BSGP: Bulk-Synchronous GPU Programming

Qiming Hou[†] Kun Zhou[‡] Baining Guo[† ‡]

[†] Tsinghua University [‡] Microsoft Research Asia

Abstract

We present BSGP, a new programming language for general purpose computation on the GPU. A BSGP program looks much the same as a sequential C program. Programmers only need to supply a bare minimum of extra information to describe parallel processing on GPUs. As a result, BSGP programs are easy to read, write, and maintain. Moreover, the ease of programming does not come at the cost of performance. A well-designed BSGP compiler converts BSGP programs to kernels and combines them using optimally allocated temporary streams. In our benchmark, BSGP programs achieve similar or better performance than well-optimized CUDA programs, while the source code complexity and programming time are significantly reduced. To test BSGP's code efficiency and ease of programming, we implemented a variety of GPU applications, including a highly sophisticated X3D parser that would be extremely difficult to develop with existing GPU programming languages.

Keywords: programable graphics hardware, stream processing, bulk synchronous parallel programming, thread manipulation

1 Introduction

The recent advent of commodity graphics hardware offers formidable raw processing power at a moderate cost. However, programming the GPU for general purpose computation is by no means easy. Existing general purpose programming languages for the GPU are based on the stream processing model as GPUs are stream processors. These languages include Brook [Buck et al. 2004], Sh [McCool et al. 2002], and CUDA [NVIDIA 2007]. Stream processing is a data centric model, in which data is organized into homogeneous streams of elements. Individual functions called *kernels* are applied to all elements of input streams in parallel to yield output streams. Complex computation is achieved by launching multiple kernels on multiple streams. This stream/kernel abstraction explicitly exposes the underlying data dependencies.

The stream processing model, while supplying high performance, makes general purpose GPU programming hard for several reasons. First, program readability and maintenance is a big issue. In order to reduce the costs of kernel launches and stream management, multiple processes have to be bundled into a single kernel so that the numbers of intermediate streams and kernel launches are minimized. This bundling is done solely based on dataflow; the bundled processes usually do not relate to each other in their high level functionalities. As a result, an optimized stream program is very difficult to read. Also, when new functionalities are added, the dataflow changes accordingly. To avoid performance penalties, all affected kernels must be refactorized to optimize for the new dataflow, and this incurs a considerable source code maintenance cost. Second, management of temporary streams is a serious headache for programmers. For programs with multiple kernels, intermediate values are passed through kernels by using temporary streams. For optimal memory usage, each stream usually has to be recycled multiple times to hold many independent values. This recycling is done manually by the programmer, and for sophisticated programs with many intermediate values, this is a tedious and error prone process. Finally, the abstraction of parallel primitives is difficult, which hinders code reuse. Specifically, many parallel primitives (e.g., *scan* [Sengupta et al. 2007]) require multiple kernel launches. When such a primitive is called by a kernel, part of the primitive needs to be bundled with the caller to reduce intermediate stream sizes and kernel launch costs. The result is a primitive with broken integrity, which makes abstraction of the primitive difficult. Because of the above problems, it is extremely difficult to write even moderately complex general purpose programs, such as the X3D parser shown later in this paper, using today's GPU programming languages.

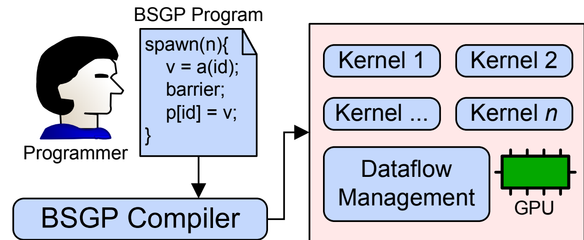


Figure 1: *BSGP system architecture. The programmer writes a single compact BSGP program which looks much the same as a sequential C program. The BSGP compiler automatically translates the BSGP program into stream kernels and generates management code for GPU execution.*

In this paper we introduce BSGP (bulk synchronous GPU programming), a new programming language for general purpose computation on the GPU. The most important advantage of BSGP is that it is easy to read, write, and maintain. BSGP is based on the BSP (bulk synchronous parallel) model [Valiant 1990]. Like any BSP program, a BSGP program looks much the same as a sequential C program. Programmers only need to supply a bare minimum of extra information, *the barriers*, to describe parallel processing on GPUs. 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. 1. 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 chore of temporary stream management. In a BSGP program, data dependencies are defined implicitly because local variables are visible and shared across supersteps. Finally, BSGP makes the abstraction of parallel primitives simple and thus facilitates source code reuse. With BSGP, a parallel primitive such as *reduce*, *scan* and *sort* can be called as a whole

[†] This work was done while Qiming Hou was an intern at MSRA.

[‡] E-mail: kunzhou@acm.org, bainguo@microsoft.com

in a single statement. Listing 1 and Listing 2 compare BSGP and CUDA programs implementing the same algorithm.

The BSGP programming model does not directly match the GPU's stream processing architecture, and we need a compiler to convert BSGP programs to efficient stream programs. To build such a compiler we need to address two challenging issues. The first is the implementation of barrier synchronization. While barriers can be directly implemented using hardware synchronization for coarse-grained parallel architectures [Hill et al. 1998], this is not possible in a stream environment. In the GPU's stream environment, it is common to create orders of magnitude more threads than can be executed simultaneously on physical processing units. Threads are dynamically distributed to available processing units for execution. Resources like registers for holding thread contexts are recycled upon completion. Synchronization of physical processing units does not affect non-executing threads, and is thus not equivalent to a (logic) barrier of threads. Waiting for all threads to finish could serve as a valid barrier, but it would also cause thread contexts to be destructed. To address this issue, we designed the compiler such that it automatically adds context saving code to make the barrier conform to BSGP semantics, as discussed in Section 4.2.

The second issue is how to generate efficient stream code. Since local variables are visible and shared across the supersteps in a BSGP program, the compiler needs to analyze the data dependencies between supersteps and automatically allocate temporary streams to save local variable values at the end of a superstep and pass these temporary streams to the subsequent supersteps. How to minimize the total number of temporary streams is critical for the efficient use of the limited video memory available on the GPU. Our solution is based on a graph optimization scheme described in Section 4.3. Thanks to the sequential organization of supersteps, we can obtain the optimal solution in polynomial time. Compared to the manual management of temporary streams in existing GPU programming languages such as CUDA, our optimization scheme can achieve similarly efficient usage of the video memory.

Additional features of BSGP include:

- Thread manipulation emulation. We transparently emulate two thread manipulation features, thread creation and destruction, with operations *fork* and *kill*, which are extremely useful in parallel programming but are missing in existing GPU programming languages.
- Thread communication intrinsics. BSGP allows remote variable access intrinsics for efficient communications between threads.
- Collective primitive operations. BSGP features a library of easy-to-use collective primitive operations including *reduce*, *scan* and *sort* (see Appendix A).

To test BSGP's code efficiency and ease of programming, we implemented a variety of GPU applications. Our experience indicates that BSGP programming is much easier and more effective than CUDA programming. To demonstrate the potential of BSGP, we present several BSGP source code examples, including a ray tracer, a particle-based fluid simulator, an X3D parser and an adaptive mesh tessellator. 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 fluid simulation 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 from [Parisi 2003]. 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.

2 Related Work

There have been a lot of works on programable graphics systems and programming languages. We refer the reader to [Buck et al. 2004] for a concise overview. In the following, we will only review the most closely related references.

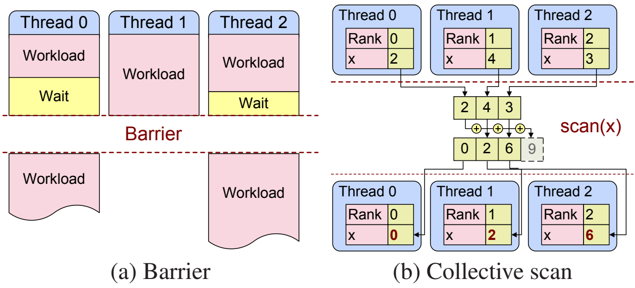
An earlier work [Peercy et al. 2000] presented a programmable shading system on top of early OpenGL's rendering pipeline. It abstracts OpenGL as a virtual SIMD processor with each render pass as an instruction. Shaders in a high level language are then compiled to this virtual architecture. Our BSGP is analogous to this work in that both implement a higher level programming model on a low level architecture through compilation. However, while [Peercy et al. 2000] targets programmable shading on the early OpenGL pipeline, our BSGP is designed for general purpose computation on modern GPUs.

Several high level programming languages have been developed for GPU programming. Shading languages including HLSL, Cg [Mark et al. 2003], and GLSL are graphics oriented and meant to be used with graphics APIs (OpenGL, DirectX, etc.). Applications written in these languages must execute explicit graphics API calls to organize data into streams and launch kernels. Computation is expressed as a sequence of shading operations on graphics primitives, not kernels acting on streams [Buck et al. 2004].

Brook [Buck et al. 2004] virtualizes the graphics pipeline and abstracts GPUs as general stream processors. By doing so it enables general purpose programming of the GPU without graphics API calls. CUDA [NVIDIA 2007] goes a step further to support advanced features like scattering and local communication in hardware, leading to a considerably more flexible environment. Both languages are based on the stream model and require explicit handling of data dependencies.

Sh [McCool and du Toit 2004] provides a meta-programming library operating on stream kernels. With Sh, kernels may be dynamically constructed and invoked using a C++ syntax. Sh also offers an algebra [McCool et al. 2004] for manipulating kernels as first class objects, enabling generation of new kernels by combining existing modules. The meta-programming framework of Sh and BSGP is concerned with different aspects of GPU programming: BSGP is for GPU programming based on the BSP model, whereas meta-programming is a general technique applicable to any programming model. Sh simplifies kernel writing while ignoring inter-kernel data dependencies. In contrast, BSGP abstracts data dependencies while ignoring module manipulation.

Accelerator [Tarditi et al. 2006] implements a data parallel array library on the GPU. Although data dependencies are hidden, the library is limited to a set of predefined parallel operators such as element-wise arithmetic, reduction and transformation. Even with an optimizing compiler, it would be extremely difficult to achieve a similar level of flexibility and efficiency as provided by a stream architecture. For instance, a KD-tree accelerated ray tracer can be implemented with a single stream kernel [Horn et al. 2007; Popov et al. 2007], but it would be very difficult to map the same ray tracer



(a) Barrier (b) Collective scan

Figure 2: *SPMD constructs* - barrier *and* collective operation.

to data parallel arrays without significant loss of performance.

Programmable graphics hardware has long been used for procedural texturing and shading in graphics applications [Olano and Lastra 1998; Peercy et al. 2000; Proudfoot et al. 2001]. Recently there has been much work on general purpose computation using GPUs (GPGPU). Examples include common primitive operations such as linear algebra, FFT [NVIDIA 2007], scan [Sengupta et al. 2007] and sorting [Gress and Zachmann 2006; Harris et al. 2007], as well as application-specific modules for image processing [NVIDIA 2007] and ray tracing [Foley and Sugerman 2005; Horn et al. 2007; Popov et al. 2007]. In most of these examples, the algorithms perform similar operations on a large number of independent elements and are thus easy to map to the stream architecture. For more complicated applications, stream implementation requires multiple kernels and intermediate streams. The design of such streams and kernels is extremely difficult for even moderately complex applications.

The BSP model provides a simple and practical framework for parallel computing in general [Valiant 1990]. The fundamental properties of BSP include easy programming, independence of target architectures, and predictable performance [Skillicorn et al. 1997]. In the context of coarse grain parallelism, the BSP model has demonstrated simpler programming and predictable performance on a variety of architectures [Hill et al. 1998]. As far as we know, BSGP is the first GPU programming language based on BSP.

Comparing to alternate models in traditional parallel computing (e.g., PRAM [Fortune and Wyllie 1978]), the BSP model has two distinguishing properties: disallowing peer-to-peer communication except at barriers and ignoring inter-processor data locality. Both properties match stream processing well, in which arbitrary interthread communication is infeasible within a kernel and persistent local storage does not physically exist.

3 BSGP and Stream Processing

In this section we illustrate the difference between BSGP and stream processing using a simple source code example. We start by explaining a few basic concepts. BSGP and stream processing are different forms of SPMD (single program multiple data) processing, in which a number of threads execute the same program in parallel. The total number of threads is called thread $size$. Each thread is given a $rank$, a unique integer from 0 to $size - 1$, which distinguishes it from other threads.

Barrier A *barrier* is a form of synchronization in SPMD programming. When a barrier is reached, execution is blocked until all threads reach the same barrier (see Fig. 2(a)). In stream processing, waiting for a kernel launch to terminate is the only form of barrier. Although CUDA-capable hardware supports local synchronization, a barrier of all threads still cannot be achieved *within* a kernel in general. Note that a kernel launch is traditionally called a pass in GPU programming.

Listing 1 Find neighboring triangles (BSGP version)

```
/*
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;
    }
}
```

Collective Operation A *collective operation* is an operation that has to be performed simultaneously by all threads. In SPMD programming, a collective operation is syntactically similar to an ordinary sequential operation except it operates on all threads semantically. The input of one thread may affect the output of other threads. For example, a collective prefix sum may be defined as `scan(x)`, and `x`'s values in all threads are collected to form a vector. After a barrier synchronization, the prefix sum is computed using the vector. The result is then redistributed to each thread's `x`, and the code execution continues (see Fig. 2(b)). Typical collective operations require barriers internally and thus are rare in stream programming.

3.1 Source Code Example

The source code example solves the following problem: Given a triangle mesh's connectivity, compute a list of the one-ring neighboring triangles for each vertex. The mesh contains m vertices and n triangles. Connectivity is given as an array of $3n$ integers ranging from 0 to $m - 1$, with each three consecutive integers representing the three vertex indices of a triangle. The BSGP source code for solving this problem is used in the X3D parser example in the results section for vertex normal computation.

Sorting Algorithm We solve the above problem using the following sorting algorithm: 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. Each sort key is then compared to its predecessor's to compute a pointer to the beginning of each vertex's list.

Listing 1 is an implementation of the above sorting algorithm using BSGP. 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 (see Appendix A for details). 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. This program matches the algo-

rithm description step by step – much like in traditional sequential programming.

Listing 2 Find neighboring triangles (CUDA version)

```
#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);
}
```

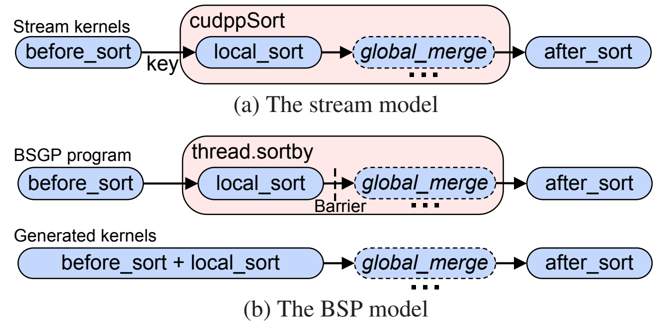


Figure 3: *Source code reuse in the stream model and the BSP model. For simplicity, we omit the internals of global merge passes and treat it as a single function. (a) In Listing 2, sort is called through CPU wrapper* `cudppSort`*. Three kernels and a temporary stream* `key` *are required; (b) In Listing 1, after inline expansion,* `local_sort` *is bundled with the preceding code in* `findFaces`*. Two kernels are generated and* `key` *is unnecessary.*

Listing 2 exhibits a CUDA (stream processing) implementation of the same algorithm, written using a sort routine from CUDPP [Harris et al. 2007]. Due to the lack of a more flexible sort, triangle and vertex IDs are packed together into the sort key. The program contains three kernels: `before_sort` prepares sort key for calling 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, the BSGP implementation is much easier to read, write and maintain. In the following, we compare other aspects of these two programs. Note that both programs are for demonstrating programming styles and neither is optimized for performance.

3.2 Explicit vs. Implicit Data Flow

The CUDA source code in Listing 2 is divided into several kernels. Data flow is explicitly specified via parameter passing, and temporary streams are allocated to hold parameter values, resulting in a larger code size. On the other hand, the BSGP version is written as a single compact procedure through the use of barriers and collective operations. From the programmer's perspective, local variables are visible across barriers, and no explicit parameter passing is needed. The actual data flow is deduced by the compiler when the stream code is generated. Temporary streams are automatically allocated and freed by the compiler when appropriate.

3.3 Efficient Code Reuse

As discussed in [Sengupta et al. 2007], source code reuse has been a serious problem in traditional GPU development. In a stream environment, a method that consists of multiple kernels like sorting is typically reused as a CPU wrapper function, since kernel launch is impossible inside another kernel. One such example is the `cudppSort` in Listing 2. It 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, as illustrated in Fig. 3(a). A temporary stream is allocated to pass the sort key.

Note that the key preparation `before_sort` and local sorting `local_sort` 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.

Although it is possible to do the bundling manually, doing so requires `local_sort` to be separated from the `cudppSort` method. This violates information hiding and considerably hinders source code reuse.

The BSP model allows barrier synchronization within a program, making collective functions possible. In an inlined collective function containing barriers, all code before the first barrier belongs to the preceding superstep by definition. The same may be applied to code after the last barrier. Bundling is thus achieved automatically as in Fig. 3(b).

3.4 Source Code Maintenance

To improve the performance of stream programs, programmers usually strive to manually bundle source code as tightly as possible in a single kernel even though these programs are semantically unrelated. For example, in Fig. 3(a), `before_sort` and `local_sort` can be bundled in a single kernel. However, such manual bundling will cause maintenance issues. For example, adding a function call in a stream program would require the surrounding kernel to be split into two. Rescheduling function calls and computation in other kernels requires all affected kernels to be refactorized to reflect the new data flow. Removing calls requires surrounding kernels to be merged to avoid performance degradation. Additional stream management code also needs to be updated manually throughout the above process. These difficulties considerably hinder source code maintenance.

With BSGP, multi-superstep algorithms may be abstracted as collective functions. Being syntactically identical, a collective function call such as `thread.sortby(x)` can be manipulated just like an ordinary function call. All necessary modifications to the final stream program are automatically made by the compiler.

4 Compiling BSGP Programs

In this section we describe the compilation of a BSGP program into stream processing code.

4.1 Compiler Design Issues

As mentioned, first we need to implement the barrier synchronization. The technical issue we need to resolve is that the stream processing model completely decouples physical processing units and logical threads. This is typically implemented in hardware without software control. Unlike software thread scheduling in traditional parallel computing environments like CHARM++ [Kale and Krishnan 1993], not all threads are executed simultaneously. Instead, threads are dynamically distributed to available processing units. Resources (e.g., registers) for holding thread contexts are recycled upon thread completion. For these reasons persistent logical thread contexts do not physically exist and thus cannot be taken for granted as in previous BSP implementations (e.g., [Hill et al. 1998]). To address this issue, we design the compiler such that it generates additional context saving code when compiling a BSGP program for stream processors.

For better performance, the context saving code produced by the compiler should be comparable or better than hand-written temporary stream management programs. Observe that the time needed to load and store variables mostly depends on the memory bandwidth and is not much affected by code generation. Based on this observation, our BSGP compiler focuses on minimizing the total amount of stored values, where a *value* means the evaluation result of an assignment's right hand side. We use the following strategy: a value is saved if and only if it is used in supersteps following its defining superstep. The BSGP compiler also carries out several important optimizations such as dead code elimination and conditional constant propagation to further reduce saved values.

Another technical issue is to minimize peak memory consumption. The naive practice of simply allocating one stream for each stored value would quickly exhaust the memory for any problem size of practical interest. The total number of allocated temporary streams has to be minimized under two constraints: 1) one temporary stream has to be allocated for each saved value; 2) values that may be used simultaneously cannot be assigned the same stream. We deal with this problem in a way analogous to the register allocation problem in compiler theory, with temporary streams being registers. Since supersteps are organized sequentially, we are able to compute the optimal solution in polynomial time using graph optimization [Ford and Fulkerson 1962].

A final challenge is about locality. Modern GPUs heavily rely on locality to achieve high performance. Severe performance penalties apply if threads with neighboring physical ranks perform heterogeneous tasks, e.g., accessing non-coherent memory addresses. Therefore, it is desirable to adjust thread ranks so as to match physical locality with algorithmic locality. We extend the original BSP model by introducing rank reassigning barriers, `barrier(RANK_REASSIGNED)`. Since physical thread ranks cannot be changed within a kernel, a logical rank reassignment is performed by shuffling stored thread contexts at a barrier.

4.2 Compilation Algorithm

A BSGP program is translated to a stream program through the following steps.

1. Inline all calls to functions containing barriers.
2. Perform optimizations to reduce data dependencies.
3. Separate CPU code and GPU code. Generate kernels and kernel launching code.
4. Convert references to CPU variables to kernel parameters.
5. Find all values that need to be saved, i.e., values used outside the defining superstep.
6. Generate code to save and load the values found in Step 5.
7. Generate temporary stream allocations.

Listing 3 is the pseudo code of the BSGP program after executing Step 1 on the program in Listing 1, i.e., expanding the inline function `thread.sortby`, which contains two supersteps and uses `barrier(RANK_REASSIGNED)` to reassign thread ranks.

Listing 4 is the pseudo code of the final generated stream program. In the following, we use Listing 3 as an example to illustrate the above compiling steps.

Step 2 performs classical compiler optimizations on the BSGP program to prevent unused variables or variables holding constant values from being saved and thus reduces context saving costs. One scheme we used is that of constant propagation from [Wegman and Zadeck 1991], with which constant values are propagated through assignments. In a BSGP program, `thread.rank` is a constant value in supersteps where it is not reassigned. Therefore, in Listing 3, `rk1` can be replaced by `thread.rank`, preventing it from being unnecessarily saved to memory.

Dead code elimination [Cytron et al. 1991] is another scheme we employed. It eliminates all source code not contributing to the program's output. Cross-superstep data dependencies in dead code may be eliminated using this optimization.

In Step 3, the BSGP program is split into a sequential set of supersteps according to barriers. For each superstep, a kernel is created that contains the code in this superstep. In the spawning func-

Listing 3 Extended version of Listing 1 by expanding the inline function `thread.sortby`.

```
findFaces(int* pf, int* hd, int* ib, int n){
    spawn(n*3){
        //superstep 1
        rk0 = thread.rank;
        f = rk0/3; v = ib[rk0];
        //BEGIN OF thread.sortby
        //allocate an internal temporary stream
        require
            sorted_id = dtempnew[thread.size]int;
        local_sort(sorted_id, v);
        barrier(RANK_REASSIGNED);
        require
            global_merge(sorted_id);
        //superstep 2
        //internal implementation of rank reassigning
        thread.oldrank = sorted_id[thread.rank];
        //END OF thread.sortby
        //allocate a temp list
        require
            owner = dtempnew[n]int;
        rk1 = thread.rank;
        pf[rk1] = f;
        owner[rk1] = v;
        barrier;
        //superstep 3
        if(rk1==0||owner[rk1-1]!=v)
            hd[v] = rk1;
    }
}
```

Value	f	v	rk0	rk1
Definition Step	1	1	1	2
Utilization Steps	2	2,3	1	2
Save	yes	yes	no	no

Table 1: *Data dependency analysis result of Listing 3*

tion, `spawn` blocks are replaced by corresponding kernel launching code. CPU code inserted via a `require` block is placed before the containing kernel's launching code.

Step 4 deduces all parameters that need to be passed to GPU kernels from the CPU. This is done by assuming all variables that are accessed at least once by both CPU and GPU are parameters. Kernel prototype and launching code are then generated, and parameter access in BSGP code is converted to specific instructions. For example, in Listing 3, `sorted_id` is a parameter. Writes to parameters in the GPU are disallowed and reported as compiling errors.

Step 5 finds for each value its defining superstep by locating its corresponding assignment. It then enumerates all uses of this value to see whether it is used in a different superstep. The value has to be saved if such a use exists. Table 1 summarizes the analysis result of Listing 3.

In Step 6, the analysis result of Step 5 is used to generate the actual value saving and loading code. For each value, the value saving code is generated at the end of its definition superstep, and the value loading code is generated at the beginning of each utilization superstep.

To support rank reassigning barriers, the value loading code generated above needs to be modified. The basic idea is to perform a logical rank reassignment by shuffling stored thread contexts at a barrier. We require each thread to set `thread.oldrank` to its previous rank after a `barrier(RANK_REASSIGNED)`, as shown in Listing 3. The compiler then moves the value loading code to immediately after the `thread.oldrank` assignment, and `thread.oldrank` is used when addressing the temporary stream. Values used in subsequent supersteps are loaded in a similar manner and moved to newly allocated temporary streams using

Listing 4 Pseudo code of final stream program for Listing 3

```
kernel pass1(int* ib, int* sorted_id, int* t0, int* t1){
    //superstep 1
    f = thread.rank/3; v = ib[thread.rank];
    local_sort(sorted_id, v);
    t0[thread.rank] = f; //value saving
    t1[thread.rank] = v; //value saving
}

kernel pass2(int* sorted_id, int* pf, int* owner,
  int* t0, int* t1, int* t2){
    //superstep 2
    thread.oldrank = sorted_id[thread.rank];
    //context shuffling for rank reassinging
    //load values from previous superstep using oldrank
    f = t0[thread.oldrank];
    v = t1[thread.oldrank];
    pf[thread.rank] = f;
    owner[thread.rank] = v;
    //v is moved to temporary stream t2 using new rank
    t2[thread.rank] = v;
}

kernel pass3(int* hd, int* owner, int* t2){
    //superstep 3
    v = t2[thread.rank]; //value loading
    if(thread.rank==0||owner[thread.rank-1]!=v)
        hd[v] = thread.rank;
}

findFaces(int* pf, int* hd, int* ib, int n){
    thread.size = n*3;
    sorted_id = dtempnew[thread.size]int;
    t0 = __newstream(thread.size);
    t1 = __newstream(thread.size);
    //launch superstep 1
    launch(thread.size,
        pass1(ib, sorted_id, t0, t1));
    global_merge(sorted_id);
    owner = dtempnew[n]int;
    t2 = __newstream(thread.size);
    //launch superstep 2
    launch(thread.size,
        pass2(sorted_id, pf, owner, t0, t1, t2));
    __freestream(t1);
    __freestream(t0);
    //launch superstep 3
    launch(thread.size,
        pass3(hd, owner, t2));
    __freestream(t2);
    //free lists allocated by dtempnew
    __dtempfree();
}
```

reassigned ranks. Subsequent supersteps may then proceed using new ranks. This is illustrated in `pass2` in Listing 4.

Finally, temporary streams are generated and assigned to saved values. We use a graph optimization algorithm to minimize peak memory consumption, as described in Section 4.3.

4.3 Minimization of Peak Memory Consumption

In this subsection, we use the simple BSGP program shown in Listing 5 as an example to explain our memory consumption optimization algorithm. Table 2 lists the values required to be saved in Listing 5.

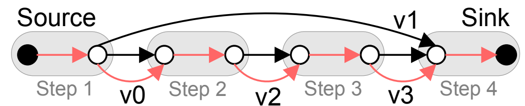
We first build a directed acyclic graph for all supersteps and saved values according to Table 2. As shown in Fig. 4(a), for each superstep, two nodes are created: one for the beginning of the superstep's execution and the other for the end. These nodes are connected by two kinds of edges. First, each node is connected to its succeeding node in execution order by a *non-value edge*. Second, for each saved value, the end node of its definition superstep is connected to the beginning node of its last utilization superstep by a *value*

Listing 5 Testing program for memory optimization.

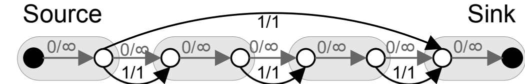
```
void test(int* a){
    spawn(1){
        //superstep 1
        v0 = a[0]; v1 = a[1];
        barrier;
        //superstep 2
        v2 = v0+v0;
        barrier;
        //superstep 3
        v3 = v1+v2;
        barrier;
        //superstep 4
        v4 = v3+v1;
        a[i] = v4;
    }
}
```

Value	v0	v1	v2	v3
Definition Step	1	1	2	3
Utilization Steps	2	3, 4	3	4

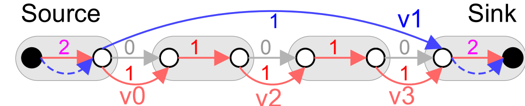
Table 2: *Data dependency analysis result of Listing 5.*



(a) Constructed graph. Edges for values are labeled with names. Nodes are labeled with corresponding superstep.



(b) Flow graph. Edges are labeled with flow requirement/capacity.



(c) Minimum flow. Edges are labeled with flow. Flow paths are colored.

Figure 4: *Minimum flow for temporary stream optimization. Nodes corresponding to the same superstep are grouped in grey boxes.*

edge. We also denote the beginning node of the first superstep as the *source*, and the end node of the last superstep as the *sink*.

An allocated temporary stream can then be mapped to a unique path from the source to the sink by connecting the value edges for all values assigned to the stream with non-value edges. The red path in Fig. 4(a) is an example. Minimizing the number of allocated temporary streams is equivalent to covering all value edges using a minimal number of paths from the source to the sink, i.e., a minimum flow in the graph, which is a classical graph theory problem [Ford and Fulkerson 1962] and the optimal solution can be computed in polynomial time.

To apply the minimum flow algorithm, each value edge is assigned a minimal flow requirement of one and a capacity of one. Each non-value edge is assigned with a minimal flow requirement of zero and a capacity of $+\infty$. The resulting graph is drawn in Fig. 4(b).

The minimum flow for Fig. 4(a) is shown in 4(c). The corresponding temporary stream allocation allocates `v0`, `v2`, `v3` in one stream and `v1` in another stream, as shown in the colored paths.

4.4 Implementation

We implemented the BSGP compiler in the CUDA GPU stream environment. The compilation of a BSGP program consists of the following stages:

1. The source code is compiled to static single assignment form (SSA) as in [Cytron et al. 1991].
2. The algorithm in Section 4.2 is carried out on each `spawn` block's SSA form.
3. Generated kernels are translated to CUDA assembly code, on which the CUDA assembler is applied. The resulting binary code is inserted into the CPU code as a constant array, and CUDA API calls are generated to load the binary code.
4. The object file or executable is generated from the CPU code by a conventional CPU compiler.

We choose the SSA representation mainly for simplification of the data dependency analysis. Since there is only one assignment for each SSA variable, the concept of value as defined in Section 4.1 is equivalent to that of an SSA variable, and this considerably simplifies data dependency analysis. Operating on SSA also allows us to directly generate optimized assembly code without calling CUDA's built-in high-level compiler and thus reduces compilation time significantly. CUDA specific optimizations, such as register allocation and instruction scheduling, are not sacrificed with our choice since a majority of them are handled in the assembler.

In addition to its stream environment, CUDA offers additional features including local communication and cached memory read. Taking advantage of these features is important for efficient implementation of many parallel primitives including *scan*. As the features do not conflict with BSGP, we provide the programmer with access to them through an interface similar to CUDA's original high level API and optionally through inline assembly. CUDA also provides a built-in profiler for kernels. To utilize it for BSGP programs, our compiler generates a log file to map generated kernels to BSGP source code positions.

Although our current implementation is for the CUDA stream environment, the compilation algorithm does not rely on CUDA specific features. Therefore it should be possible to port BSGP to other stream environments, such as ATI's CTM [ATI 2006].

Limitation A limitation of our BSGP compilation algorithm is its inability to handle flow control across barriers. The original BSP model permits barriers to be placed in flow control structures, assuming all barriers are reached by either all threads or no threads. This feature is useful for architectures on which the whole processing is done in parallel. In stream processing, however, a control processor is available for uniform flow controls and the need for such barriers is reduced. This limitation is inherited from stream processing, where a barrier may only be achieved by kernel launch termination.

5 BSGP Language Constructs

BSGP is a C like language with special constructs dedicated for BSP programming on the GPU. In this section we discuss several constructs related to GPU programming.

`spawn` and `barrier` As illustrated in Listing 1, 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. `barrier(RANK_REASSIGNED)` is a special type of barrier for reassigning thread ranks, as explained in Section 4.

In BSGP programs, we assume variables defined in `spawn` blocks are local to individual threads. Everything else resides in video memory, i.e., the global memory shared by all threads. For exam-

ple, in Listing 1, variable `f` is local to each thread, while memory pointed to by `pf` is global.

Cooperation with CPU using `require` In stream processing, certain crucial operations such as resource allocation and detailed kernel launch configuration are only available to the control processor. To address this issue, we allow the control processor code to be inserted into BSGP source code as a `require` statement block. At run time, code inserted this way is executed before the containing superstep is launched.

Emulating Thread Manipulation (`fork` and `kill`) For applications that involve data amplification/reduction, such as the adaptive tessellation example shown later in this paper, it is desirable to create/destroy threads to improve load balancing as the application data is amplified/reduced. In coarse-grain parallel programming, thread manipulation is accomplished by primitives such as fork and kill. Unfortunately current GPUs do not support these thread manipulation primitives; only the total number of threads is specified at a kernel launch. To address this problem, we emulate thread manipulation through a set of collective APIs using a multi-superstep algorithm.

Listing 6 illustrates our thread manipulation APIs using the sample code for extracting numbers from different regions of a plain text file. Utility routines are omitted for simplicity. An expanded version of this source code is used for parsing XML fields in the X3D parser shown later in this paper.

Listing 6 Number extraction. Utility routines are omitted for simplicity

```
/*
extract numbers from plain text
input:
    begin/end: offset to begin/end of regions
    n: number of regions to parse
returns:
    parsed numbers
*/
float* getNumbers(int* begin, int* end, int n){
    float* ret = NULL;
    spawn(n){
        id = thread.rank;
        s = begin[id]; e = end[id];
        pt = s+thread.fork(e-s+1);
        c = charAt(pt-1); c2 = charAt(pt);
        thread.kill(isDigit(c)||!isDigit(c2));
        require
            ret = dnew[thread.size]float;
        ret[thread.rank] = parseNumber(pt);
    }
    return ret;
}
```

Initially, a thread is spawned for each region. The thread forks an additional thread for each character in the region. All threads except those corresponding to the first letter of a number are killed. Remaining threads proceed to extract the number and write the result to a newly allocated list. A detailed description of `fork` and `kill` is given later in Appendix A.

Thread manipulation is implemented by adjusting `thread.size` in the `require` block and reassigning thread ranks. No additional compiler intervention is needed.

Reducing barriers using `par` The BSGP compilation algorithm always bundles the first superstep of a function with its caller. This can sometimes lead to suboptimal results because of the sequential order of BSGP supersteps. Listing 7 provides an example. In both `sort_idx(A)` and `sort_idx(B)`, the `local_sort` part should have been bundled into the superstep that generates `A` and `B`. However, our compiler simply inlines the `sort_idx` code and yields three supersteps as follows:

```
sorter(int n){
    spawn(n){
        A = functionA();
        B = functionB();
        //first sort_idx
        require
            sorted_idA = dtempnew[n]int;
        local_sort(sorted_idA, A);
        barrier;
        require
            global_merge(sorted_idA);
        idxA = sorted_idA[thread.rank];
        //second sort_idx
        require
            sorted_idB = dtempnew[n]int;
        local_sort(sorted_idB, B);
        barrier;
        require
            global_merge(sorted_idB);
        idxB = sorted_idB[thread.rank];
        //more code
    }
}
```

In theory it is possible for a compiler to automatically deduce that `sort_idx(B)` is independent of the first sort `sort_idx(A)`. In practice, routines like `sort_idx` often use low-level features such as local communication extensively, making the deduction extremely difficult. For this reason, we provide the `par` construct to let the programmer control the bundling behavior by restructuring the code. A `par` may be used in Listing 7 to reduce one pass as shown in Listing 8.

The `par` construct specifies that all statements in the following block are independent of each other. During compilation, the compiler aligns barriers in the statements and bundles the corresponding supersteps together as illustrated in Fig. 5. After inline expansion and `par` expansion, Listing 8 yields an optimized result with two supersteps:

```
sorter(int n){
    spawn(n){
        A = functionA();
        B = functionB();
        //the two sorts in par
        require
            sorted_idA = dtempnew[n]int;
        local_sort(sorted_idA, A);
        require
            sorted_idB = dtempnew[n]int;
        local_sort(sorted_idB, B);
        barrier;
        require{
            global_merge(sorted_idA);
            global_merge(sorted_idB);
        }
        idxA = sorted_idA[thread.rank];
        idxB = sorted_idB[thread.rank];
        //more code
    }
}
```

The `par` construct has the limitations of neither including any optimization nor allowing more detailed control over superstep bundling. Nevertheless, `par` can handle simultaneous independent calls to the same collective function, which is what typically occurs in applications.

Communication Intrinsics (`thread.get` and `thread.put`) BSGP also supports remote variable access intrinsics for thread communication. The interface is analogous to that of the BSPlib [Hill et al. 1998].

Listing 7 Pseudo code of `sort_idx` and a program that computes sort index for two values

```
inline int sort_idx(int x){
    require
        sorted_id = dtempnew[n]int;
    local_sort(sorted_id, x);
    barrier;
    require
        global_merge(sorted_id);
    return sorted_id[thread.rank];
}

sorter(int n){
    spawn(n){
        A = functionA();
        B = functionB();
        idxA = sort_idx(A);
        idxB = sort_idx(B);
        //more code
    }
}
```

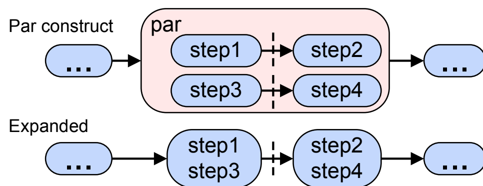


Figure 5: *Illustration of the* par *construct. Barriers between step 1,2 and 3,4 are aligned. Corresponding passes are then merged.*

`thread.get(r, v)` gets `v`'s value in the previous superstep from the thread with rank `r`. If `r` is not a valid rank, an undefined result is returned.

`thread.put(r, p, v)` stores `v`'s value to `p` in the thread with rank `r`. The stored value can only be read after the next barrier. If `r` is not a valid rank, no operation is performed. `put` is considered to be an assignment to `p`.

Handling communication intrinsics in the BSGP compiler is simple. Note that communication intrinsics are always used together with a barrier. This implies that the affected variable is always used across the barrier, and a temporary stream has already been allocated. Each communication operation is then converted to a read/write on the stream, using the source/target thread rank as the index. Also, in the case of a `thread.put`, the default saving code for the affected variable is removed.

Using communication routine `thread.get`, we further simiplify the BSGP program in Listing 1. As shown in Listing 9, `thread.get` is used to get the preceding thread's sort key by fetching `v`'s value before the `barrier` from thread `rk-1`. With this simplification the temporary list `owner` is removed.

Primitive Operations We provide a library of parallel primitives as collective functions. See Appendix A for details. All primitives are entirely implemented in the BSGP language without special compiler intervention. Programmers may write new primitives with similar interfaces.

While basic primitives like scan can be implemented portably using the communication intrinsics, currently we opt to make extensive use of CUDA specific features for maximal performance. For example, our optimized scan is $6.5\times$ faster than a portable version of the same algorithm for an array of 1M elements. However, this does not affect portability of BSGP applications, as explicit communication may be avoided by using the optimized primitives.

Listing 8 `par` example

```
sorter(int n){
    spawn(n){
        A = functionA();
        B = functionB();
        par{
            idxA = sort_idx(A);
            idxB = sort_idx(B);
        }
        //more code
    }
}
```

Listing 9 Find neighboring triangles (BSGP updated version using communication)

```
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);
        rk = thread.rank;
        pf[rk] = f;
        barrier;
        if(rk==0||thread.get(rk-1,v)!=v)
            hd[v] = rk;
    }
}
```

6 Experimental Results

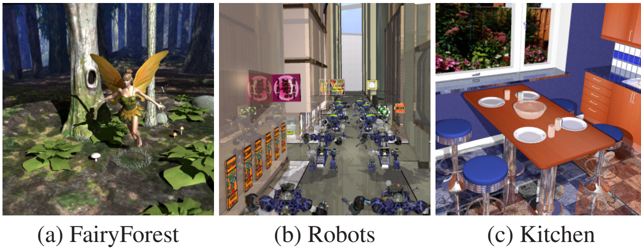
In this section, we use a few applications to demonstrate the advantages of BSGP over the stream programming language CUDA. As mentioned, the most important advantage of BSGP is that it is easy to read, write, and maintain programs. This assessment is necessarily subjective, and the best way to verify it is to examine BSGP source code and compare it with CUDA source code. For this reason, we provide the source code of all examples mentioned in the paper as supplementary materials in the DVD-ROM.

In addition to examining source code, we can compare the sizes of BSGP and CUDA source code as an objective measure of code complexity. We also compare the performance and memory consumption of the BSGP and CUDA code to ensure that BSGP's ease-of-programming does not come at the cost of performance and memory consumption. Our hardware platform for these comparisons is a PC with an Intel Xeon 3.7GHz CPU and a GeForce 8800 GTX graphics card.

GPU Ray Tracing Our first example is GPU ray tracing. We implemented the GPU ray tracing algorithm in [Popov et al. 2007] using both BSGP and CUDA. The algorithm is based on stackless kd-tree traversal and ray packets. It requires the construction of a kd-tree with "ropes" as a preprocess. Note that, as the cached memory read is exposed in CUDA via linear texture, the prefetching scheme using shared memory in [Popov et al. 2007] becomes unnecessary and packet traversal loses most of its memory latency advantages. For this reason, we just implemented a non-packet ray tracer for maximal flexibility and minimal register usage.

[Popov et al. 2007] uses a single kernel to do both ray tracing and shading. This complicates the ray tracing kernel and increases register usage. It also reduces the number of simultaneously executing threads (GPU occupancy) and hinders performance. Our ray tracer performs ray tracing and other computation such as shading and ray generation in different kernels. This modification results in better GPU occupancy (50% compared to the 33% reported in [Popov et al. 2007]) and a $2 \sim 3$ times speed up over [Popov et al. 2007].

Now we compare the code complexity and performance of the



(a) FairyForest (b) Robots (c) Kitchen

Figure 6: *Test scenes for GPU ray tracing. (a) FairyForest with 174K triangles and one point light. (b) Robots with 72K triangles and three point lights. The image is generated with one ray bounce. (c) Kitchen with 111K triangles and one point light. The image is generated with four ray bounces. All images are rendered at* 1024×1024 *image resolution.*

BSGP and CUDA implementations. As summarized in Table 3, BSGP has a clear advantage in code complexity, with a code size roughly 40% less than that of CUDA. More importantly, by examining the source code the reader will see that the BSGP source code naturally follows the ray tracing algorithm: three high level GPU functions in the BSGP code correspond to eye ray tracing, reflection/refraction ray tracing, and shadow ray tracing and shading computation respectively. In contrast, CUDA requires the programmer to manually optimize stream/kernel organization and manage temporary streams, resulting in ten kernels. The more compact BSGP source code also makes maintenance easier.

Code complexity	CUDA	BSGP	Reduction
Code lines	815	475	42%
Code bytes	19.0k	12.1k	36%
# GPU funcs	10	3	70%

Table 3: *Code complexity comparison. "# GPU funcs" is the number of top-level GPU functions, i.e., kernels in CUDA or functions with* `spawn` *in BSGP. All code comments are stripped for fair comparison. Source code is available in the supplementary materials.*

Table 4 summarizes the performance and memory consumption for the BSGP and CUDA programs on three standard test scenes. The test scenes and viewpoints are shown in Fig. 6. The BSGP and CUDA programs have similar performance and memory consumption. In order to achieve high performance, both programs are carefully written and optimized. The same programmer developed both programs and found the BSGP program to be much easier to write. Specifically, the BSGP program took around one day to get an initial implementation and $2 \sim 3$ days for debugging and optimization. In comparison, the CUDA program took around $2 \sim 3$ days for initial coding and $4 \sim 5$ days for debugging and optimization.

Scene	FPS		Memory usage	
	CUDA	BSGP	CUDA	BSGP
FairyForest	9.73	9.97	80Mb	80Mb
Robots	3.36	3.42	145Mb	153Mb
Kitchen	4.00	4.61	144Mb	150Mb

Table 4: *Comparison for performance and memory consumption.*

Particle-based Fluid Simulation We also implemented the "particle" demo provided in CUDA SDK, using BSGP (see Fig. 7). For simplicity, we only rewrote the sorting-based simulation module using BSGP, and reused the non-CUDA GUI code in the demo.

The BSGP version of the module was written in about an hour, and contains 154 lines of code. Including the GUI code, the final BSGP project contains 1579 lines of code, and runs at 290 FPS. The CUDA project contains 2113 lines of code, and runs at 187 FPS. The line counts exclude comments. Sort modules in the CUDA project are also excluded for fairness, because BSGP provides a sort primitive in its library. Frame rates are measured after a simulation reaches a steady state from the default initial position.

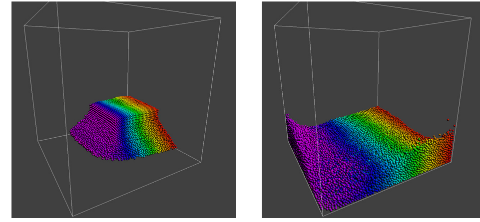


Figure 7: *Two frames from the particle-based fluid simulation.*

Overall, the BSGP version has a clear advantage over CUDA in terms of code complexity. Also, while neither implementation is very well optimized, the BSGP version is about 50% faster. The BSGP compiler generates more optimal kernel/temporary stream organization than in the hand-coded CUDA version.

GPU X3D Parser Our next example is a GPU X3D parser. X3D is the ISO standard for real-time 3D computer graphics and the successor to Virtual Reality Modeling Language (VRML). Although the GPU is the natural choice for rendering X3D scenes, existing X3D browsers rely on the CPU for parsing X3D files [Parisi 2003; ARTIS 2004]. This is slow because considerable parsing is required to convert an X3D file to a format ready for GPU rendering. An additional overhead is that the parsing result needs to be copied to the GPU before rendering. The result is a relatively long wait to load X3D scenes for display.

We implemented a GPU X3D parser using BSGP. The parsing is accelerated by the GPU and copying of results to the GPU is avoided. As a result, the loading time of X3D scenes is significantly reduced. Details of the parsing algorithm are described in the supplementary material in the DVD-ROM. Note that our X3D parser is only a prototype. We currently only parse data related to rendering and ignore scripts and other advanced content.

We did not implement the GPU X3D parser using CUDA because it would be extremely difficult due to the following reasons:

- Implementing the parser on the GPU is far from straightforward. X3D is composed of many independent constructs. When implementing the parser in BSGP, we found it highly desirable to add new features incrementally. This allows a unit test to be performed after each addition. Unfortunately, incrementally adding features to a CUDA program is very difficult because the dataflow changes with newly-added features and to avoid performance penalties, all affected kernels must be refactorized to optimize for the new dataflow.
- The source code complexity would be too high. Our BSGP code is grouped into 16 top-level GPU functions, with each one or two functions corresponding to an algorithmic step. For an equivalent CUDA program, the source code would be scattered among 82 kernels as in our BSGP compilation result.

We benchmarked our X3D parser against the commercial Flux system [Parisi 2003] and the open source X3DTookKit [ARTIS 2004]. The scenes are downloaded from Flux's official website: `http://www.mediamachines.com/`.

Compressed X3D files are decompressed to plain text format prior to our benchmark. The loading/parsing time comparison is shown in Table 5. Our GPU parser is orders of magnitude faster than CPU parsers. The benefit of the GPU parser is most noticeable for scenes with high geometry complexity. Even with I/O time included, our

(a) Paladin Woman (b) Building

Figure 8: *Rendering results from our X3D parser. (a) Paladin Woman with 7.03MB plain text and two textures; (b) Building with 1.56MB plain text and 25 textures.*

Scene	Parser	T_{total}	T_{IO}	T_{parse}
Fig. 8(a)	Ours	183ms	132ms	51ms
	Flux	2948ms		2816ms
	X3DTK	3132ms		3000ms
Fig. 8(b)	Ours	609ms	586ms	23ms
	Flux	836ms		250ms
	X3DTK	2950ms		2370ms

* T_{total} for Flux is measured by a daemon program that tracks open file dialog and pixel color change.

Table 5: *X3D loading/parsing time comparison. T_{total} is the total loading time, i.e., the time span between file name specification and first frame being ready to render. T_{IO} is the time for loading the raw X3D file and textures, which is the same for all programs. $T_{parse} = T_{total} - T_{IO}$ is the parsing time.*

system is still an order of magnitude faster for the scene with detailed geometry in Fig. 8(a).

GPU Adaptive Tessellation The final example is an adaptive tessellation routine described in [Moreton 2001]. The routine is used in a displacement map based terrain renderer to perform view-dependent tessellation.

The terrain renderer first generates a fixed amount of initial triangles and sends them to the tessellator. View dependent tessellation is then performed to generate tessellated triangles. The displacement map is looked up to produce the final geometry. The final rendering is done using OpenGL.

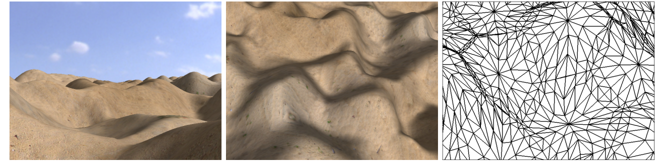
The tessellation routine contains the following steps:

1. View culling: cull input triangles outside the view frustum.
2. Compute view-dependent tessellation factor.
3. Compute output size.
4. Allocate memory to hold final geometry.
5. Generate tessellated triangles.

We implemented the tessellation algorithm using BSGP, both without and with thread manipulation. Instead of spending a lot of time to optimize the code for high performance, we choose to obtain a quick implementation in each case.

The implementation without thread manipulation naturally follows the algorithm steps. The computation is parallelized over all input triangles, i.e., one thread is created for each input triangle. View culling is performed using a conditional judgement `if`. For each triangle inside the view frustum, the view-dependent tessellation factor is calculated. Then the output size is computed and memory allocation is performed in a `require` block. Finally, geometry is generated and written to allocated memory.

The implementation with thread manipulation also follows the algorithm steps. First, the computation is parallelized over all triangles, i.e., one thread is initially created for each input triangle. View culling is performed using `thread.kill`. After the output size is computed and memory is allocated, a thread is forked for each output vertex using `thread.fork`. All subsequent computations are thus parallelized over all output vertices. Since the number of output vertices is much greater than the number of initial input triangles, GPU's large scale parallelism is fully exploited. Coordinates for each vertex are computed entirely in parallel and are directly returned to the terrain renderer without temporarily storing in memory.



(a) Side view (b) Top view (c) Tessellated mesh

Figure 9: *Terrain rendering results. The terrain is generated from a 512×512 displacement map. All images are generated at 640×480. (a) side view, 1.14M tessellation generated vertices; (b) top view, 322k tessellation generated vertices; (c) zoom in of the tessellation pattern. For illustration, the tessellation is coarser than used in actual rendering.*

View	BSGP (no thread man.)		BSGP (with thread man.)	
	T_{tess}	FPS	T_{tess}	FPS
Fig. 9(a)	43.9ms	21.0	3.62ms	142
Fig. 9(b)	5.0ms	144	2.1ms	249

Table 6: *Tessellation time and render performance. T_{tess} is the time taken to generated terrain geometry.*

Table 6 compares the performance of the two implementations. It can be seen that the version with thread manipulation significantly outperforms the other version, which does not exploit the full parallelism in Step 5 and degrades tessellation performance by a factor of 10 at the high detail level in Fig. 9(a).

7 Conclusion and Future Work

We have presented BSGP, a new programming language for general purpose computation on the GPU. The most important advantage of BSGP is that it makes it easy to read, write, and maintain GPU programs. The reader can appreciate BSGP's ease of programming from our comparative analysis of BSGP and CUDA programs implementing the same algorithm. Additional supporting evidence from our experiments is that, for every pair of well written BSGP and CUDA programs targeting the same application, the BSGP program always has a significantly lower code complexity.

With the ever increasing computing power available on the GPU, there is a strong demand for programming tools that can harness this formidable raw power. Our contribution is not only to bring the successful BSP model to GPU programming but also to show how to design BSGP and its compiler so that ease of programming can be achieved without sacrificing performance on GPUs. Indeed, our experiments indicate that BSGP programs achieve similar or better performance than well-optimized CUDA programs. With BSGP's ease of programming and competitive performance, programmers are empowered to tackle more complex GPU applications, including applications that would be extremely difficult to develop with existing GPU programming languages.

For future work we are interested in applying meta-programming techniques to BSGP. This will allow algebraic manipulation of collective functions and dynamic generation of BSGP programs. Secondly, the BSP model may potentially serve as a unified programming model for both coarse-grained parallel architectures and fine-grained parallel architectures including GPUs. We can therefore

envision the development of a unified BSP environment for parallelizing programs for CPUs as well as GPUs, and the power of parallel computing may thus be exploited with minimal extra work.

Acknowledgements

The authors would like to thank Matt Scott for his help with video production. We are also grateful to the anonymous reviewers for their helpful comments.

References

ARTIS, 2004. X3DToolKit homepage. http://artis.imag.fr/Software/X3D/.

ATI, 2006. Researcher CTM documentation. http://ati.amd.com/companyinfo/researcher/documents.html.

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.

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.

FOLEY, T., AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of Graphics Hardware*, 15–22.

FORD, L. R., AND FULKERSON, D. R. 1962. *Flows in Networks*. Princeton University Press.

FORTUNE, S., AND WYLLIE, J. 1978. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, 114–118.

GRESS, A., AND ZACHMANN, G. 2006. GPU-ABiSort: optimal parallel sorting on stream architectures. In *Parallel and Distributed Processing Symposium*, 45–54.

HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A., 2007. CUDPP homepage. http://www.gpgpu.org/developer/cudpp/.

HILL, J. M. D., MCCOLL, B., STEFANESCU, D. C., GOUDREAU, M. W., LANG, K., RAO, S. B., SUEL, T., TSANTILAS, T., AND BISSELING, R. H. 1998. Bsplib: The bsp programming library. *Parallel Comput. 24*, 14, 1947–1980.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. In *I3D'07*, 167–174.

KALE, L. V., AND KRISHNAN, S. 1993. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Notices 28*, 10, 91–108.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. In *Proceedings of SIGGRAPH '03*, 896–907.

MCCOOL, M., AND DU TOIT, S. 2004. *Metaprogramming GPUs with Sh*. AK Peters Ltd.

MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Proceedings of Graphics hardware*, 57–68.

MCCOOL, M., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. *ACM Trans. Graph. 23*, 3, 787–795.

MORETON, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of Graphics Hardware*, 25–32.

NVIDIA, 2007. CUDA homepage. http://developer.nvidia.com/object/cuda.html.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of SIGGRAPH'98*, 159–168.

PARISI, T. 2003. Flux: lightweight, standards-based web graphics in xml. In *ACM SIGGRAPH 2003 Web Graphics*, 1–1.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH'00*, 425–432.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance gpu ray tracing. In *Proceedings of Eurographics'07*, 415–424.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH'01*, 159–170.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware*, 97–106.

SKILLICORN, D. B., HILL, J. M. D., AND MCCOLL, W. F. 1997. Questions and Answers about BSP. *Scientific Programming 6*, 3, 249–274.

TARDITI, D., PURI, S., AND OGLESBY, J. 2006. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 325–335.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Communications of the ACM 33*, 8, 103–111.

WEGMAN, M. N., AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2, 181–210.

A BSGP Primitive Operations

Currently, we provide three kinds of BSGP primitives. The implementation of these primitives are described in the supplementary material.

Supported data parallel primitives include:

- **`reduce(op, x)`**: Collective reduction of `x` using operator `op`. The returned value is the reduction result. `op` has to be associative, such as max, min and $+$;
- **`scan(op, x)`**: Collective forward exclusive scan of `x` using associative operator `op`. Scan result overwrites `x`. The returned value is the reduction result as a byproduct;
- **`compact(list, src, keep, flag)`**: Collective stream compaction. Each `src` whose `keep` is `true` is compacted and appended to `list`. `flag` specifies whether `list` should be cleared before appending;
- **`split(list, src, side, flag)`**: Collective stream splitting. Every `src` is split according to `side` into two pieces, which are then appended to `list`, with the `false` piece preceding the `true` one. `flag` has similar semantic as in `compact`;
- **`sort_idx(key)`**: Collective sorting and index returning. Let K_i be `key` in thread with rank i and r_i be `sort_idx`'s return value. Then r_i satisfies $K_{r_i} \leq K_{r_j}$ for $i \leq j$;

Supported rank adjusting primitives include:

- **`thread.split(side)`**: Split threads. The rank is reassigned such that a thread with a `false side` has a smaller rank than a thread with a `true side`. Relative rank order is preserved among threads of the same `side`;
- **`thread.sortby(key)`**: Collective rank reassignment sorting. Let K_i be `key` in thread with rank i. Thread ranks are adjusted such that after `thread.sortby` returns, $K_i \leq K_j$ for all $i \leq j$. Relative rank order is preserved among threads with the same `key`, i.e., the sort is stable;

Supported thread manipulation primitives include:

- **`thread.kill(flag)`**: Kill the calling thread if `flag` is true;
- **`thread.fork(n)`**: Fork `n` child threads. All child threads inherit the parent's local variables. A unique ID between 0 and `n-1` is returned to each child thread. The parent thread no longer exists after `fork`;

Note that both `fork` and `kill` reassign resulting threads' ranks to numbers in the range of `0...thread.size-1` while preserving parent threads' relative rank order.