Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization

Qiming Hou* Kun Zhou† Baining Guo*‡

*Tsinghua University †Zhejiang University ‡Microsoft Research Asia

Abstract

We present a novel framework for debugging GPU stream programs through automatic dataflow recording and visualization. Our debugging system can help programmers locate errors that are common in general purpose stream programs but very difficult to debug with existing tools. A stream program is first compiled into an instrumented program using a compiler. This instrumenting compiler automatically adds to the original program dataflow recording code that saves the information of all GPU memory operations into log files. The resulting stream program is then executed on the GPU. With dataflow recording, our debugger automatically detects common memory errors such as out-of-bound access, uninitialized data access, and race conditions – these errors are extremely difficult to debug with existing tools. When the instrumented program terminates, either normally or due to an error, a dataflow visualizer is launched and it allows the user to examine the memory operation history of all threads and values in all streams. Thus the user can analyze error sources by tracing through relevant threads and streams using the recorded dataflow.

A key ingredient of our debugging framework is *the GPU interrupt*, a novel mechanism that we introduce to support CPU function calls from inside GPU code. We enable interrupts on the GPU by designing a specialized compilation algorithm that translates these interrupts into GPU kernels and CPU management code. Dataflow recording involving disk I/O operations can thus be implemented as interrupt handlers. The GPU interrupt mechanism also allows the programmer to discover errors in more active ways by developing customized debugging functions that can be directly used in GPU code. As examples we show two such functions: `assert` for data verification and `watch` for visualizing intermediate results.

Keywords: GPGPU, Stream Programming, Debugging, Interrupt

1 Introduction

To utilize the graphics processing unit (GPU) for general purpose computation, several high level languages have been developed recently, including CUDA, Brook+, Compute Shader and OpenCL [Lefohn et al. 2008]. These languages are based on the stream processing model, which matches the GPU's underlying stream architecture [Buck et al. 2004]. Programmers develop GPU applications

*e-mail: hqm03ster@gmail.com
†e-mail: kunzhou@acm.org
‡e-mail: bainguo@microsoft.com

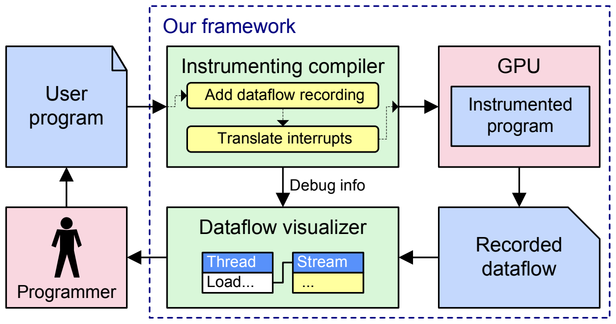


Figure 1: *System architecture of our debugging framework.*

by writing GPU stream programs, in which data is organized into uniform streams of elements and individual functions called kernels are applied to all elements of input streams in parallel to generate output streams. Both gather and scatter memory operations are allowed in GPU programs. While these C-like languages significantly reduce the difficulty of stream programming, debugging general purpose stream programs[1] remains an open problem.

Most existing GPU debugging tools are developed specifically for shader debugging. A shader has a simple, fixed dataflow [Microsoft 2007; NVIDIA 2008c]. Within individual rendering passes, there is a predefined correspondence between output vertices/pixels and shader threads. Erroneous shader threads can easily be located by visualizing shader outputs in the graphics pipeline. The error source can then be analyzed via simple means such as single stepping and visualization of graphics objects (textures, frame buffers, etc.). While multi-pass shaders can achieve highly complicated dataflow, typical shader debuggers still view multipass dataflow as simple, pre-defined dataflow between render-targets, textures and vertex buffers in individual passes. Thread-level inter-pass relationships are lost.

Debugging general purpose stream programs is a much more challenging problem. The reason is two-fold. First, a complex stream program usually requires multiple kernels and lots of streams. The dataflow between kernels is controlled by the programmer and can be arbitrarily complex. Since errors may propagate through the dataflow, an error in one kernel can go through many passes to manifest itself in another seemingly unrelated, perfectly correct kernel. In such cases, visualizing the outputs alone cannot locate the original error source among the kernels or threads. The programmer has to analyze the error source by investigating all possible kernels one by one, which is very time consuming and error prone. Second, the dataflow itself is a source of error. Dataflow errors like *race conditions* are non-existent in previous shader programming but are realistic threats to stream programming. A race condition is a situation where an output depends on execution order of two or more threads. Errors associated with race conditions are typically

[1]In this paper, the term "stream program" is interpreted as a multi-threaded program where the threads may be scheduled in any manner to hide memory/instruction latency.

non-deterministic and difficult to reproduce and detect manually.

In this paper we present a novel framework for debugging general purpose stream programs through automatic dataflow recording and visualization. As shown in Fig. 1, a stream program is first compiled using an instrumenting compiler. This compiler automatically inserts dataflow recording code into the original program to save the addresses, values and source code positions of all GPU memory operations into log files, and then generates an instrumented program that can be executed on the GPU. When the instrumented program terminates, either normally or due to an error, a dataflow visualizer is launched to analyze the recorded dataflow. Since the memory operation history of all threads and values in all streams has been recorded, the user can effectively analyze error sources by tracing through relevant threads and streams using the recorded dataflow. Another advantage of dataflow recording is that the instrumenting compiler can automatically insert code to perform runtime detection of memory errors on all streams. Several common errors, such as out-of-bound access, uninitialized data access and race conditions, are automatically detected during execution. Once such an error is detected, its information is recorded. The program is then aborted and the dataflow visualizer is launched to help the programmer locate the error.

A key ingredient of our debugging framework is *the GPU interrupt*, a novel mechanism that we introduce for calling CPU subroutines as normal functions from inside GPU code. The GPU interrupt is analogous to a software interrupt on the CPU: at each interrupt call, the processor context (i.e., the contexts of all calling threads) are saved and the interrupt handler, a CPU subroutine, is then called to operate on the saved thread contexts from which the threads resume. The interrupt handler has a global view of the thread contexts and can efficiently process contexts from all threads *en masse*. Currently, our GPU interrupt is achieved by a specialized compilation algorithm that effectively translates interrupts to GPU kernels and CPU management code.

The GPU interrupt is an extremely powerful debugging mechanism. Using GPU interrupts, our instrumenting compiler first inserts into the original stream program dataflow recording interrupts which handle disk I/O, and then translates the program along with these interrupts to a new stream program. Because the instrumented program runs directly on the GPU, it remains faithful to the original stream program in hardware specific behaviors. Moreover, the performance overhead for debugging is minimized. In all our experiments, the instrumentation introduces only a moderate (3-4$\times$) performance overhead. As a result, errors can be reproduced interactively.

An additional benefit of the GPU interrupt is that it allows the programmer to actively discover errors by developing his/her own debugging functions as interrupt handlers. These customized debugging functions can be called inside GPU code for error discovery in a more controlled manner. For example, simple debugging functions like `assert` are useful for detecting errors (see Listing 1). Another example is `watch` described in Section 4.4, which can be used to visualize values as colored bullets and is thus more intuitive and effective than a `printf`-style debug.

The remainder of this paper is organized as follows. The following section briefly reviews related work. Section 3 describes the syntax of the GPU interrupt as well as how to design an efficient compilation algorithm for GPU interrupts. Section 4 presents the actual debugging framework, including dataflow recording, automatic memory error detection, dataflow visualization and some custom debugging functions. Several debugging examples are described in Section 5, and the paper concludes with discussions about future work in Section 6.

2 Related Work

Most existing GPU debuggers are designed for the graphics pipeline and programmable shading [Purcell 2005; Microsoft 2007; NVIDIA 2008c]. See [Owens et al. 2007] for an excellent survey. Among these works, [Duca et al. 2005] presented a debugging system that allows the analysis of the complete rendering pipeline. In their system, programmers can write SQL-style queries to examine graphics states, including buffer data, shader variables, and the fixed dataflow between vertices and fragments. [Strengert et al. 2007] proposed a system specialized for shader debugging. Their system provides single stepping and variable watches of shaders using program instrumentation. Designed for the graphics pipeline, these debuggers cannot handle the complicated dataflow in stream programs for general purpose computation.

The CUDA framework provides built-in debugging support based on device emulation [NVIDIA 2008b]. By emulating all GPU code on the CPU, CPU functions such as `assert` and `printf` can be called in kernels. However, device emulation seriously hinders usability for two reasons. First, CPU emulation is over two orders of magnitude slower than the GPU. As a result, many real-time applications can only be debugged off-line. Second, CPU emulation may fail to reproduce some bugs occurring on the GPU as noted in [NVIDIA 2008b]. For example, CPUs and GPUs may not follow the same floating point standards. The result is that the same operation would yield different results, making it hard to reproduce precision related bugs (see the tessellation example in Section 5.3). Also, emulated threads are typically scheduled in a more deterministic manner on the CPU than the actual GPU, which significantly reduces the occurrence of race conditions. In contrast, the instrumented programs generated by our debugging system run directly on the GPU with moderate (3-4$\times$) performance overhead, and can reproduce all bugs in the original program.

NVIDIA has also released a CUDA-GDB debugger that runs directly on the GPU[2]. While that solves the emulation related problems mentioned above and provides a way to debug individual kernels, it still does not take dataflow into account, and thus cannot help to analyze and debug errors propagated through dataflow kernels. Also, our debugger can be easily ported to architectures without native GDB support like Compute Shader and OpenCL.

[Boyer et al. 2008] recently proposed a method to automatically detect CUDA shared memory race conditions and bank conflicts by program instrumentation. However, their work is limited to shared memory related analysis within a single kernel and ignores global dataflow. In addition, their work requires the program to be executed in device emulation mode and thus causes a slowdown of up to 800$\times$ when compared to a non-instrumented execution on the actual GPU.

Embedding CPU code in GPU code has been investigated in BSGP's `require` construct [Hou et al. 2008]. `require` resembles our GPU interrupt mechanism in that both rely on context saving. However, `require` only allows CPU code to be executed at barriers, i.e., natural kernel terminations. In contrast, the GPU interrupt can be inserted anywhere in a kernel. In addition, the GPU interrupt can take parameters from the GPU and return values to the GPU in a function-like manner.

Record/replay techniques have been used to debug traditional parallel programs on the CPU [Ronsse and Bosschere 1999; Ronsse et al. 2003]. However, these works focus on recording memory operation orders, which is significantly less useful than dataflow in

[2]Currently, CUDA-GDB is only supported on Linux. Also, two GPUs are required to run an application with graphics user interface (GUI) when debugging – one for debugging and one for the application.

GPU programs. In addition, CPU debuggers typically assume a small number of threads (tens to hundreds). They may not scale well to GPUs, on which it is common to have millions of threads.

3 GPU Interrupt

In this section we first introduce the syntax of our GPU interrupts. Then we discuss several challenging issues in designing an efficient interrupt compilation algorithm in Section 3.2 and present the algorithm itself in Section 3.3. Finally, we go over some extensions and limitations.

For simplicity we limit our discussion to CUDA since it is the state-of-the-art GPU programming language for general purpose computation. Our GPU interrupts and debugging framework should work equally well with any other GPU programming language.

3.1 Syntax

In CUDA, each function has its type which specifies whether it executes on the CPU or the GPU, and whether it is callable from the CPU or the GPU. As illustrated in Table 1, currently there are three types of functions: __host__, __device__ and __global__. As an extension of these languages, we introduce a novel function type, __interrupt__, which specifies a CPU function callable from the GPU.

Caller	CPU function	GPU function
CPU	__host__	__global__
GPU	__interrupt__ (our extension)	__device__

Table 1: *Function types in CUDA and our interrupt extension. The __host__ qualifier declares a CPU function that is callable from the CPU only, __global__ declares a GPU function that is callable from the CPU only, and __device__ declares a GPU function that is callable from the GPU only. Our __interrupt__ declares a CPU function that is callable from the GPU only.*

An interrupt can be called from any GPU code position, including inside a loop or a control flow instruction. Since it is a CPU function, it can call any __global__ function inside its implementation. Interrupt calling can be even nested or recursive, i.e., an interrupt can be called in kernels which are invoked by other interrupt handlers or its own handler (see Appendix for an example).

We now demonstrate the interrupt syntax using an example. Listing 1 is a triangle normal computation routine written in CUDA with GPU interrupts. assert is used to detect degenerate triangles. It calls the assertfail interrupt to display error messages when the assertion fails. As illustrated in line 24, calling an interrupt is similar to calling a GPU function. In function implementation, an interrupt is similar to a CPU function with a few exceptions. The function parameters are considered as GPU lists of their declared types, e.g., the rank in Listing 1. The user can get corresponding GPU pointers using the .d member in the GPU list structure. The number of threads calling the interrupt is also passed to the implementation as an implicit parameter interrupt::size.

3.2 Compilation Challenges

In complex CUDA programs, threads running the same kernel may take different execution paths due to control flow instructions. The differences in execution path among threads will be reinforced by interrupts. Different threads in the same kernel launch may have different interruption status, i.e., threads may be interrupted at different locations and some threads may terminate without calling any interrupt at all. For example, in Listing 1, most threads may pass the verification and will not call assertfail. In current graphics hardware, however, the only known way to transfer the control from the GPU to the CPU is by kernel termination, which is the same for all threads. The compilation algorithm thus has to make all threads terminate together, while maintaining the illusion of an inhomogeneous thread state for the original kernel. This results in two challenging issues in designing an efficient compilation algorithm for interrupts.

First, the interrupt processing mechanism has to be carefully designed to minimize overhead. A naive approach would be to test each thread's interruption status and then call the interrupt individually for each thread that will be interrupted. Unfortunately, this approach would result in intolerable overhead due to the high latency in each CPU/GPU data transfer operation. To address this issue, we first group threads with the same interruption status together on the GPU. Within each group, the interrupt handler and interruption position is the same. This allows interrupt processing to be performed *only once* for each group instead of for each thread. Furthermore, we combine parameters from all threads in the same group together and pass them to interrupt handlers as GPU temporary streams. This allows interrupt handlers to have full control over CPU/GPU data transfers. A handler may even invoke other kernels to process parameters on the GPU and thus avoid copying data back altogether. As a byproduct of this mechanism, our system allows nested and recursive interrupts. An interrupt may be called in kernels invoked by other interrupt handlers as well as its own handler.

The second challenge is resuming interrupted threads. Since the only way to transfer the control from the GPU to the CPU is by kernel termination, each thread has to be terminated at each interrupt. The kernel has to be re-launched later to resume the interrupted threads. Since each thread may be interrupted at a different position, we need to know the corresponding instruction pointers to correctly resume the interrupted threads. However, GPU instruction pointers are not directly accessible on current GPUs. Our solution to this problem is also based on the thread grouping scheme. Each thread stores a group ID when it reaches an interrupt or terminates normally. When the threads are re-launched, a switch statement based on the group ID is executed at the beginning of the kernel to jump to the corresponding address.

Listing 1 Normal computation with degeneracy detection

```
1  __device__ void assert(int flag);
2  __global__ void calcnormal(float3* N, float3* P, int n){
3      int id=threadIdx.x+blockIdx.x*blockDim.x;
4      if(id>=n)return;
5      float3 v0=P[id*3], v1=P[id*3+1], v2=P[id*3+2];
6      v1-=v0; v2-=v0;
7      float3 N0=cross(v1,v2);
8      assert(length(N0)>1e-5f*(dot(v1,v1)+dot(v2,v2)));
9      N[id]=normalize(N0);
10 }
11
12 __interrupt__ void assertfail(int rank){
13     int n=interrupt::size;
14     int* pr=new int[n];
15     cuMemcpyDtoH(pr,rank.d,n*sizeof(int));
16     for(int i=0;i<n;i++)
17         printf("Assert failed at thread %d\n",pr[i]);
18     delete pr;
19     exit(1);
20 }
21
22 __device__ void assert(int flag){
23     if(!flag)
24         assertfail(threadIdx.x+blockIdx.x*blockDim.x);
25 }
```

3.3 Compilation Algorithm

In the following we use Listing 1 to explain the compilation algorithm. The pseudo code after compilation is shown in Listing 2[3]. The algorithm works on a per-kernel basis. For each kernel that calls at least one interrupt, the following steps are performed:

1. Translate interrupt call.
2. Add return address dispatch code to the kernel.
3. Add interrupt processing code after kernel launch.
4. Translate interrupt handlers.
5. Add a loop to repeatedly launch the kernel and process the interrupt until all threads have reached kernel end.

In Step 1, the interrupt call is translated into a sequence of statements. For example, the `assertfail` in Listing 1 is translated into lines 24 to 29 in Listing 2 as follows:

- Store interrupt parameters (`__in` in line 24).
- Write interrupt ID to a temporary stream (`__ipid` in line 25). Normal termination is also associated with an ID. This ID is written to `__ipid` at the original kernel termination (line 33).
- Save processor context (`__ctx` in line 26).
- Jump to kernel end (the `return` in line 27).
- A label for the dispatch code to jump to in next launch (`__position0` in line 28).
- Restore processor context (line 29).
- Load interrupt return value, if any. As `assertfail` does not return any value, there are no such statements in Listing 2.

Note that each thread in a kernel launch can either call an interrupt or terminate normally. Since the kernel in Listing 1 has only one interrupt call, each thread's interruption state (i.e., the interrupt ID `__ipid`) has only two possible values, 0 for interrupted and 1 for normal termination. In case that the kernel contains multiple interrupt calls, the interrupt ID is no longer a 0/1 value. A distinct ID value is assigned for each interrupt call and the normal thread termination. Normal termination is assigned with the maximum ID. We also assign consecutive IDs to calls to the same interrupt at different program positions.

The context saving requires some special care. To avoid excessive GPU memory consumption, temporary streams allocated for context saving have to be minimized. In [Hou et al. 2008], a minimum flow algorithm is proposed for temporary stream allocation. However, it assumes context saving to be only performed at barriers, which is not necessarily true in our case. To address this problem, we adopt the graph coloring algorithm described in [Hack et al. 2006] to allocate temporary streams. In this algorithm, the kernel is first converted to static single assignment (SSA) form as in [Cytron et al. 1991]. We then construct an interference graph for active variables at interrupts by using these variables as nodes and drawing an edge between each variable pair that coexists at the same interrupt. Each valid temporary stream allocation scheme corresponds to a vertex coloring of this graph. By operating on the SSA form, the graph is guaranteed to be chordal, and optimal coloring can be computed in polynomial time. See [Hack et al. 2006] for algorithmic details and proofs.

In Step 2, a switch statement (lines 9 $\sim$ 11) is added to jump back to interrupt return points like `__position0`. Note that the statement is not executed in the initial kernel launch. It is only executed in

[3] Note that the pseudo code is only meant for illustration. The actual compiler works on a non-human-readable intermediate representation instead of source code.

Listing 2 Pseudo code of translated Listing 1

```
 1  //modified kernel
 2  template<int __first>
 3  __global__ void __true_calcnormal(
 4          float3* N, float3* P, int n,
 5          void* __in, int* __ipid, void* __ctx, int __sz){
 6      //dispatch code
 7      int __rank=threadIdx.x+blockIdx.x*blockDim.x;
 8      if(!__first){
 9          switch(__ipid[__rank]){
10          case 0:goto __position0;
11          case 1:return;
12          }
13      }
14      //begin of original kernel
15      int id=threadIdx.x+blockIdx.x*blockDim.x;
16      if(id>=n)goto __oldreturn;
17      float3 v0=P[id*3], v1=P[id*3+1], v2=P[id*3+2];
18      v1-=v0; v2-=v0;
19      float3 N0=cross(v1,v2);
20      //expanded assert
21      if(!(length(N0)>1e-5f*(dot(v1,v1)+dot(v2,v2)))){
22          //expanded interrupt assertfail(rank)
23          //save parameter
24          ((int*)__in)[__rank]=threadIdx.x+blockIdx.x*blockDim.x;
25          __ipid[__rank]=0;
26          __contextsave(__rank,__ctx,__sz,id,N0);
27          return;
28      __position0:
29          __contextload(__rank,__ctx,__sz,id,N0);
30      }
31      N[id]=normalize(N0);
32  __oldreturn:
33      __ipid[_rank]=1;
34  }
35
36  __host__ void assertfail(void* __in_sorted, int __nthread){
37      //get parameter streams from base, size, offset
38      stream<int> rank;
39      rank=get_substream<int>(__in_sorted,__nthread,0);
40      //begin of original code
41      int n=__nthread;
42      int* pr=new int[n];
43      cuMemcpyDtoH(pr,rank.d,n*sizeof(int));
44      for(int i=0;i<n;i++)
45          printf("Assert failed at thread %d\n",pr[i]);
46      delete pr;
47      exit(1);
48  }
49
50  //kernel launches replaced by calls to this function
51  __host__ void calcnormal(float3* N, float3* P, int n,
52  launch_dims /* block/grid dimensions */){
53      int __sz=get_thread_count(launch_dims);
54      int* __ipid=new_stream(sizeof(int)*__sz);
55      void* __in=new_stream(sizeof(int)*__sz);
56      void* __in_sorted=new_stream(sizeof(int)*__sz);
57      void* __ctx=new_stream((sizeof(int)+sizeof(float3))*__sz);
58      //the interrupt handling loop
59      for(int first=1;;first=0){
60          if(first)
61              __true_calcnormal<1><<<launch_dims>>>(
62                  N,P,n,
63                  __ipid,__in,__ctx,__sz);
64          else
65              __true_calcnormal<0><<<launch_dims>>>(
66                  N,P,n,
67                  __ipid,__in,__ctx,__sz);
68          //sort __in with respect to __ipid
69          //returns # threads at each ip
70          int __interrupt_sizes[2];
71          void* __in_sorted[1];
72          __sort_ipid(
73              __interrupt_sizes,__in_sorted,__in,__ipid,2);
74          if(__interrupt_sizes[0])
75              assertfail(__in_sorted[0],__interrupt_sizes[0]);
76          __del_streams(__in_sorted,1);
77          //break if all threads have ended
78          if(__interrupt_sizes[1]==n){break;}
79      }
80      del_stream(__ctx);
81      del_stream(__in_sorted);
82      del_stream(__in);
83      del_stream(__ipid);
84  }
```

resumed launches after interrupt processing. This is represented by the `__first` template parameter in Listing 2.

In Step 3, code is added after kernel launch to group together threads with similar interruption states. The routine, `__sort_ipid` in line 72, compacts interrupt input parameters to a temporary stream `__in_sorted` and computes the number of interrupted threads, `__interrupt_sizes[0]`, as well as the number of terminated threads `__interrupt_sizes[1]`. Threads are sorted with respect to their interrupt IDs. For efficiency, the sort is performed on the GPU. Because the number of different interrupt calls in a kernel is usually small, a base 2 radix sort suffices. Since consecutive IDs are assigned to different calls to the same interrupt in Step 1, the sort organizes the parameters consecutively into a single list. The interrupt handler is then executed once for all calls. This way, calls to the same interrupt at different code positions can be combined and processed together efficiently.

Note that the temporary streams for interrupt input parameters have to be allocated before kernel launch because GPUs do not allow memory allocation within kernels. Therefore, we need to reserve sufficient memory for all interrupts. Observing that each thread in a launch can call at most one interrupt, we share streams among parameters of different interrupts to avoid excessive memory consumption. We do this by packing similarly sized parameters (e.g., `int` and `float`) together. For each size s, we compute n_s, the maximal number of parameters having size s among all interrupts called. Then n_s streams with element size s are reserved prior to kernel launch. Parameter streams for each interrupt are allotted from these n_s streams independently. For example, when two interrupts `interruptA(int a)` and `interruptB(int b,int c)` are called in the same kernel, two streams are reserved. The parameters `a` and `b` are stored in one stream, whereas `c` is stored in the other.

In Step 4, if the number of interrupted threads is non-zero, the interrupt handler is called as an ordinary CPU function as illustrated in line 74. Also the function prototype is modified to take a packed interrupt parameter stream `__in_sorted` and to include the thread count `__nthread`, as is shown in line 36. Extra code is added to extract individual parameter sub-streams (line 39). The interrupt parameter packing scheme has already been explained above.

Finally in Step 5, a loop is added to launch the kernel repeatedly until all threads have reached the kernel end, as illustrated in lines 59 and 78. Note that such a loop is necessary even if the kernel contains only one interrupt, because some threads may be interrupted multiple times before termination, as is the case when the interrupt is called inside a `for` or `while` loop.

3.4 Discussions

We have extended the CUDA language to allow programmers to define and call interrupts in a CUDA program according to the syntax described in Section 3.1. The described interrupt compilation algorithm has been implemented in a prototype instrumenting compiler, which can convert CUDA programs containing interrupts to executable programs.

Note that NVIDIA may have a mechanism to suspend/resume GPU execution in current hardware, since the GDB debugger can pause execution [NVIDIA 2008a]. If such a mechanism is exposed in the programming interface in the future, it could be used to provide a hardware/partial-hardware implementation of our GPU interrupts, which may be more efficient than our current compiler-based approach. This may improve the performance of interrupts to a level suitable for non-debug operations, e.g., simulating page fault to provide virtual memory support.

Listing 3 Using an interrupt member function

```
struct cassert{
    char* f;
    int l;
    __interrupt__ void fail(){
        printf("Assertion failure at \"%s\"(%d)\n",f,l);
        exit(1);
    }
};

#define assert(d) {\
    require{\
        cassert as;\
        as.f=__FILE__;as.l=__LINE__;\
    }\
    if(d){as.fail();}\
}
```

CPU Parameters Sometimes interrupts need to receive additional parameters directly from the CPU. For example, a real `assertfail` requires a file name and a line number to indicate where the error occurs. In such cases, the programmer can put all CPU parameters in a structure and define the interrupt as a member function of the structure. This interrupt member function can then be called using a CPU object. Listing 3 provides an example of using an interrupt member function. The interrupt `fail` is declared as a member function of structure `cassert`. The macro `assert` constructs a CPU object `as` and uses it to call `fail`.

The construction of the CPU parameter object requires special care. In certain situations, the object construction is inseparable from some GPU code fragments. An example is the file name and line number at `assert` in Listing 3. They can only be obtained via two ANSI C macros, `__FILE__` and `__LINE__`, at the `assert` statement, which is in GPU code. In such cases, we allow the programmer to directly embed the object construction in GPU code using a `require` construct, which allows a block of CPU code to be inserted into GPU code as shown in [Hou et al. 2008]. The compiler relocates code inside a `require` block to execute it before the kernel containing the `require` is launched.

Interrupt member functions are compiled in a similar manner as ordinary interrupts. All `require` blocks are processed before interrupt compilation as shown in [Hou et al. 2008].

Limitations While the GPU interrupt mechanism provides a general approach for calling CPU functions from inside GPU code, our current implementation of the mechanism has some limitations. First, the algorithm is designed for the basic stream model we assume. It assumes the processor context only contains live variables and an instruction pointer. Advanced features may introduce additional processor states that have to be saved as part of the processor context. To handle such situations, the context saving part in our algorithm has to be extended. Examples include CUDA's shared memory and block barrier synchronization. To handle them, shared memory content and the set of threads reaching each barrier have to be included in the context saving. This extension is an interesting direction of future work. Note that our current algorithm has no problem with advanced features that do not introduce new processor states, e.g., texture, constant memory, atomic operations.

Second, since we save the contexts of all threads of a stream kernel, the processor context may also become too large to be saved. This could happen if a kernel's live variables are too large, e.g., if a large local array is allocated for each thread. The problem is inherently tied to the decoupling of computation and physical processor in the stream processing model. Due to the abstraction of physical processors in the stream model, the processor context can only be described by the contexts of all threads, whose size is unbounded.

One solution to this problem is to make stronger assumptions of the processing model, e.g., make assumptions about the physical processors.

Finally, to provide best effort service in all cases, our system is designed to make limitations take effect in program behavior rather than in compilation failure. For example, a program with mixed asserts and block synchronization would be compiled successfully. The program would run normally if no assertion is triggered, but it would crash or hang when there is an assertion failure. Likewise, if a programmer uses `debug::watch` in a kernel that uses shared memory, the watch visualizer would show up correctly, but the shared memory content would be thrashed after `debug::watch` returns.

4 Debugging System

The ultimate goal of debugging is to locate erroneous code fragments in the program and fix the resulting problems. This requires a mechanism to trace from exposed errors to root error sources. Our debugging system provides dataflow recording and visualization for this purpose.

In our framework, the programmer first compiles a stream program using the instrumenting compiler, which automatically adds dataflow recording code in the program. Although dataflow recording requires CPU code such as disk I/O operations, it can be easily implemented by inserting dataflow recording interrupts. The instrumenting compiler will then translate these interrupts and generates an instrumented program that can run directly on the GPU. When the instrumented program terminates normally or aborts due to a detected error, all recorded dataflow is written to temporary log files. The dataflow visualizer is then automatically launched to analyze the recorded dataflow.

In the following, we explain the debugging progress using a real example – a merge sort program written in CUDA. As shown in Listing 4, the kernel `bsmerge` takes an input stream `a` with `n` elements. Every `sz` elements in `a` are assumed to be sorted. `bsmerge` merges neighboring sorted subsequences in `a` to form a `2*sz` long sorted subsequence and writes the result to `b`. Finally, `msort` calls `bsmerge` repeatedly to merge neighboring sorted sequences until all `n` elements are sorted.

Listing 4 A buggy merge sort in CUDA

```
__global__ void bsmerge(int* b,int* a,int n,int sz){
    //@Binary search merge pass
    int id=blockIdx.x*blockDim.x+threadIdx.x;
    if(id>=n)return;
    //examine input
    int k0=a[id];
    debug::watch(k0);
    //enforce left<right order on equal numbers
    int k=k0-!(id&sz);
    //binary search in neighboring sz elements
    int l0,l,r;
    l0=l=((id&-sz)^sz);r=min(l+sz,n)-1;
    while(l<=r){
        int m=(l+r)>>1;
        int rk=a[m];
        if(rk<=k)l=m+1;else r=m-1;
    }
    //(l-l0) elements less than k0 in the neighbor
    b[(id&~sz)+(l-l0)]=k0;
}

void msort(int* b,int* a,int n){
    int nb=((n+255)>>8);
    int* tar=b;
    for(int sz=1;sz<n;sz+=sz){
        int* tmp;
        bsmerge<<<nb,256>>>(b,a,n,sz);
        tmp=a;a=b;b=tmp;
    }
    if(tar!=a){cuMemcpyDtoD(tar,a,n*sizeof(int));}
}
```

For each element `k0`, `bsmerge` computes the number of elements that are less than `k0` in the neighboring subsequence. This is done using a binary search. To break ties among elements with equal values, an order is enforced between the two subsequences by decrementing elements from one subsequence by 1. However, this tie breaking scheme is problematic. The decrement may cause an integer overflow and thus result in inconsistent ordering. This inconsistency may in turn lead to race conditions and uninitialized values, producing undefined results.

4.1 Dataflow Recording

By default, the instrumenting compiler will insert code to automatically record the following information:

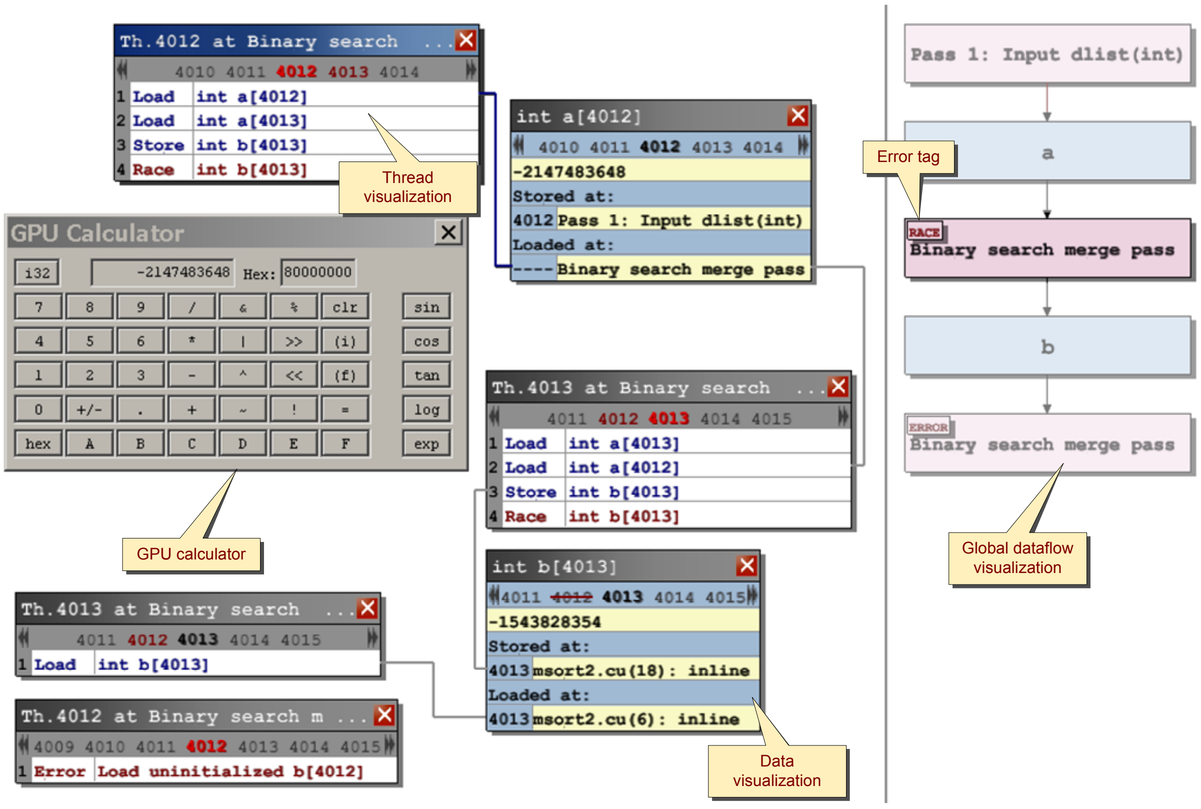
- Thread memory access history. Prior to each GPU memory access, recording routines are automatically added to write the address and the value loaded/stored to a log file. Note that disk I/O is required in the recording routines, and they have to be implemented using interrupts.
- Stream information. For each allocated stream, we record its name, type, size, and base address. Name, size and base address are obtained by intercepting GPU memory allocation APIs. The element type of each stream is determined upon the first time it is passed to a kernel. At runtime, this information is used to detect out-of-bound memory access, and is written to a log file. Note that the element type detection would fail if the debugged program does any non-trivial casting. Currently we rely on the programmer to provide a casting-free version for debugging using `#ifdef`.
- Stream initialization status. For each stream, a boolean initialization tag array is maintained on the GPU. Currently this array consumes one integer per stream element. Such tags are updated during memory writes and copies. Note that this information is only used for runtime detection of uninitialized memory accesses and is not written to log files.
- Source code positions. The compiler records the source code position at each GPU memory operation (i.e., memory access and allocation). At GPU code positions, we also record an inline function expansion history, which is equivalent to a call stack dump. Note that this information is generated during compilation and saved to a log file.

User-Controlled Dataflow Recording While dataflow recording can be fully automatic, recording the whole dataflow for all streams induces significant runtime overhead and is unnecessary for most debugging tasks where only a small portion of the complete dataflow is required. Optional user control is thus desirable in practice. We provide programmers with several functions for this purpose. Like interrupts, these functions can be regarded as extensions of the CUDA language. Table 2 is a list of these functions.

- Enabling/disabling recording via `__dflog_begin` and

Function	Usage
`__dflog_begin()`	Enable recording
`__dflog_end()`	Disable recording
`__dflog_forceinit(list)`	Mark `list` as initialized
`debug::record(expr)`	Record an expression

Table 2: *Functions for user-controlled dataflow recording.*



```
Th.4012 at Binary search ...
4010 4011 4012 4013 4014
1 Load   int a[4012]
2 Load   int a[4013]
3 Store  int b[4013]
4 Race   int b[4013]

int a[4012]
4010 4011 4012 4013 4014
-2147483648
Stored at:
4012 Pass 1: Input dlist(int)
Loaded at:
---- Binary search merge pass

GPU Calculator
i32   -2147483648   Hex: 80000000
7   8   9   /   &   %   clr   sin
4   5   6   *   |   >>  (i)   cos
1   2   3   -   ^   <<  (f)   tan
0   +/- .   +   ~   !   =     log
hex A   B   C   D   E   F     exp

Th.4013 at Binary search ...
4011 4012 4013 4014 4015
1 Load   int a[4013]
2 Load   int a[4012]
3 Store  int b[4013]
4 Race   int b[4013]

int b[4013]
4011 4012 4013 4014 4015
-1543828354
Stored at:
4013 msort2.cu(18): inline
Loaded at:
4013 msort2.cu(6): inline

Th.4013 at Binary search ...
4011 4012 4013 4014 4015
1 Load   int b[4013]

Th.4012 at Binary search m ...
4009 4010 4011 4012 4013 4014 4015
1 Error  Load uninitialized b[4012]

Pass 1: Input dlist(int)
a
RACE Binary search merge pass
b
ERROR Binary search merge pass
```

Figure 2: *The dataflow visualizer.*

__dflog_end. The instrumenting compiler automatically inserts a boolean variable in the instrumented program to control the recording status. Programmers can call these functions anywhere in a stream program to switch recording on/off at runtime. These functions are often used to cull unnecessary data and control the record size. They are especially useful for bugs that can only be reproduced with user interaction.

- Initialization status control via __dflog_forceinit. It sets initialization tags of all elements in list to *initialized*. This is provided to avoid false uninitialized errors caused by unrecorded memory operations or external streams.
- Expression recording via debug::record. The value of a user specified expression expr can be explicitly recorded. Such values will appear later in the dataflow visualizer. This function is easily implemented using an interrupt.

4.2 Automatic Memory Error Detection

An important advantage of dataflow recording is that the instrumenting compiler can automatically insert code to perform runtime detection of several classes of memory errors. Once an error is detected, its information is automatically recorded using interrupts. The program is then aborted and the dataflow visualizer is launched for error analysis.

Out-of-bound access can be detected in a straightforward fashion by testing each memory access address against the recorded stream base address and size. For this purpose, stream information is looked up prior to each kernel launch and passed to the GPU as additional parameters.

To detect uninitialized accesses, the boolean initialization array described in the last section is set to uninitialized in stream allocation. Subsequent memory copies and writes will set the affected address range's status to initialized.

Race condition detection is more complicated. As race conditions may be intentional in certain situations (e.g., chaotic relaxation algorithms [Chazan and Miranker 1969]), we do not abort the debugged program upon discovery of race conditions; instead they are displayed as warnings during dataflow visualization. We perform race condition detection off-line in the dataflow visualizer.

Note that in the stream processing model, only inter-thread data dependency can cause race conditions since threads cannot perform arbitrary synchronization with each other. A race condition occurs if and only if two or more threads access the same memory address and at least one access is a write. This makes traditional race condition detection techniques (e.g., [Savage et al. 1997] and [Flanagan and Freund 2004]) unnecessary. With all memory accesses recorded, race condition check can be done in a straightforward fashion by examining all accesses to the same addresses in each kernel launch.

4.3 Dataflow Visualization

Once the instrumented program terminates, the dataflow visualizer is launched to analyze the recorded dataflow. If an error is detected, a window for the error causing thread is opened immediately after visualizer start-up. The programmer can then navigate through the dataflow via dependency and peer relationships between threads and stream elements.

Fig. 2 is a screen shot of a debugging process of Listing 4. See the supplementary video for a more detailed demonstration of this process. In this case, thread 4012 in the second bsmerge launch has read an uninitialized element b[4012] and the program is terminated. The dataflow visualizer is launched and thread 4012's operation history is displayed. The programmer then navigates to a number of thread/stream elements in the following order: thread 4013 in the second `bsmerge` launch (a peer thread of thread 4012), element `b[4013]` (load), thread 4013 in first launch (store), thread 4012 in first launch (race condition peer). At this point, the cause of error becomes clear. In the first `bsmerge` launch, thread 4012 and 4013 should have sorted `a[4012]` and `a[4013]` and stored the result to `b[4012]` and `b[4013]`. However, they both wrote to `b[4013]` instead and `b[4012]` is never written. Further investigation exposes the root cause: `a[4012]` is `0x80000000`, the minimum `int` value. The subtraction in Listing 4, line 9 resulted in an integer overflow and thus made subsequent comparisons inconsistent.

Here is a list of the most important functionalities provided by the dataflow visualizer:

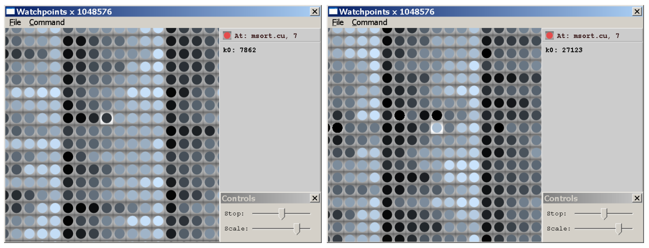
- Thread visualization. All recorded memory operations of a thread are displayed in the order in which they happened. The `Th ...` windows in Fig. 2 are thread windows.
- Data visualization. For each value written to each data element, i.e., *data instance*, the value itself and all operations on the value are displayed. The `int ...[...]` windows in Fig. 2 are data windows.
- Peer tracing. From each thread and data instance, the programmer can navigate through their peers. Each thread's peers are threads in the same kernel launch. Each data instance's peers are data instances in the same stream. Threads with errors and uninitialized data instances are marked. Peers are displayed in the top bar in thread/data windows.
- Global dataflow visualization. All kernel launches and input/output streams are organized as a flow chart. Kernel launches containing errors or race conditions are tagged. The kernel launch or data instance of current focus is highlighted. This is illustrated in the flow chart in the right of Fig. 2.
- Dependency tracing. The programmer can navigate from a thread to an accessed data instance, or vice versa, by clicking on the corresponding operation. Dependencies between displayed threads and data instances are also visualized as lines.
- Going back to the source code. The call stack at each memory access or error can be mapped to source code position. The corresponding file will be opened in a text editor, with the cursor at the line of interest.
- GPU calculator. A calculator is provided to perform some key GPU operations that are rarely found in CPU calculators, e.g., `__int_as_float`. For maximum consistency, all floating point operations of the calculator are executed on the GPU.

4.4 Custom Debugging Functions

Debugging is not always triggered by erroneous program behavior and output – this type of error discovery is completely passive for the programmer. On the CPU, the programmer can utilize debugging functions such as `assert` to discover errors in a more active and controlled manner. Our GPU interrupt allows programmers to develop their own debugging functions and make these debug functions available to GPU programs.

The use of `assert` is illustrated in Listing 1. It performs data verification. Another useful function is `printf` which can be used to examine GPU intermediate results. However, while `printf` can be easily implemented using interrupts, the huge data size of typical stream programs makes `printf` inefficient.

We implemented a more efficient function, `debug::watch`, allowing programmers to visualize intermediate results as pixel colors in an image. This function provides intuitive knowledge of massive data to the programmer. It also makes errors manifest as inconsistent pixels, which are easier to recognize by casual examination than `printf` dumps. In line 7 of Listing 4, a `debug::watch` interrupt is used to visualize values as colored bullets. As illustrated in Fig. 3, the bullets are colored with a brightness proportional to the sort key. Sorted subsequences correspond to the progressively brighter bullet groups in Fig. 3(a). Erroneously ordered elements correspond to inconsistent brightness as in Fig. 3(b).



(a) Correct result (b) Erroneous result

Figure 3: *Colored-bullet visualization of partially sorted lists in Listing 4. Light dots indicate smaller values and dark dots indicate larger values.*

The `debug::watch` visualizer has a few additional features. Numerical values can be displayed by mousing over the corresponding bullet. The color can be set and adjusted at run-time. Bullet size can be adjusted to examine errors at different scales. To allow detection of inconsistently colored bullets of sub-pixel size, the bullets are rendered with HDR color and analytical anti-aliasing. Unlike the dataflow visualizer which is launched after the instrumented program terminates, the `debug::watch` visualizer is automatically launched during the execution of the instrumented program.

Note that `assert` and `debug::watch` are not new language features; they are functions enabled by GPU interrupts. The programmer can implement his/her own customized visualization functions in a similar way.

5 Experimental Results

We have implemented the described instrumenting compiler and the dataflow visualizer in a prototype debugging system *CUDAdb* for CUDA programs. The system is used on a daily basis by all GPU programmers in our lab and has been proven to be very useful in several projects. We are going to make the system publicly available in the near future. In the following, we describe several GPU applications that we debugged using CUDAdb. These applications are simple and easy-to-describe examples chosen solely to demonstrate different aspects of our system. All examples are conducted on a machine with an Intel Xeon Dual-Core 3.7GHz CPU and a NVIDIA Geforce GTX 280 graphics card. Our experiments indicate that CUDAdb can effectively help the programmer to discover and locate commonly-occurring programming errors that are very difficult to debug with existing debugging tools.

5.1 Reyes-Style Subdivision

The first example is an implementation of the Reyes-style subdivision algorithm in [Patney and Owens 2008]. The algorithm performs adaptive subdivision for Bézier patches in breadth-first order. In each iteration, a bound and split kernel is launched to process all patches in parallel. A screen space bounding box is computed for each patch. Patches outside the view frustum are culled. Patches with a bounding box larger than a predefined threshold are split into two sub-patches by using the De Casteljau algorithm. Other patches are left unmodified. All newly-generated and unmodified

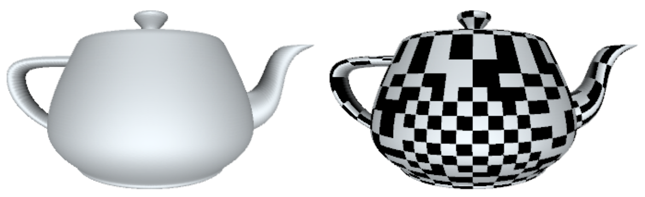
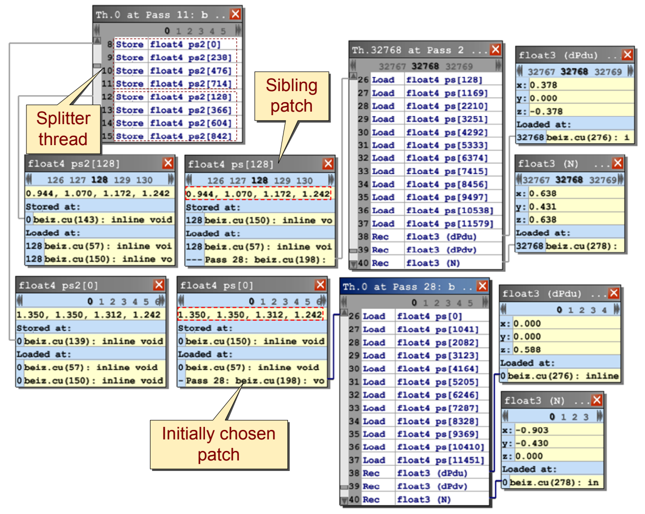

(a) Correct result (b) Erroneous result

Figure 4: *Reyes-style rendering of a teapot. In (b), some patches are incorrectly shaded, resulting in a broken checkerboard pattern.*



```
Th.0 at Pass 11: b ..
               0 1 2 3 4 5
 8 Store  float4 ps2[0]
 9 Store  float4 ps2[238]
10 Store  float4 ps2[476]
11 Store  float4 ps2[714]
12 Store  float4 ps2[128]
13 Store  float4 ps2[366]
14 Store  float4 ps2[604]
15 Store  float4 ps2[842]
```

```
float4 ps2[128]
   126 127 128 129 130
0.944, 1.070, 1.172, 1.242
Stored at:
0   beiz.cu(143): inline void
Loaded at:
128 beiz.cu(57): inline voi
128 beiz.cu(150): inline vo
```

```
float4 ps[128]
   126 127 128 129 130
0.944, 1.070, 1.172, 1.242
Stored at:
128 beiz.cu(150): inline vo
Loaded at:
128 beiz.cu(57): inline voi
--- Pass 28: beiz.cu(198):
```

```
float4 ps2[0]
            0 1 2 3 4 5 6
1.350, 1.350, 1.312, 1.242
Stored at:
0 beiz.cu(139): inline void
Loaded at:
0 beiz.cu(57): inline void
0 beiz.cu(150): inline void
```

```
float4 ps[0]
            0 1 2 3 4 5 6
1.350, 1.350, 1.312, 1.242
Stored at:
0 beiz.cu(150): inline void
Loaded at:
0 beiz.cu(57): inline void
- Pass 28: beiz.cu(198): vo
```

```
Th.32768 at Pass 2 ...
   32767 32768 32769
26 Load  float4 ps[128]
27 Load  float4 ps[1169]
28 Load  float4 ps[2210]
29 Load  float4 ps[3251]
30 Load  float4 ps[4292]
31 Load  float4 ps[5333]
32 Load  float4 ps[6374]
33 Load  float4 ps[7415]
34 Load  float4 ps[8456]
35 Load  float4 ps[9497]
36 Load  float4 ps[10538]
37 Load  float4 ps[11579]
38 Rec   float3 (dPdu)
39 Rec   float3 (dPdv)
40 Rec   float3 (N)
```

```
float3 (dPdu) ..
 32767 32768 32769
x: 0.378
y: 0.000
z: -0.378
Loaded at:
32768 beiz.cu(276): i
```

```
float3 (N) ...
32767 32768 32769
x: 0.638
y: 0.431
z: 0.638
Loaded at:
32768 beiz.cu(278):
```

```
Th.0 at Pass 28: b ..
              0 1 2 3 4 5
26 Load  float4 ps[0]
27 Load  float4 ps[1041]
28 Load  float4 ps[2082]
29 Load  float4 ps[3123]
30 Load  float4 ps[4164]
31 Load  float4 ps[5205]
32 Load  float4 ps[6246]
33 Load  float4 ps[7287]
34 Load  float4 ps[8328]
35 Load  float4 ps[9369]
36 Load  float4 ps[10410]
37 Load  float4 ps[11451]
38 Rec   float3 (dPdu)
39 Rec   float3 (dPdv)
40 Rec   float3 (N)
```

```
float3 (dPdu) ...
        0 1 2 3 4
x: 0.000
y: 0.000
z: 0.588
Loaded at:
0 beiz.cu(276): inline
```

```
float3 (N) ...
        0 1 2 3
x: -0.903
y: -0.430
z: 0.000
Loaded at:
0 beiz.cu(278): in
```

Figure 5: *Error analysis of the buggy Bézier subdivision.*

patches are then copied to a new, compact list. The iteration continues until all patches are unmodified. The resulting patches are uniformly subdivided into sub-pixel sized quads using a dicing kernel. Finally the quad vertices are shaded, and rasterized using OpenGL.

The bug examined here is a real bug that occurred during our implementation. As illustrated in Fig. 4(b), the buggy program produced correct geometry but inccorrect shading.

Observing that the error pattern resembles the patch pattern, the programmer uses the dataflow visualizer to compare the splitting process of two neighboring patches. To do this, the programmer first chooses an arbitrary patch from the dice kernel's input stream `ps`. The programmer then traces through data dependency to find the thread that generated the patch via splitting. Its sibling patch is found by examining the splitting thread's access history. The programmer then traces back to a dicing thread for the sibling patch, as is illustrated in Fig. 5.

At this point the cause of error becomes clear. The two patches have opposite normals due to opposite derivatives `dPdu`. The inconsistently ordered derivative is due to the opposite control point coordinate ordering of the sibling patches. Further analysis reveals the bug: the programmer implemented the De Casteljau routine for Bézier patch subdivision based on a Wikipedia article [Wikipedia 2008]. While that routine produces correct geometry, it does not maintain derivative orientation. After fixing this routine, the program works correctly and produces Fig. 4(a). Listing 5 shows the problematic De Casteljau routine.

Note that this bug is difficult to find using traditional means for two reasons. First, while this error has its origin in the splitting algorithm, it manifests itself in the unrelated dicing/shading kernel. Without a global view of dataflow, considerable time would have

Listing 5 The problematic De Casteljau routine

```
__device__ float deCasteljau(
float4 a,float t,float4& side0,float4& side1){
    float p=lerp(a.x,a.y,t),
        q=lerp(a.y,a.z,t),
        r=lerp(a.z,a.w,t);
    float u=lerp(p,q,t),v=lerp(q,r,t);
    float w=lerp(u,v,t);
    side0=make_float4(a.x,p,u,w);
    //this causes inconsistent orientation
    //should be: side1=make_float4(w,v,r,a.w);
    side1=make_float4(a.w,r,v,w);
    return w;
}
```

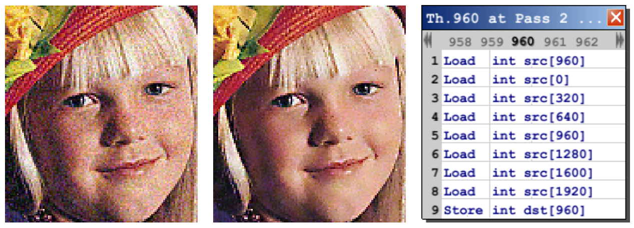
been wasted in examining the dicing/shading kernel, which actually contains no bugs. Second, the best way to analyze this bug is to compare a correct patch with a neighboring incorrect patch. However, the subdivision algorithm does not store neighboring patches consecutively. Without a global view of the dataflow, locating a pair of neighboring patches can be very time consuming. By providing global view of dataflow, our dataflow logging framework provides effective solutions to both problems.

Due to the cost of dataflow recording and memory error detection, our instrumentation results in some performance/memory overhead. In this subdivision example, the peak memory is 36MB for the original program and 43MB for the our instrumented program. The frame rate decreased from 142fps to 66fps when dataflow recording is disabled. Recording the dataflow of one frame causes a 1-2 seconds stall. Note that in practice, the dominant source of overhead is context saving. The overhead of an interrupt is more dependent on the kernel it is inserted to rather than what the interrupt actually does. As a consequence, it is inherently difficult to generalize quantitative results in one example to another. The overhead evaluation needs to be performed on a case-by-case basis.

5.2 Image Denoising

While our framework is designed for inter-kernel debugging, its dataflow visualization features are also very useful when examining the work flow of individual kernels. In this example, we implemented an image denoising algorithm. The algorithm takes an noisy image like in Fig. 6(a) as input. Bilateral filtering is performed within a 7×7 window around each pixel. The ratio of pixels above a pre-defined weight threshold is also computed. The output is computed by blending the filtered pixel value and the original pixel value with a weight based on this ratio. The denoised image is shown in Fig. 6(b).

When implementing the algorithm, the programmer made a typo during a copy-paste operation. The resulting program failed to denoise the input image. Its output is still noisy, and looks similar to the input image. In the following, we demonstrate the process of



```
Th.960 at Pass 2 ...
 958 959 960 961 962
1 Load  int src[960]
2 Load  int src[0]
3 Load  int src[320]
4 Load  int src[640]
5 Load  int src[960]
6 Load  int src[1280]
7 Load  int src[1600]
8 Load  int src[1920]
9 Store int dst[960]
```

(a) Noisy image (b) Denoised image (c) A buggy thread

Figure 6: *Noised input, denoised output and the access history of a buggy thread.*

Listing 6 The main loop of the denoise kernel

```
1  // rwindow = 3
2  for(int i=-rwindow;i<=rwindow;i++){
3      for(int j=-rwindow;i<=rwindow;i++){
4          float3 c=cextract(src[
5              min(max(y+i,0),h-1)*w+
6              min(max(x+j,0),w-1)]);
7          //...
8      }
9  }
```

locating this bug using CUDAdb.

The program is compiled using our instrumenting compiler and then executed. The access history of an arbitrary thread is examined as in Fig. 6(c). Note that this thread has only loaded 8 pixels. In a correct implementation, each thread should load at least 49 pixels (7×7 window). Further investigation shows that the programmer wrote `i` instead of `j` in the inner `for` statement (line 2 in Listing 6). As a result, the program only loops over the first column of the filter window.

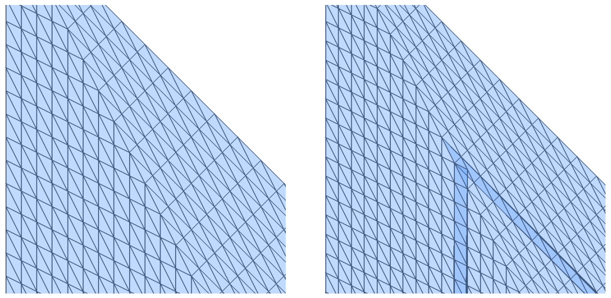
5.3 Triangle Tessellation

We select the triangle tessellation example because it demonstrates the advantage of the direct GPU execution of our instrumented program over CUDA's built-in CPU emulation. The triangle tessellation algorithm is taken from [Hou et al. 2008] and implemented using CUDA.

The tessellation algorithm is parallelized over all output vertices. It organizes output vertices in a number of triangle strips. For each output vertex, the triangle strip containing it is first computed. The barycentric coordinates are then calculated based on the triangle strip ID. For some of the vertices, a quadratic equation needs to be solved to get the strip ID. This involves a `floor` integer rounding of a square root and adding an epsilon value. How to choose an appropriate epsilon value depends largely on experience. An inexperienced programmer often makes mistakes and chooses an improper epsilon that generates erroneous vertices with sufficiently large tessellation factors. As a precaution, an `assert` is inserted to validate the triangle strip ID. In the following, we compare two implementations of `assert`: GPU interrupt and CPU emulation.

For simplicity, our implementation only deals with a single triangle. We set the integer rounding epsilon as 10^{-6}. With a tessellation factor of 143, the integer rounding off produces an incorrect value, resulting in several invalid triangles as illustrated in Fig. 7. Table 3 is a comparison of the behavior and performance among the original program, our instrumented program and CUDA's built-in device emulation. To allow a performance comparison, the programs are not aborted on assertion failures. Only error messages are printed. Also in the instrumented program, dataflow recording is disabled and can be enabled when the user presses a hotkey at run-time. The overhead in the performance comparison only includes memory error detection and `assert`. For fairness, the CPU floating point internal representation is set to a similar precision as the GPU using the method described in [NVIDIA 2008b].

As shown in Table 3, CPU emulation failed to reproduce the bug due to higher arithmetic precision on the CPU. The emulation is also over 300× slower than the original program. In contrast, our interrupt-based approach successfully reproduced the bug via direct device execution. With all memory error checks enabled and the `assertfail` interrupt called in each iteration, the instrumented program is only about 4× slower than the original program. In terms of memory consumption, the original program consumes around 10MB while our instrumented program takes about 12MB.



(a) Correct result at factor 119 (b) Erroneous result at factor 143

Figure 7: *Zoom-in of tessellated triangles. In (b), two malformed long triangles are overlapping with other triangles.*

Program	Original	Our Instrumented	CPU Emulation
Error reproduced	yes	yes	no
Frame rate (fps)	710	175	2.14

Table 3: *Behavior and performance comparison of tessellation with precision problem.*

6 Conclusion

We believe the described debugging framework significantly advances the state of the art in GPU stream programming. While most existing GPU debugging tools are only suitable for shader debugging, we found our system to be highly effective for debugging general purpose stream programs and for locating errors that are extremely difficult with existing tools. A key ingredient of our debugging system – and a major contribution of this paper – is the GPU interrupt, a novel mechanism that allows calling CPU functions from GPU code. With the GPU interrupt, dataflow recording can be implemented in a straightforward fashion as interrupt handlers. The GPU interrupt also empowers GPU programmers by allowing them to actively discover errors by writing their own debugging functions as interrupt handlers. In the future we expect the GPU interrupt to be widely useful beyond debugging, much the same way the CPU interrupt is for CPU programming.

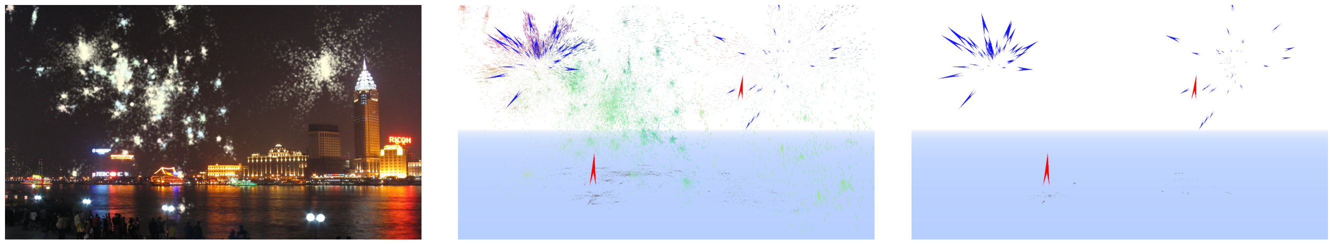
For future work, we plan to incorporate the interrupt mechanism into other general purpose GPU programming languages including Compute Shader, OpenCL and BSGP. Currently, our GPU interrupt is implemented through a special compilation algorithm. We hope to see future GPUs with support for this mechanism at the hardware level. Secondly, we are also interested in exploring the use the dataflow recorded by our debugger as interactive education materials. With a properly designed visualization scheme, fresh GPU programmers can run code samples with different parameters and observe the corresponding dataflow changes. This visualization can provide intuitive knowledge of the work flow of GPU programs – a type of knowledge that is difficult to obtain otherwise.

Acknowledgements

The authors would like to thank Matt Callcut for his help with video production. We are also grateful to the anonymous reviewers for their helpful comments. Kun Zhou is partially supported by the NSF of China (No. 60825201) and NVIDIA.

References

Boyer, M., Skadron, K., and Weimer, W. 2008. Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems*.



(a) Fireworks rendering, 150fps (b) Particles, 112fps (c) Non-leaf particles, 117fps

Figure 8: *One frame in the particle simulation, rendered in three different modes. This frame contains 42k particles. The rendering resolution is* 1024×576 *with* 2×2 *supersampling.*

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3, 777–786.

CHAZAN, D., AND MIRANKER, W. L. 1969. Chaotic relaxation. *Linear Algebra Appl. 2*, 199–222.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

DUCA, N., NISKI, K., BILODEAU, J., BOLITHO, M., CHEN, Y., AND COHEN, J. 2005. A relational debugging engine for the graphics pipeline. *ACM Trans. Gr. 24*, 3, 453–463.

FLANAGAN, C., AND FREUND, S. N. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL '04*, ACM, New York, NY, USA, 256–267.

HACK, S., GRUND, D., AND GOOS, G. 2006. Register Allocation for Programs in SSA-Form. In *Compiler Construction 2006*, Springer, A. Zeller and A. Mycroft, Eds., vol. 3923, 247–262.

HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Gr. 27*, 3, 9.

LEFOHN, A., HOUSTON, M., BOYD, C., FATAHALIAN, K., FORSYTH, T., LUEBKE, D., MUNSHI, A., OLICK, J., OWENS, J., PELLACINI, F., PHARR, M., AND SHOPF, J. 2008. Beyond programmable shading. ACM SIGGRAPH 2008 Course Notes.

MICROSOFT, 2007. PIX: the Direct3D profiling and debugging tool. DirectX 10 SDK.

NVIDIA, 2008. CUDA-GDB: The NVIDIA CUDA Debugger.

NVIDIA, 2008. NVIDIA CUDA Programming Guide 2.0.

NVIDIA, 2008. NVIDIA Shader Debugger.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1, 80–113.

PATNEY, A., AND OWENS, J. D. 2008. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Gr. 27*, 5, 1–8.

PURCELL, T. 2005. GPGPU: general-purpose computation on graphics hardware: debugging tools. ACM SIGGRAPH 2005 Course Notes.

RONSSE, M., AND BOSSCHERE, K. D. 1999. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst. 17*, 2, 133–152.

RONSSE, M., CHRISTIAENS, M., AND BOSSCHERE, K. D. 2003. Debugging shared memory parallel programs using record/replay. *Future Gener. Comput. Syst. 19*, 5, 679–687.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

STRENGERT, M., KLEIN, T., AND ERTL, T. 2007. A hardware-aware debugger for the OpenGL shading language. In *Graphics Hardware '07*, 81–88.

WIKIPEDIA, 2008. De Casteljau's algorithm. http://en.wikipedia.org/wiki/De_Casteljau's_algorithm.

Appendix: Particle Firework Example

This example demonstrates customized debugging functions and nested interrupts. We implemented a GPU particle system based fireworks demo. In this demo, each particle has three possible states: primary, secondary, and leaf. Primary and secondary particles will explode to generate new particles at a predefined time after their creation, whereas leaf particles simply fade out and disappear. Each primary particle explodes into 64 secondary particles and 4096 leaf particles. Each secondary particle explodes into 256 leaf particles.

Interrupts are used to implement a particle visualization system. By using interrupts, visualization in GPU kernels can be implemented with simple and clear code. This is illustrated in Fig. 8. Fig. 8(a) is the fireworks rendering. Fig. 8(b) is a visualization of the particles. Primary and secondary particles are shown as large red arrows and medium blue arrows, respectively. Leaf particles are shown as small green triangles. Triangles shadows are also rendered on a bottom surface to better illustrate their near-far ordering relationships. The size and orientation of the arrows/triangles indicate particle velocity. Fig. 8(c) omits all leaf particles, providing a more clear view of primary and secondary particles. The system also allows the user to switch between these three modes at run-time.

Listing 7 is the pseudo code of the particle visualization system. The core of the system is a triangle drawing interrupt `drawtriangle`. Using `drawtriangle`, particle visualization is implemented as a single kernel `gpu_drawparticles`. `gpu_drawparticles` tests particle type and calls `drawtriangle` multiple times to draw particle shapes. Finally, `gpu_drawparticles` is wrapped as an interrupt `drawparticles` and called in the particle simulation kernel. Note that no CPU/GPU data transfer is involved in the visualization pipeline.

Interrupts play an important role in this visualization system. Interrupts allow utility routines like `drawtriangle` to be called

Listing 7 Pseudo code of the particle visualization system

```
__interrupt__ void drawtriangle(
float3 a,float3 b,float3 c,float4 color){
    n=interrupt::size;
    //......
    glDrawArrays(GL_TRIANGLES,0,n*3);
}

__global__ void gpu_drawparticles(particles){
    //draw base triangle
    drawtriangle(......);
    if(is_nonleaf()){
        //draw arrow tails
        drawtriangle(......);
        drawtriangle(......);
    }
}

__interrupt__ void drawparticles(particles){
    gpu_drawparticles<<<...>>>(particles);
}

__global__ void simulate(particles, mode){
    //......
    if(should_visualize(particles[id], mode)){
        drawparticles(particles[id]);
    }
}
```

as ordinary functions in GPU kernels and thus permits a simple implementation. Without interrupts, the system has to be implemented using multiple passes with sophisticated dataflow. Simple `if` statements in Listing 7 have to be mapped to a multi-pass list split or compact. As a consequence, any change in the visualization scheme would require considerable re-factorization work.