in high-level languages. In *Computing in Science and Engineering*, volume 10, pages 70–73, 2008.

[17] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[18] Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008.

APPENDIX

GPGPU in OpenGL Shading Language (GLSL)

While paper primarily discusses OpenCL, this appendix discusses GLSL, the OpenGL Shading Language, a much older and more limited language. EPGPU's concept of FILL kernels is derived directly from GLSL.

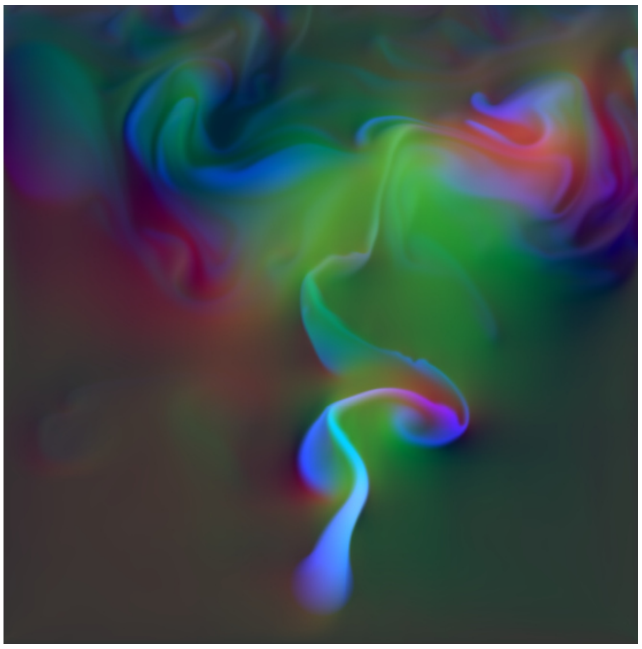

Figure 7: GLSL multigrid fluid dynamics, at 5ms/frame.

The most salient feature of GLSL is that it is intended for graphics. 2D or 3D textures are the primary source of data for GLSL shaders, and can be freely sampled at any location exactly as in OpenCL or CUDA. However, GLSL only supports a single externally visible "kernel" per source file, named main, and the only output from this kernel is a single multichannel builtin variable named "gl_FragColor." As the name implies, this variable determines the color written to the corresponding pixel in the current framebuffer, which can be bound to a 2D texture or 2D slice of a 3D texture.

Because the GLSL driver determines where and when each pixel is written, the driver has complete freedom to organize textures' memory layout to optimize performance. Internally, GLSL textures use a space-filling curve layout in memory [2], which results in excellent access locality for stencil, transpose, as well as more regular access patterns. Crucially, unlike OpenCL or CUDA, GLSL provides ready access to texture mipmaps, which are exceptionally useful for multigrid.

Figure 7 shows a 2D incompressible fluid dynamics simulator written in GLSL, and running at 200 million finished pixels per second (200fps at 1024x1024) using RGBA half float 8 byte pixels. In just 50 lines of GLSL, this simulator applies Stam-style upwind advection, solves the Helmholtz problem using a multigrid penalty method based on texture mipmaps, and applies all boundary conditions.

GLSL does not provide any access to GPU global memory (it has no pointers), nor GPU local memory (you cannot declare thread groups or shared data), and GLSL functionality can only be accessed by tediously making dozens of OpenGL calls. For some applications, these are serious limitations. But the conceptual execution model of GLSL is valuable: it easy to learn, literally impossible to crash, and scales to extremely high performance and surprisingly complex problems.