BSGP: Bulk-Synchronous GPU Programming

Thomas Hauenstein, Etienne Assoumou Mengue

Hardware-Software-Co-Design

Universität Erlangen-Nürnberg

sithhaue@cip.informatik.uni-erlangen.de, sietasso@cip.informatik.uni-erlangen.de
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
Motivation

➢ Graphic cards offer great amounts of processing power

➢ However, programming the GPU for general purpose computing is by no means easy:
  ▪ Program readability and maintenance
  ▪ Handling of temporary streams for “inter-kernel value passing” is difficult and error prone.
  ▪ Inefficient code reuse

➢ These reasons led to the development of BSGP, a new programming language for general purpose computing on GPUs, which is easy to read, write and maintain.
Motivation (2)

- A typical CUDA program consists of several kernels
- Supplies high performance
- Makes GPU programming hard

- C-like programming language
- Looks much like a sequential C program
- Based on the BSP (Bulk synchronous parallel) model
- Similar or better performance than other languages (i.e. CUDA)
Motivation (3)

- Based on the BSP (Bulk Synchronous Parallel) model
  - Several processors are connected by a communication network
  - Each processor has its own local memory
  - Data exchanged via the communication network is available after the next barrier synchronization
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
BSGP and Stream processing

Before we start, a few definitions first...

**Barrier:**
A form of synchronization in SPMD programming
When a thread reaches the **barrier** statement, it suspends its execution until all threads reach the barrier too.

**Superstep**
Everything between two barriers is called a **superstep**.
A **superstep** is translated into a GPU kernel by the BSGP compiler

**Collective operation:**
A **collective operation** is an operation, that has to be performed simultaneously by all threads.
The input of one thread may affect the output of other threads
BSGP and Stream processing (2)

- Comparison between BSGP and CUDA based on a small example:
  - Given a triangle mesh's connectivity, compute a list of the one-ring neighboring triangles for each vertex.

```
<table>
<thead>
<tr>
<th>vertex</th>
<th>triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>t1, t2, t3, t4, t5</td>
</tr>
<tr>
<td>v2</td>
<td>t4, t5, t6, t7, t8, t9</td>
</tr>
<tr>
<td>v3</td>
<td>......</td>
</tr>
</tbody>
</table>
```

- \( m \) vertices, \( n \) triangles, connectivity: array with \( 3n \) integers ranging from 0 to \( m - 1 \)
BSGP and Stream processing (3)

Basic idea:
- Triplicate each triangle and associate each of these triangles with one vertex
- Sort the triplicated triangles using the associated vertex indices as the sort key
- Triangles sharing the same vertex are grouped together
- Compare each sort key with its predecessor, to get a pointer to the beginning of each vertex's list
BSGP and Stream processing (4)

➢ The BSGP code is pretty much straightforward, compact and easy to read:

```c
/*
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;
  }
}
```

➢ However, this does not hold for the CUDA implementation....
BSGP and Stream processing (5)

Listing 2 Find neighboring triangles (CUDA version)

```c
#include "cudaPP.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]<<16)+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&0x0fffff);
        owner[rk]=(k>>16);
    }
}
__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
templ: used twice for different purpose
1. temporary stream 1 for cudaPPSort
2. associated vertex of each face (owner)
temp2: temporary stream 2 for cudaPPSort */
```

```c
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* templ;
    int* temp2;
    cudaMemcpy((void**)&key,n3*sizeof(unsigned int));
    cudaMemcpy((void**)&sorted,n3*sizeof(unsigned int));
    cudaMemcpy((void**)&templ,n3*sizeof(int));
    cudaMemcpy((void**)&temp2,n3*sizeof(int));
    before_sort<<<ng,szblock>>>(key,ib,n3);
    //call the CUDAPP sort
    {
        CUDAProcSortConfig sp;
        CUDAProcScanConfig scanconfig;
        sp.numElements = n3;
        sp.datatype = CUDAPP_UINT;
        sp.sortAlgorithm = CUDAPP_SORT_RADIX;
        scanconfig.direction = CUDAPP_SCAN_FORWARD;
        scanconfig.exclusivity = CUDAPP_SCAN_EXCLUSIVE;
        scanconfig.maxNumElements = n3;
        scanconfig.maxNumRows = 1;
        scanconfig.datatype = CUDAPP_UINT;
        cudaProcInitializeScan(&scanconfig);
        cudaProcScan(sp,scanconfig);
        cudaProcSort(sorted,key,templ,temp2,&sp,0);
        cudaProcFinalizeScan(sp,scanconfig);
    }
    after_sort<<<ng,szblock>>>(pf,templ,sorted,n3);
    make_head<<<ng,szblock>>>(hd,templ,n3);
    cudaFree(temp2);
    cudaFree(templ);
    cudaFree(sorted);
    cudaFree(key);
}
```
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
BSGP Language Constructs

➢ **spawn** and **barrier**

- \texttt{spawn(x)} launches \texttt{x} threads for the execution of the current block
- Variables defined inside \texttt{spawn} blocks are local to individual threads
- Everything else resides in the global video memory
- We've already heard about barriers...

```c
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;
    }
}
```
BSGP Language Constructs (2)

- **require**
  - Insert and execute CPU code in BSGP code
  - The code is executed before launching the kernel

```c
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;
    }
}
```
BSGP Language Constructs (3)

- **fork and kill**
  - CUDA or the like do not allow explicit thread destruction/creation
  - BSGP emulates this by changing the total number of threads and reassigning thread ranks

```c
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;
}
```
BSGP Language Constructs (4)

- **par**
  - Specifies, that all statements in the following block are independent of each other:

```cpp
sorter(int n) {
    spawn(n) {
        A = functionA();
        B = functionB();

        par{
            idxA = sort_idx(A);
            idxB = sort_idx(B);
        }
        // more code
    }
}
```
BSGP Language Constructs (5)

- **thread.get and thread.put**
  - Allow communication between threads:
    - `thread.get(r, v)` gets `v`'s value in the previous superstep from the thread with rank `r`.
    - `thread.put(r, p, v)` stores `v`'s value to `p` in the thread with rank `r`. The value can only be read after the next barrier.

```
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;
        if(rk==0) owner[rk] = v;
        barrier;
        if(rk==0) | owner[rk-1] == v) {
            hd[v] = rk;
        }
    }
}
```
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
Compiling BSGP programs

- Compiler Design Issues:
  - Barrier synchronization:
    - Unlike in traditional software thread scheduling, not all threads are executed simultaneously on GPU hardware.
    - Threads are dynamically distributed to available processing units.
    - Resources like registers, holding the current thread context, are recycled upon thread completion => There is no persistent logical thread context!
    - It follows, that the BSGP compiler has to generate additional context saving code.
    - To keep this overhead at a low level, values are only saved, if they're used in the next superstep.
  - Minimizing the peak memory consumption:
    - Similar to the register allocation problem in compiler theory.
    - Graph optimization to compute the optimal solution in polynomial time.
  - Locality:
    - Adjust thread ranks so as to match physical locality with algorithmic locality.
Compiling BSGP programs (2)

```
spawn(n)
c = 0.12
x = a[rank]
g = exp(-x*x)
f = g
scan(f)
f *= c
a[rank] = g*f
```

```
inline function
```

```
spawn(n)
c = 0.12
x = a[rank]
g = exp(-x*x)
f = g
local_scan(f)
bARRIER
add_result(f)
f *= 0.12
a[rank] = g*f
```

```
Optimize
```

```
spawn(n)
x = a[rank]
g = exp(-x*x)
f = g
local_scan(f)
bARRIER
add_result(f)
f *= 0.12
a[rank] = g*f
```

```
Liveness analysis
```

```
Final stream program
```

```
pass0(a,t0,t1)
x = a[rank]
g = exp(-x*x)
f = g
local_scan(f)
t0[rank] = f
t1[rank] = g
```

```
pas01(a,t0,t1)
f = t0[rank]
g = t1[rank]
add_result(f)
f *= 0.12
a[rank] = g*f
```

```
lanscher()
t0 = new stream(n)
t1 = new stream(n)
launch(n, pass0(a,t0,t1))
launch(n, pass1(a,t0,t1))
delete t0; delete t1
```

```
generate kernels and kernel launching code
```

```
generate temporary stream management
```

```
pas00(a)
x = a[rank]
g = exp(-x*x)
f = g
local_scan(f)
```

```
pas01(a)
add_result(f)
f *= 0.12
a[rank] = g*f
```

```
lanscher()
launch(n, pass0(a))
bARRIER(f,g)
lanscher(n, pass1(a))
```
Compiling BSGP programs (3)

Step 2.: \(rk1\) can be replaced by \textit{thread.rank}

Step 4.: variables accessed at least once by the CPU and the GPU are parameters

Step 5.: decide whether a variable has to be saved

**Listing 3** Extended version of Listing 1 by expanding the inline function \textit{thread.sortby}.

```c
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;
        [...]
    }
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>f</th>
<th>v</th>
<th>rk0</th>
<th>rk1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition Step</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Utilization Steps</td>
<td>2</td>
<td>2,3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Save</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Compiling BSGP programs (4)

Minimization of Peak Memory Consumption

Listing 5: Testing program for memory optimization.

```c
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;
    }
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition Step</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Utilization Steps</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

x/y: flow requirement/capacity
Compiling BSGP programs (5)

- Implementation
  1. The source code is compiled to static single assignment form (SSA)
  2. The described compilation algorithm is applied to every spawn block's SSA form
  3. Generated kernels are translated into CUDA assembly code
  4. This assembly code is inserted into the CPU code as a constant array
  5. CUDA API calls are generated to load the binary code
  6. The executable is generated from the CPU code by a conventional CPU compiler
Outline

➢ Motivation

➢ BSGP and Stream processing

➢ BSGP Language Constructs

➢ Compiling BSGP programs

➢ Experimental Results

➢ Conclusion
Experimental Results

GPU Ray Tracing
Experimental Results (2)

➢ GPU Ray Tracing
  ▪ Both BSGP and CUDA are implemented and optimized by the same programmer
  ▪ Clear advantage in code complexity
  ▪ Similar performance and memory usage

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>BSGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render fps</td>
<td>4.00</td>
<td>4.61</td>
</tr>
<tr>
<td>Mem usage</td>
<td>144M</td>
<td>150M</td>
</tr>
<tr>
<td>Code lines</td>
<td>815</td>
<td>475</td>
</tr>
<tr>
<td># GPU funcs</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Coding days</td>
<td>2~3</td>
<td>1</td>
</tr>
<tr>
<td>Tuning days</td>
<td>4~5</td>
<td>2~3</td>
</tr>
</tbody>
</table>
Experimental Results (3)

➢ Adaptive Tessellation
  ▪ A displacement map based terrain renderer

<table>
<thead>
<tr>
<th>View</th>
<th>no thread man.</th>
<th>with thread man.</th>
<th># vert output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{tess}$</td>
<td>FPS</td>
<td>$T_{tess}$</td>
</tr>
<tr>
<td>Side</td>
<td>43.9ms</td>
<td>21.0</td>
<td>3.62ms</td>
</tr>
<tr>
<td>Top</td>
<td>5.0ms</td>
<td>144</td>
<td>2.1ms</td>
</tr>
</tbody>
</table>
Experimental Results (4)

- X3D parser
  - ISO standard for real-time 3D computer graphics
  - Currently no CUDA implementation because the implementation would be too complex

![Image of Paladin Woman and Building](image-url)

<table>
<thead>
<tr>
<th>Scene</th>
<th>Parser</th>
<th>$T_{total}$</th>
<th>$T_{IO}$</th>
<th>$T_{parse}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 8(a)</td>
<td>Ours</td>
<td>183ms</td>
<td>132ms</td>
<td>51ms</td>
</tr>
<tr>
<td></td>
<td>Flux</td>
<td>2948ms</td>
<td>2816ms</td>
<td>23ms</td>
</tr>
<tr>
<td></td>
<td>X3DTK</td>
<td>3132ms</td>
<td>3000ms</td>
<td>250ms</td>
</tr>
<tr>
<td>Fig. 8(b)</td>
<td>Ours</td>
<td>609ms</td>
<td>586ms</td>
<td>23ms</td>
</tr>
<tr>
<td></td>
<td>Flux</td>
<td>836ms</td>
<td>250ms</td>
<td>250ms</td>
</tr>
<tr>
<td></td>
<td>X3DTK</td>
<td>2950ms</td>
<td>2370ms</td>
<td></td>
</tr>
</tbody>
</table>

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

- Motivation
- BSGP and Stream processing
- BSGP Language Constructs
- Compiling BSGP programs
- Experimental Results
- Conclusion
Conclusion

➢ BSGP is a good approach to implement a new programming languages for GPUs, which is easy to read, write and maintain.

➢ The presented results showed, that BSGP has a much lower code complexity in comparison to CUDA, while performing pretty well.

➢ So it might be worth a look :-}
Conclusion

➢ BSGP is a good approach to implement a new programming languages for GPUs, which is easy to read, write and maintain.

➢ The presented results showed, that BSGP has a much lower code complexity in comparison to CUDA, while performing pretty well

➢ So it might be worth a look :-)
References

➢ Kun Zhou, Qiming Hou, Baining Guo, 2008. BSGP: Bulk Synchronous GPU Programming
➢ http://www.kunzhou.net/
➢ BSGP Programming Guide
Questions

Questions?
BSGP Language Constructs (6)

- Some minor differences to the standard C-syntax
  - `int[100] foo` instead of `int foo[100]`
  - For `goto` labels, the `:` is put in front of the label name
  - `else if` can be abbreviated as `elif`
  - Shortened `for`-loops: `for i=0:10`
  - BSGP allows implicit variable declaration
  - Single-line conditional statements (`if/while`) are followed by a `:`
    - `if a==0:return 1`
Compiling BSGP programs (delete)

Compilation Algorithm

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