Front. Electr. Electron. Eng. 2012, 7(1): 5–15
DOI 10.1007/s11460-012-0187-x

RESEARCH ARTICLE

Kun ZHOU

GPU parallel computing: Programming language, debugging tools and data structures

© Higher Education Press and Springer-Verlag Berlin Heidelberg 2012

Abstract With many cores driven by high memory bandwidth, today's graphics processing unit (GPU) has involved into an absolute computing workhorse. More and more scientists, researchers and software developers are using GPUs to accelerate their algorithms and applications. Developing complex programs and software on the GPU, however, is still far from easy with existing tools provided by hardware vendors. This article introduces our recent research efforts to make GPU software development much easier. Specifically, we designed BSGP, a high-level programming language for general-purpose computation on the GPU. A BSGP program looks much the same as a sequential C program, and is thus easy to read, write and maintain. Its performance on the GPU is guaranteed by a well-designed compiler that converts the program to native GPU code. We also developed an effective debugging system for BSGP programs based on the GPU interrupt, a unique feature of BSGP that allows calling CPU functions from inside GPU code. Moreover, using BSGP, we developed GPU algorithms for constructing several widely-used spatial hierarchies for high-performance graphics applications.

Keywords graphics processing unit (GPU), parallel computing, programming languages, debugging tools, data structures

1 Introduction

The graphics processing unit (GPU) was originally designed for three-dimensional (3D) graphics applications, such as video games. It is usually on a video card, which is connected to the computer through a PCI-E connection. And basically, it is used to offload 3D graphics rendering from the CPU.

Over the past few years, GPUs have evolved from a simple device with fixed functions to a complex and powerful device with highly programmable components. And modern GPUs are designed to put more transistors on data processing rather than data caching and flow control, making their floating-point capability greatly outperform that of the CPU. With many cores driven by high memory bandwidth, they are very suitable for compute-intensive, highly parallel computations. More and more scientists, researchers and software developers are using GPUs to accelerate their algorithms and applications, including physical simulations, computational finance, medical imaging, and computational biology. We refer the readers to NVIDIA's CUDA website (http://developer.nvidia.com/category/zone/cuda-zone) to get a better understanding of the various kinds of applications that can be accelerated by the GPU.

Developing complex GPU programs and software, however, is still far from easy with existing tools provided by hardware vendors. First, existing general purpose programming languages like CUDA and OpenCL are based on the stream processing model, which matches GPU's underlying architecture. While this model can supply high performance, it also makes GPU programming hard due to its low level nature. Second, it is very difficult for existing debugging tools to debug complex general purpose GPU programs due to the large number of threads and the arbitrarily flexible dataflow in the programs. And finally, it is challenging to construct some data structures like kd-trees and octrees on the GPU, which are widely used in many applications.

In this article, we introduce our recent research efforts to make GPU software development much easier by addressing the above challenges. Specifically, in Sect. 2, we describe BSGP, a high-level programming language for general-purpose computation on the GPU [1]. A BSGP program looks much the same as a sequential C program, and is thus easy to read, write and maintain. Its performance on the GPU is guaranteed by a well-

Received October 10, 2011; accepted December 7, 2011

Kun ZHOU (✉)
College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China
E-mail: kunzhou@acm.org

designed compiler that converts the program to GPU kernels. In Sect. 3, we introduce BSGP's debugging system based on the GPU interrupt, which allows calling CPU functions from inside GPU code [2]. Using BSGP, we have developed GPU algorithms for constructing several widely-used spatial hierarchies for high-performance graphics applications [3–5], which are described in Sect. 4.

2 Bulk-synchronous GPU programming

2.1 Streaming processing model and bulk synchronous parallel model

Existing general purpose programming languages for the GPU (e.g., CUDA and OpenCL) all evolved from the Brook language [6]. These languages are based on the stream processing model, in which programmers organize data into uniform streams of elements, and write kernels that are applied to all elements of input streams in parallel to produce output streams. For complex computations, programmers need to launch multiple kernels on multiple streams, and use temporary streams to pass intermediate values through kernels.

According to the analysis in Ref. [1], this stream processing model can supply high performance because it matches GPU's underlying architecture (see Fig. 1). However, it also makes GPU programming hard for several reasons. The first issue is program readability and maintenance. In order to reduce the costs of kernel launches and stream management, programmers often need to bundle multiple processes into a single kernel. This bundling is done solely based on dataflow, which means the bundled processes usually do not relate to each other in their high level functionalities. This makes optimized stream programs difficult to read. Also, when adding new functionalities to old programs, the dataflow will change. To avoid performance penalties, all affected kernels must be refactorized and re-bundled, which greatly increases the source code maintenance cost. The second issue is about the management of temporary streams. For programs with multiple kernels, intermediate values are passed through kernels by using temporary streams. In order to reduce the GPU memory consumption, each temporary stream usually has to be recycled multiple times to hold many independent values. This is quite analogous to register reuse when programming with assembly languages. The recycling is done manually by the programmer, and for complex programs with many intermediate values, this is tedious and error prone. And finally, the abstraction of parallel primitives is difficult, which makes code reuse inefficient. Many parallel primitives, like sort and scan, require multiple kernels. When such a primitive is called by a kernel, part of the primitive needs to be bundled with the caller to achieve better performance, which breaks the integrity of the primitive.

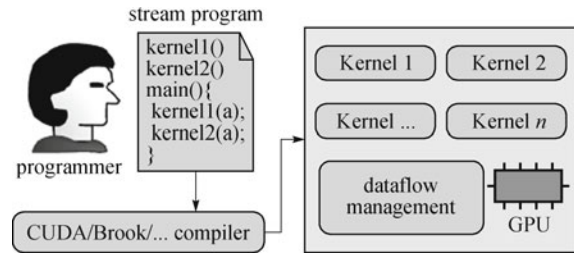


Fig. 1 Stream programming

The BSGP language is designed to solve the above problems [1]. It is based on the bulk synchronous parallel (BSP) model, which was first introduced by Valiant in 1990 [7].

Like CUDA and OpenCL, BSGP is a C-like language with several special constructs dedicated for GPU programming. A BSGP program looks much the same as a sequential C program. Programmers only need to supply *barriers* to describe parallel processing on GPUs. This is the first-class feature of BSGP that programmers use to describe global synchronization on the GPU. When the barrier is reached, execution is blocked until all threads reach the same barrier. Note that in CUDA and OpenCL, waiting for a kernel launch to terminate is the only form of global barrier. BSGP removes this awkward coupling between source code modules and global synchronization. It allows programmers to put barriers inside a GPU function. The statements between two barriers are automatically deduced as a superstep and translated into a GPU kernel by the BSGP compiler, as shown in Fig. 2. Each superstep is thus executed in parallel by a number of threads, and all supersteps are delimited by barrier synchronizations to ensure steps are executed in sequential order with respect to each other. Another advantage of BSGP is that programmers are freed from the tedious task of temporary stream management. In BSGP, data dependencies are defined implicitly because local variables are visible and shared across supersteps. The actual data flow is deduced by the compiler. And temporary streams are automatically allocated and freed by the compiler. Finally, BSGP makes the abstraction of parallel primitives simple and thus facilitates source code reuse. With BSGP, a parallel primitive such as *reduce*,

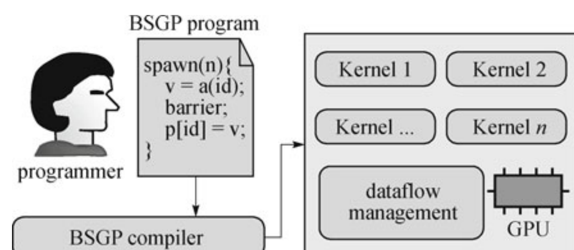


Fig. 2 BSGP system architecture [1]

scan and *sort* can be called as a whole in a single statement.

2.2 Source code example

We used a simple example in Ref. [1] to demonstrate BSGP's advantages compared to CUDA.

As shown in Fig. 3, for each vertex of a triangular mesh, the example computes a list of the vertex's one-ring neighboring triangles. For example, there are five triangles in the one-ring neighborhood of vertex V_1, and six triangles in the neighborhood of V_2. The algorithm is often used for vertex normal computation in many graphics applications.

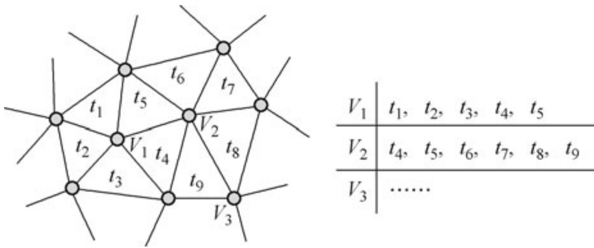


Fig. 3 Computing each vertex's one-ring neighbors

We solve the problem using a two-step algorithm. First, each triangle is triplicated and associated with its three vertices. The triplicated triangles are sorted using the associated vertex indices as the sort key. After sorting, triangles sharing the same vertex are grouped together to create a concatenated list of all vertices' neighboring triangles. Second, each sort key is then compared to its predecessor's to compute a pointer to the beginning of each vertex's list.

Figure 4 gives the BSGP implementation of the above algorithm. The `spawn` statement creates $3n$ threads on the GPU to execute the enclosed statements. `thread.sortby` is a rank adjusting primitive, which reassigns thread ranks to match the order of sort keys.

```
/*
input:
    ib: pointer to element array
    n: number of triangles
output:
    pf: concatenated neighborhood list
    hd: per-vertex list head pointer
temporary:
    owner: associated vertex of each face
*/
findFaces(int* pf, int* hd, int* ib, int n){
    spawn(n*3){
        rk = thread.rank;
        f = rk/3;            //face id
        v = ib[rk];          //vertex id
        thread.sortby(v);
        //allocate a temp list
        require
            owner = dtempnew[n]int;
        rk = thread.rank;
        pf[rk] = f;
        owner[rk] = v;
        barrier;
        if(rk==0||owner[rk-1]!=v)
            hd[v] = rk;
    }
}
```

Fig. 4 Finding neighboring triangles (BSGP code) [1]

This primitive preserves each sort key's correspondence with other data. To compare a sort key with that of a predecessor, all sort keys are stored in a temporary list `owner`. After a barrier synchronization, the predecessor's sort key is then gathered from the list and a comparison is performed to yield each vertex's head pointer. A `require` statement is used to insert the resource allocation CPU code into BSGP code. This program matches the algorithm description step by step — much like in traditional sequential programming.

Figure 5 shows the CUDA implementation of the same algorithm. The program contains three kernels: `before_sort` prepares sort key for calling the sort routine from CUDPP (http://www.gpgpu.org/developer/cudpp/), `after_sort` unpacks the sorting result and fills the concatenated list of neighboring triangles, and finally `make_head` computes head pointers. The `findFaces` function launches these kernels, calls the sorting primitive, and maintains temporary streams.

Compared with the CUDA version, it is very clear that the BSGP implementation is much easier to read, write and maintain. In the following, we compare other aspects of these two programs.

First, in CUDA, data flow is explicitly specified via parameter passing. Temporary streams are manually allocated to hold parameter values, such as the sort `key` in Fig. 5. This results in a larger code size. While in BSGP, local variables are visible across barriers. No explicit parameter passing is needed. The actual data flow is deduced by the compiler, and temporary streams are automatically allocated and freed by the compiler when appropriate.

Second, source code reuse is a serious problem in CUDA. Because kernel launch is impossible inside a kernel, a function having multiple kernels like the sort primitive, `cudppSort`, in Fig. 5, is typically reused as a CPU wrapper function. The function performs a local sorting pass and a number of global merging passes on the input stream. Under such a setting, sort key preparation and local sorting are done in two separate passes (see Fig. 6 for an illustration). Moreover, a temporary stream is allocated to pass the sort key. Note that the key preparation and local sorting can actually be bundled in a single kernel without using a temporary stream for the sort key. Separating them results in an extra kernel launch and an extra stream. This is inefficient. Of course, an experienced GPU programmer can do this bundling manually. However, this will break the integrity of the sort primitive, and programmers cannot call the function in a single statement. In contrast, BSGP allows barrier synchronization within a function and thus makes collective functions possible. For example, `thread.sortby` in

```
#include "cudpp.h"
const int szblock=256;
__global__ void
before_sort(unsigned int* key,int* ib,int n3){
    int rk=blockIdx.x*szblock+threadIdx.x;
    if(rk<n3){
        key[rk]=(ib[rk]<<16u)+rk/3;
    }
}

__global__ void
after_sort(int* pf,int* owner,unsigned int* sorted,int n3){
    int rk=blockIdx.x*szblock+threadIdx.x;
    if(rk<n3){
        int k=sorted[rk];
        pf[rk]=(k&0xffff);
        owner[rk]=(k>>16u);
    }
}

__global__ void
make_head(int* hd,int* owner,int n3){
    int rk=blockIdx.x*szblock+threadIdx.x;
    if(rk<n3){
        int v=owner[rk];
        if(rk==0||v!=owner[rk-1])
            hd[v]=rk;
    }
}

/*
interface is the same as BSGP version
temporary streams:
    key: sort keys
    sorted: sort result
    temp1: used twice for different purpose
        1. temporary stream 1 for cudppSort
        2. associated vertex of each face (owner)
    temp2: temporary stream 2 for cudppSort
*/
void findFaces(int* pf,int* hd,int* ib,int n){
    int n3=n*3;
    int ng=(n3+szblock-1)/szblock;
    unsigned int* key;
    unsigned int* sorted;
    int* temp1;
    int* temp2;
    cudaMalloc((void**)&key,n3*sizeof(unsigned int));
    cudaMalloc((void**)&sorted,n3*sizeof(unsigned int));
    cudaMalloc((void**)&temp1,n3*sizeof(int));
    cudaMalloc((void**)&temp2,n3*sizeof(int));
    before_sort<<<ng,szblock>>>(key,ib,n3);
    //call the CUDPP sort
    {
        CUDPPSortConfig sp;
        CUDPPScanConfig scanconfig;
        sp.numElements = n3;
        sp.datatype = CUDPP_UINT;
        sp.sortAlgorithm = CUDPP_SORT_RADIX;
        scanconfig.direction = CUDPP_SCAN_FORWARD;
        scanconfig.exclusivity = CUDPP_SCAN_EXCLUSIVE;
        scanconfig.maxNumElements = n3;
        scanconfig.maxNumRows = 1;
        scanconfig.datatype = CUDPP_UINT;
        scanconfig.op = CUDPP_ADD;
        cudppInitializeScan(&scanconfig);
        sp.scanConfig = &scanconfig;
        cudppSort(sorted, key, temp1, temp2, &sp, 0);
        cudppFinalizeScan(sp.scanConfig);
    }
    after_sort<<<ng,szblock>>>(pf,temp1,sorted,n3);
    make_head<<<ng,szblock>>>(hd,temp1,n3);
    cudaFree(temp2);
    cudaFree(temp1);
    cudaFree(sorted);
    cudaFree(key);
}
```

Fig. 5 Finding neighboring triangles (CUDA code) [1]

Fig. 4 is an inlined collective function containing a barrier. All code before the barrier belongs to the preceding superstep by definition. Bundling is thus achieved automatically by the compiler.

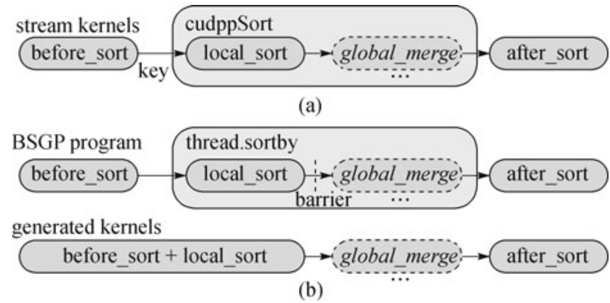


Fig. 6 Code reuse in stream model and BSP model [1]. (a) Stream model; (b) BSP model

2.3 BSGP language constructs and compiler

As aforementioned, BSGP is a C-like language with special constructs dedicated for GPU programming. In the following we briefly introduce several important BSGP constructs related to GPU programming.

- `spawn` and `barrier`: A `spawn` statement executes a block of GPU code using the total number of threads as a parameter. A barrier synchronization is indicated by a `barrier` statement. A barrier may appear directly in BSGP code or in functions inlined into BSGP code.
- `require`: A `require` statement allows the programmer to insert a block of CPU code into BSGP code. These CPU codes are responsible for resource allocation and kernel launch configuration. At run time, code inserted this way is executed before the containing superstep is launched.
- `fork` and `kill`: For applications that involve data amplification/reduction, it is desirable to create/destroy threads to improve load balancing as the application data is amplified/reduced. We provide two thread manipulation methods, `fork` and `kill`, for thread creation and destruction. These methods are extremely useful in parallel programming but are missing in existing GPU languages.
- `thread.get` and `thread.put`: BSGP supports remote variable access intrinsics for thread communication. In particular, `thread.get(r,v)` gets `v`'s value in the previous superstep from the thread with rank `r`, and `thread.put(r,p,v)` stores `v`'s value to `p` in the thread with rank `r`.
- Primitive operations: We also provide a library of parallel primitives as collective functions, such as reduce, scan and sort.

The BSGP programming model does not directly match the GPU's architecture, and we need a compiler to convert BSGP programs to efficient stream programs. There are lots of technical details in implementing the BSGP compiler. Please refer to Ref. [1] for details.

At a high level, the compiler can be divided into a front end and a back end. The front end first compiles the BSGP source code to static single assignment form

(or SSA form), then performs compiler optimizations, and finally generates kernels, kernel launching code and stream management code. The back end then translates the generated kernels to native GPU code. Currently, we generate CUDA assembly code, and call the CUDA assembler to create the binary code.

2.4 Experimental results

As described in Ref. [1], we tested BSGP's code efficiency and ease of programming using a variety of GPU applications, including a ray tracer, a particle-based fluid simulation demo, an X3D parser and an adaptive mesh tessellator. Based on these testing results, we drew the conclusion that BSGP programming is much easier and more effective than CUDA programming. In the ray tracer example, BSGP and CUDA implementations achieve similar performance and memory consumption. However, BSGP enjoys much lower source code complexity as measured in terms of code sizes and structures. In the particle example, the BSGP implementation has a clear advantage over the implementation provided in CUDA SDK in terms of code complexity. Also, while neither implementation is very well optimized, the BSGP version is about 50% faster. The X3D parser example is highly sophisticated, with 82 kernels and 19K lines of assembly code after compilation. The dataflow of this program changes constantly with the addition of new functionalities. Implementing the parser using conventional stream programming would require continuous kernel rewriting and refactorization, and has not been attempted in previous work. Our BSGP implementation of the parser, running on the GPU, is up to 15 times faster than an optimized CPU implementation. In the tessellation example, we demonstrate that our BSGP thread manipulation primitive *fork* is capable of improving the performance by a factor of 10 with little extra coding.

3 GPU interupts and debugging tools

Once programmers finish writing a GPU program, the next thing is to debug the program to make it work correctly. In 2009, we developed a dataflow-centric debugging system for BSGP programs [2]. As shown in Fig. 7, in our system, a BSGP program is first compiled by an instrumenting compiler to add dataflow recording code. This instrumented program then directly runs on the GPU and the recorded dataflow is visualized to help the user to locate bugs. Right after our debugging system was developed, NVIDIA released a new debugging tool integrated in Parallel Nsight. Compared to this debugging tool, our dataflow-centric debugging system has several unique features such as runtime memory error detection and dataflow visualization. We also allow programmers to customize their own debugging functions using GPU interrupts. Please refer to Ref. [2] for details.

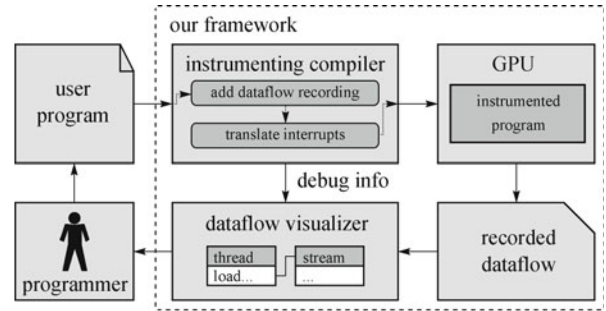


Fig. 7 BSGP debugging system architecture [2]

Dataflow recording in the debugging system requires disk I/O operations, which can only be performed in CPU functions. We therefore need a mechanism to allow calling CPU functions from inside GPU code. This is exactly what the GPU interrupt is designed for.

In the following, we first introduce the concept and syntax of GPU interrupts, then briefly describe the debugging tools provided in our system.

3.1 GPU interrupts

In existing general purpose GPU languages like 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 Fig. 8, currently there are three types of functions: `__host__`, `__device__` and `__global__`. Programmers are not allowed to call CPU functions from the GPU.

caller	CPU function	GPU function
CPU	`__host__`	`__global__`
GPU	`__interrupt__` (our extension)	`__device__`

Fig. 8 Function types in CUDA and our interrupt extension [2]

As an extension of the BSGP language, we introduced a novel function type, `__interrupt__` , which specifies a CPU function callable from the GPU. An interrupt can be called from any GPU code position, including inside a loop or a control flow instruction. Because it is a CPU function, it can call any `__global__` function inside its implementation. Interrupt calling can be even nested or recursive. As shown in Fig. 9, the concept of GPU interrupt is analogous to a CPU software interrupt. When a GPU interrupt is called, the GPU thread suspends. The CPU executes the interrupt handler, and then the GPU thread resumes after the handler returns.

Figure 10 gives an example of GPU interrupt. It is a triangle normal computation routine. `assert` is used

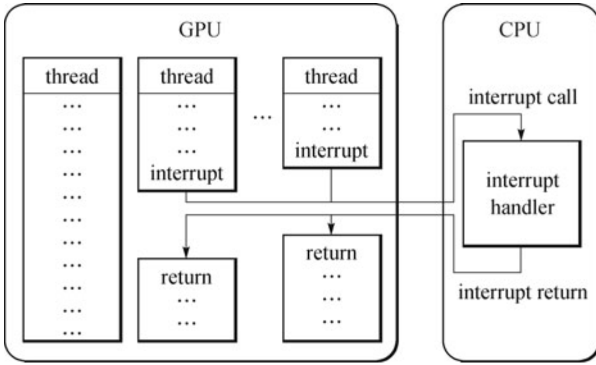


Fig. 9 GPU interrupt execution process

```
__device__ void assert(int flag);
__global__ void calcnormal(float3* N, float3* P, int n){
    int id=threadIdx.x+blockIdx.x*blockDim.x;
    if(id>=n)return;
    float3 v0=P[id*3], v1=P[id*3+1], v2=P[id*3+2];
    v1-=v0; v2-=v0;
    float3 N0=cross(v1,v2);
    assert(length(N0)>1e-5f*(dot(v1,v1)+dot(v2,v2)));
    N[id]=normalize(N0);
}

__interrupt__ void assertfail(int rank){
    int n=interrupt::size;
    int* pr=new int[n];
    cuMemcpyDtoH(pr,rank.d,n*sizeof(int));
    for(int i=0;i<n;i++)
        printf("Assert failed at thread %d\n",pr[i]);
    delete pr;
    exit(1);
}

__device__ void assert(int flag){
    if(!flag)
        assertfail(threadIdx.x+blockIdx.x*blockDim.x);
}
```

Fig. 10 Normal computation with degeneracy detection [2]

to detect degenerate triangles. It calls the `assertfail` interrupt to display error messages using `printf` when the assertion fails. As you can see, calling an interrupt is similar to calling a GPU function.

We designed a 5-step algorithm to implement GPU interrupts in the BSGP compiler. The main challenge is to minimize the interrupt processing overhead. The basic idea is to first group threads with the same interruption status together on the GPU, and then perform interrupt processing for each group instead of for each thread. Please refer to Ref. [2] for more details.

3.2 Debugging tools

Our debugging system mainly consists of three parts: 1) a few `assert`/`printf` style debugging functions; 2) run-time memory error detection, such as out-of-bound access and race conditions; 3) dataflow recording and visualization.

As mentioned in Sect. 3.1, `assert` is the simplest form of debugging. Just like its counterpart in sequential C programming, it is very useful and widely used. For example, we used more than 60 asserts in our RenderAnts project [8]. After seeing an assertion fail, most programmers would add a `printf` to dump some values around the assert. The `debug::watch` function does essentially the same on the GPU in a more fancy way. It visualizes values as colored bullets (see Fig. 11). 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.

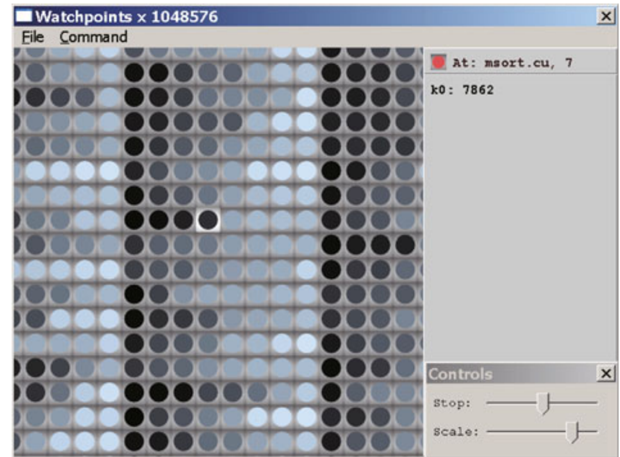


Fig. 11 Colored-bullet visualization of a variable [2]

Our 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. This kind of checks are more costly than `assert` and `debug::watch`, but they can discover bugs earlier.

Once the instrumented program terminates (normally or abnormally), the dataflow visualizer is launched to analyze the recorded dataflow. If an error is detected at runtime, 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, as illustrated in Fig. 12.

3.3 Examples

Several debugging examples are described in Ref. [2] to demonstrate the effectiveness of the debugging system, among which the image denoising example is particularly interesting as it is caused by a common bug met by every programmer.

In this example, we implemented an image denoising algorithm based on bilateral filtering. The algorithm takes a noisy image like in Fig. 13(a) as input. Bilateral filtering is performed within a 7×7 window around each pixel. The denoised image is shown in Fig. 13(b).

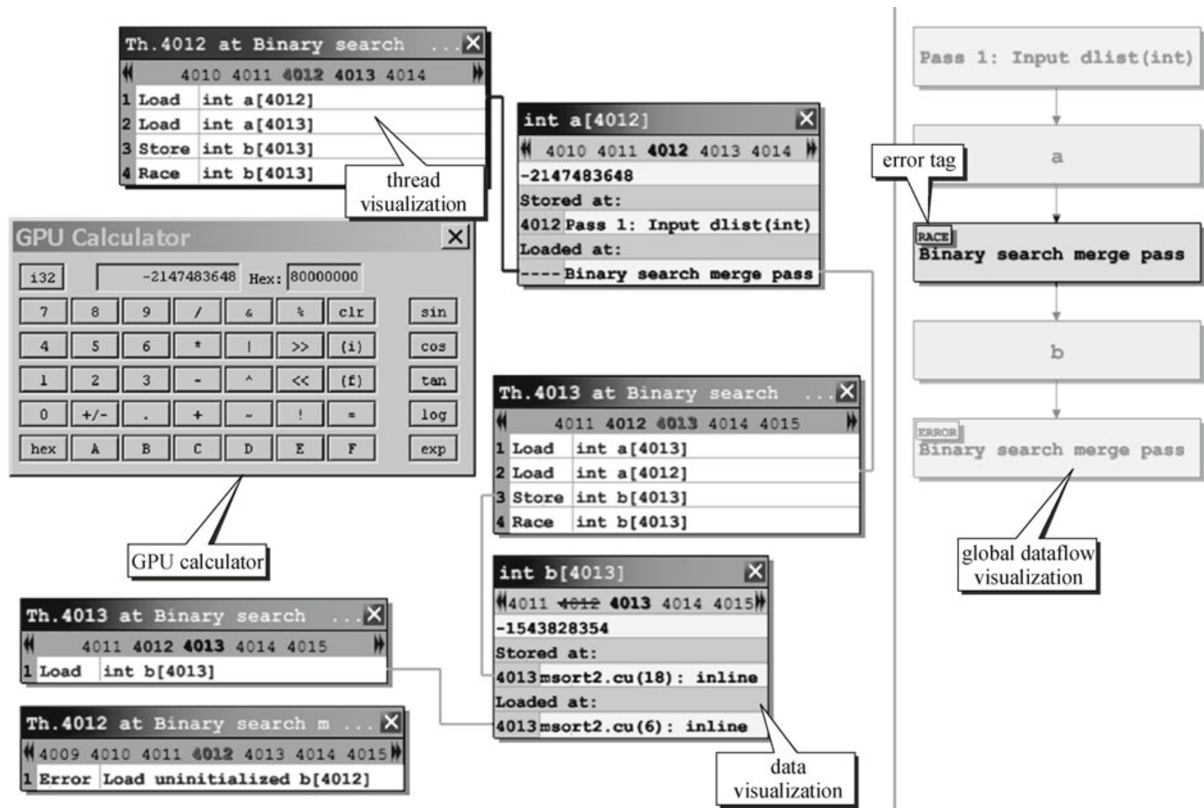


Fig. 12 Dataflow visualizer [2]

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.

We compiled the program using our instrumenting compiler and executed it. The access history of an arbitrary thread is examined as in Fig. 13(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 (the third line in Fig. 14). As a result, the program only loops over the first column of the filter window.

4 Data structures and applications

Using the BSGP language, we have developed GPU algorithms for constructing several widely used data

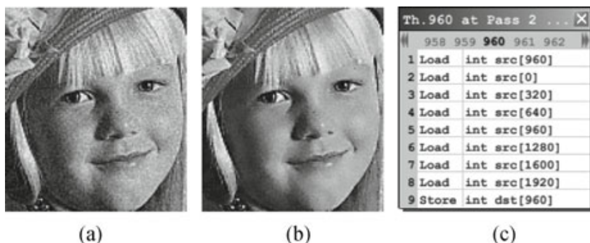


Fig. 13 Image denoising example [2]. (a) Noisy image; (b) denoised image; (c) a buggy thread

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

Fig. 14 Main loop of denoising kernel [2]

structures such as kd-trees [3,4] and octrees [5].

4.1 KD-trees

KD-tree is a well-known space-partitioning data structure for organizing points in k-dimensional space. In the 3D case, the bounding box of a scene will be split recursively until some ending condition is reached. In every split, a tree node is divided into two child nodes by a splitting plane.

Because of its fundamental importance in computer graphics applications, many fast kd-tree construction algorithms have been proposed in recent years, mostly based on the CPU (e.g., Ref. [9]). In 2008, we introduced the first kd-tree construction algorithm on the GPU [3]. The algorithm achieves real-time performance by exploiting the GPU's streaming architecture at all stages of kd-tree construction. It is significantly faster than well-optimized single-core CPU algorithms and competitive with multi-core CPU algorithms. We also demonstrated

the potential of the algorithm in applications involving dynamic scenes, including GPU ray tracing, interactive photon mapping, and point cloud modeling.

The main challenge in designing a kd-tree algorithm for the GPU is to maximally exploit the GPU's streaming architecture when parallelizing kd-tree construction, which we addressed in two ways. As illustrated in Fig. 15, we first build the tree in the top-down, breadth-first search (BFS) order. This results in a number of nodes at the same tree level, which can be processed independently. We spawn a new thread for every node to feed the GPU with a large number of nodes. Although this strategy works well for deep tree levels, as you can imagine, it is very inefficient for upper tree levels. Because at upper tree levels, the number of nodes at the same tree level is small. Fortunately, each of these nodes contains a large number of geometric primitives, which we call large nodes. We therefore propose to parallelize the computation over all primitives contained in large nodes.

Figure 16 shows the pseudo code of the algorithm, assuming the input is a set of triangles. After an initialization step, the algorithm builds the tree in a BFS manner,

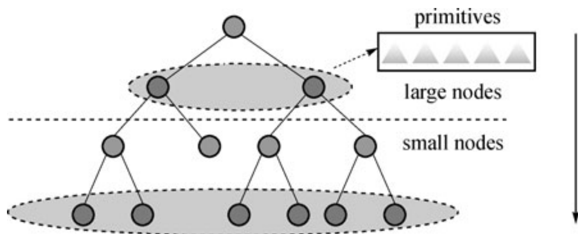


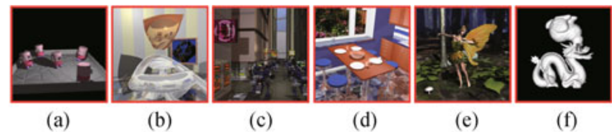
Fig. 15 Illustration of our GPU kd-tree construction

```
procedure BuildTree(triangles:list)
begin
  // initialization stage
  nodelist ← new list
  activelist ← new list
  smalllist ← new list
  nextlist ← new list
  Create rootnode
  activelist.add(rootnode)
  for each input triangle t in parallel
    Compute AABB for triangle t
  // large node stage
  while not activelist.empty()
    nodelist.append(activelist)
    nextlist.clear()
    ProcessLargeNodes(activelist, smalllist, nextlist)
    Swap nextlist and activelist
  // small node stage
  PreprocessSmallNodes(smalllist)
  activelist ← smalllist
  while not activelist.empty()
    nodelist.append(activelist)
    nextlist.clear()
    ProcessSmallNodes(activelist, nextlist)
    Swap nextlist and activelist
  // kd-tree output stage
  PreorderTraversal(nodelist)
end
```

Fig. 16 Pseudo code of our GPU kd-tree construction [3]

for both large nodes and small nodes. Finally, all nodes of the tree are reorganized and stored. In the initialization stage, global memory is allocated for tree construction and the root node is created. Additionally, a streaming step is performed to compute the axis aligned bounding box (AABB) for each input triangle. In our current implementation, the user-specified threshold for large/small node is set as $T = 64$. More algorithm details about the large/small node stages and tree output stages can be found in Ref. [3].

Based on our real-time kd-tree construction, we implemented a GPU ray tracer for arbitrary dynamic scenes. The ray tracer achieves interactive rates with shadow and multi-bounce reflection/refraction. As shown in Fig. 17, we compared our algorithm with a well-optimized off-line CPU algorithm on several publicly available scenes. Our kd-tree construction is about 10 times faster for all scenes. In all cases, our trees offer similar ray tracing performance, which proves the high quality of our trees in practical applications.



scene	off-line CPU builder		our GPU builder	
	T_{tree}	T_{trace}	T_{tree}	T_{trace}
(a)	0.085 s	0.022 s	0.012 s	0.018 s
(b)	0.108 s	0.109 s	0.017 s	0.108 s
(c)	0.487 s	0.165 s	0.039 s	0.157 s
(d)	0.559 s	0.226 s	0.053 s	0.207 s
(e)	1.226 s	0.087 s	0.077 s	0.078 s
(f)	1.354 s	0.027 s	0.093 s	0.025 s

Fig. 17 Test scenes for kd-tree construction and timings [3]

We also implemented GPU photon mapping, in which photon tracing, photon kd-tree construction and nearest photon query are all performed on the GPU on the fly. Combined with our GPU ray tracer, the photon mapping is capable of rendering shadows, reflection/refraction, as well as realistic caustics for dynamic scenes and lighting at interactive rates on a single PC (see Fig. 18). Such performance has not been achieved in previous work.

Our real-time kd-tree construction can also be used for dynamic point clouds to accelerate nearest neighbor queries. The queried neighbors are used for estimating local sampling densities and calculating the normals (see

Fig. 18 Caustics rendering using photon mapping [3]. (a) A metal ring; (b) a glass of champagne

Fig. 19 Sampling density and normal estimation [3]

Fig. 19).

4.2 Memory scalable kd-trees

One limitation of the GPU kd-tree construction algorithm described above is that its BFS construction order consumes excessive GPU memory, which becomes a serious issue for interactive applications involving very complex models. Current GPUs have a different memory architecture than CPUs. The on-board memory on GPUs is limited to a few GBs. Moreover, GPUs have high memory bandwidth, much smaller per-thread caches and GPU's memory limitation cannot be virtualized by on-demand paging. It is thus important to design GPU-based algorithms that can cope with these memory architecture characteristics of GPUs for interactive applications.

In Ref. [4], we proposed to use the partial breadth-first search (PBFS) construction order to build kd-trees. The key idea is to make proper tradeoffs between memory consumption and level of parallelism to control memory consumption while maximizing performance. As illustrated in Fig. 20(b), the algorithm uses PBFS to automatically adapt the number of nodes being processed simultaneously based on available memory and thus allows the peak memory consumption to be controlled without swapping any data out of the GPU. Compared to the exhaustive BFS, the PBFS only processes part of the nodes at a time. When some trunks of the tree are completely constructed, the corresponding memory is released so that we can process the remaining nodes.

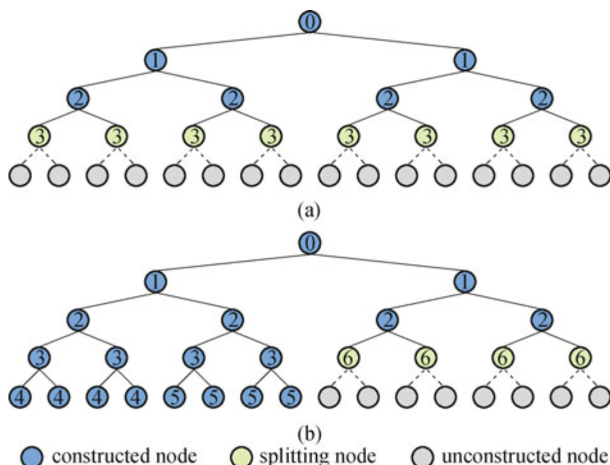


Fig. 20 Different kd-tree construction orders. The number in each node corresponds to the iteration it is created in Ref. [4]. (a) BFS kd-tree construction; (b) PBFS kd-tree construction

As shown in Fig. 21, instead of executing the large node stage and the small node stage sequentially as in the original GPU kd-tree construction algorithm, we alternate between large node and small node construction. Our observation is that it is unnecessary to wait until all small roots are generated because the small roots are continuously generated along the whole process of the large node stage. At any time we find the small roots are too many to be split simultaneously, we launch a small node stage to complete the construction of as many small roots as available memory allows. After this small node stage, all temporary data associated with the completed nodes are discarded. We then go back to the large node stage to continue generating small roots.

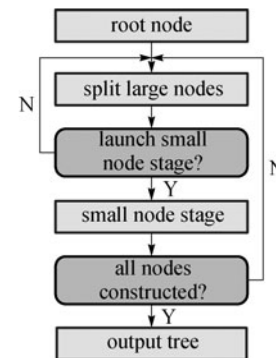


Fig. 21 Our alternating kd-tree construction pipeline [4]

According to experiments, this PBFS-based algorithm can handle scenes nearly an order of magnitude larger than the BFS-based GPU algorithm. Figure 22 shows a large animated scenes, which begins with 560K triangles and ends with about 7M triangles. Figure 23 demonstrates how our performance and memory consumption changes with respect to the scene complexity. As

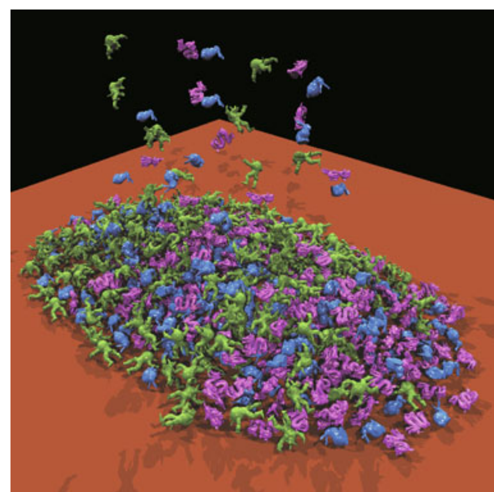


Fig. 22 An animated scene of 612 falling objects [4]

illustrated in Fig. 23(a), the memory peak of our construction algorithm exhibits a two-phase behavior. When the scene is small and can fit in available memory, the peak grows rapidly at a roughly linear speed. As the scene becomes larger, our PBFS scheme takes effect, and the memory peak oscillates at a relatively steady level. As the scene size increases further, the memory consumed by the scene geometry increases, and the memory available for kd-tree construction decreases. Our construction algorithm thus reduces its memory peak accordingly. Regardless of the memory peak behavior, our construction time grows linearly with the number of triangles, as shown in Fig. 23(b). The PBFS scheme successfully controls peak memory consumption at minimal performance tradeoff.

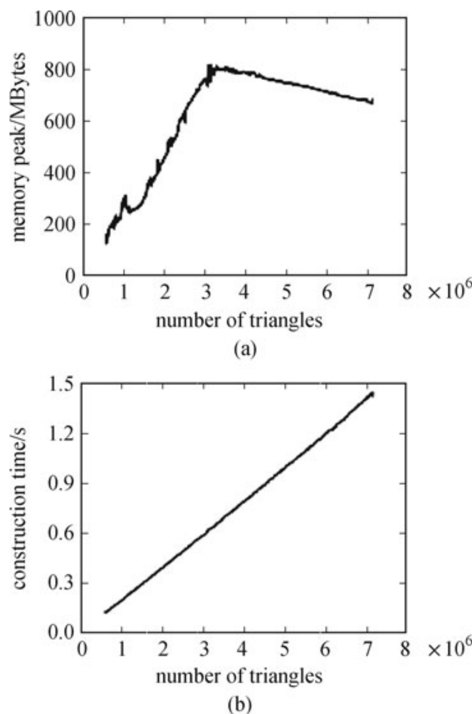


Fig. 23 Memory peak and performance of our construction algorithm for the animated scene shown in Fig. 22 [4]. (a) Memory peak; (b) construction performance

4.3 Octrees

Octree is another important data structure in computer graphics applications. Following our kd-tree algorithms, we proposed a BFS-based method to construct an octree for a set of 3D points on the GPU, and used it for parallel surface reconstruction [5].

The algorithm builds octrees in real-time and uses BFS order traversals to exploit the fine-grained parallelism of the GPU. The constructed octrees contain the information necessary for GPU surface reconstruction. In particular, the octree data structure provides fast access to tree nodes, faces, edges and vertices, as well as the neighborhood information of each tree node. According to experiments, our GPU algorithm can achieve five surface reconstructions per second for a set of 500K points, which is over two orders of magnitude faster than previous CPU algorithms (e.g., Ref. [10]).

As shown in Fig. 24, we can use the algorithm to reconstruct fluid surfaces for real-time fluid simulation. For 32K particles, the simulation procedure alone runs at around 240 frames per second. With our fluid surface reconstruction at octree depth 6, the whole program runs at about 50 frames per second. The reconstructed surface is directly shaded on the GPU. The program also allows users to interact with the fluid.

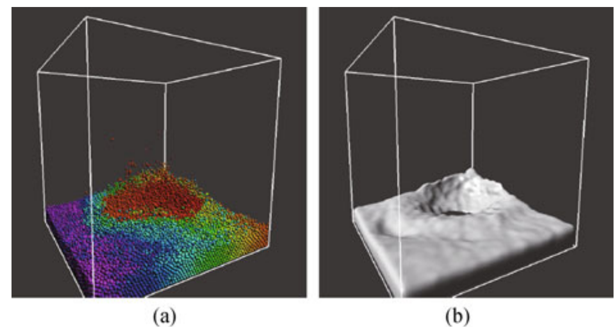


Fig. 24 Real-time fluid surface reconstruction [5]. (a) Particles; (b) reconstructed surface

5 Conclusions

This article introduced our recent work on GPU parallel computing to simplify GPU software development. Specifically, we describe BSGP, a high-level programming language for general-purpose computation on the GPU [1], BSGP's debugging system based on the GPU interrupt [2], and some widely-used spatial hierarchies for high-performance graphics applications including kd-tress [3,4] and octrees [5].

For future work, we plan to investigate other high-level GPU programming languages and tools, such as object-oriented GPU programming languages. We are also interested in developing domain specific languages and data structures for many non-graphics applications.

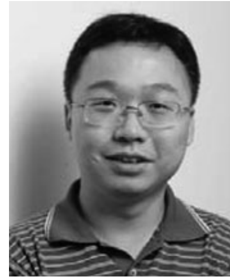
Acknowledgements This research was partially supported by the National Natural Science Foundation of China (Grant No. 60825201).

References

1. Hou Q, Zhou K, Guo B. BSGP: Bulk-synchronous GPU programming. ACM Transactions on Graphics, 2008, 27(3): Article 19, 1–13
2. Hou Q, Zhou K, Guo B. Debugging GPU stream programs through automatic dataflow recording and visualization. ACM Transactions on Graphics, 2009, 28(5): Article 153, 1–11
3. Zhou K, Hou Q, Wang R, Guo B. Real-time kd-tree construc-

tion on graphics hardware. ACM Transactions on Graphics, 2008, 27(5): Article 126, 1–11

4. Hou Q, Sun X, Zhou K, Lauterbach C, Manocha D. Memory-scalable GPU spatial hierarchy construction. IEEE Transactions on Visualization and Computer Graphics, 2011, 17(4): 466–474
5. Zhou K, Gong M, Huang X, Guo B. Data-parallel octrees for surface reconstruction. IEEE Transactions on Visualization and Computer Graphics, 2011, 17(5): 669–681
6. Buck I, Foley T, Horn D R, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream computing on graphics hardware. ACM Transactions on Graphics, 2004, 23(3): 777–786
7. Valiant L G. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103–111
8. Zhou K, Hou Q, Ren Z, Gong M, Sun X, Guo B. RenderAnts: Interactive Reyes rendering on GPUs. ACM Transactions on Graphics, 2009, 28(5): Article 155, 1–11
9. Shevtsov M, Soupikov A, Kapustin A. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. Computer Graphics Forum, 2007, 26(3): 395–404
10. Kazhdan M, Bolitho M, Hoppe H. Poisson surface reconstruction. In: Proceedings of the Fourth ACM/Eurographics Symposium on Geometry Processing. 2006, 61–70



Kun ZHOU is currently a Cheung Kong Professor of the College of Computer Science and Technology at Zhejiang University. He is also a member of the State Key Laboratory of CAD & CG, where he leads the Graphics and Parallel Systems Lab. Prior to joining Zhejiang University in 2008, he was a Leader Researcher of the Internet Graphics Group at Microsoft Research Asia. He received his BS and PhD degrees in computer science from Zhejiang University in 1997 and 2002, respectively. He works in the areas of computer graphics and GPU parallel computing, and has published more than 30 ACM SIGGRAPH (TOG) papers and held over 20 patents. He is on the editorial board of ACM Transactions on Graphics, The Visual Computer, and the Frontiers of Computer Science. He was named one of the world's top 35 young innovators by MIT Technology Review in 2011.