BSGP: Bulk Synchronous GPU Programming

Qiming Hou - Tsinghua University
Kun Zhou - MSRA
Baining Guo - MSRA

Presented by Fu-Chung Huang
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
An Analogy

Procedure A

WorkStation A

Materials

Some Handling

Memory-less

WorkStation B

Procedure B

WorkStation C

Procedure C

Products

Local variables visible across work
The Streaming Model

Procedure A

WorkStation A

Materials

Some Handling

Memory-less

WorkStation B

Procedure B

Super Step 1

Kernel A(…)
{
  ...
  ...
}

Ping!

Super Step 2

Kernel B(…)
{
  ...
  ...
}

Pong!

Super Step 3

Kernel C(…)
{
  ...
  ...
}

Ping!
Stream Bundling

Materials → WorkStation A → WorkStation B → WorkStation C → Products

Some Handling

WorkStation D

Texture

Data B

Data D

Data B

Data D

Kernel Bundle(...)
{
  Get Data B; Process 1st half;
  Get Data D; Process 2nd half;
  Composite Output;
}
## Comparison

<table>
<thead>
<tr>
<th>Streaming (CUDA, Cg)</th>
<th>BSGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event based</td>
<td>Thread base</td>
</tr>
<tr>
<td>Faster (sometimes)</td>
<td>Easier to write</td>
</tr>
<tr>
<td>Temporary stream management</td>
<td>Variables are local throughout super-step</td>
</tr>
<tr>
<td>Need dirty tricks to optimize</td>
<td>Looks like sequential C, Readability</td>
</tr>
</tbody>
</table>
Bulk Synchronous Parallel (BSP) Model

- **Properties**
  - Simple to write
  - Independent of target architecture
  - Predictable performance

- **Good Match to GPU**
  - No communication except at barrier (end of super-step)
  - Ignoring inter-processor data locality (cache structure)
Contribution

- New paradigm for GPU programming
  - Easier to write, read, and maintain

- Automatic data management
  - On top of streaming model
  - Compiler design issue

- Better performance with much shorter code
  - ~50% shorter

- Allow incremental development
  - Difficult with current GPU programming
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
Example: 1-Ring Neighborhood Finding

Surface Description

<table>
<thead>
<tr>
<th>f0</th>
<th>v3</th>
<th>v2</th>
<th>v0</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>v4</td>
<td>v3</td>
<td>v1</td>
</tr>
<tr>
<td>f2</td>
<td>v1</td>
<td>v4</td>
<td>v2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Input:
- triangle meshes

Output:
- Vertices’ neighbor faces
- In parallel
Algorithm: 1-Ring Neighborhood Finding

1. Triplicate \( v \) tri with \( V \) as key
2. Sort the \( 3n \) elements
3. Find the head pointers
4. Return face array and head pointers
1-Ring Neighborhood Finding BSGP

- The inputs and outputs

```c
findFaces(int* face, int* head, int* input, int n) {
    spawn(n*3) {
        id = thread.rank;
        v = input[id];  // vertex id
        f = id/3;        // face id
        thread.sortBy(v);
        // allocate a temp list
        require
            owner = dtmpnew[n] int;
            new_id = thread.rank;
            owner[new_id] = v;
            barrier;
            if (new_id == 0 || owner[new_id-1] != v)
                head[v] = new_id;
    }
}```
1-Ring Neighborhood Finding

BSGP

```
findFaces(int* face, int* head, int* input, int n) {
    spawn(n*3) {
        id = thread.rank;
        v = input[id];    // vertex id
        f = id/3;          // face id
        thread.sortby(v);  // allocate a temp list
        require
            owner = dtmpnew[n] int;
        new_id = thread.rank;
        owner[new_id] = v;
        barrier;
        if (new_id == 0 || owner[new_id-1] != v)
            head[v] = new_id;
    }
}
```
BSGP Syntax

- spawn
  - Creating threads to run codes concurrently

- require
  - Cooperating with CPU
  - Executed only once before kernel launches

- barrier
  - Synchronizing every threads to have the same progress
1-Ring Neighborhood Finding CUDA (using CUDPP)

```c
const int szblock=256;
findFaces(int* face, int* head, int* input, int n) {
    int n3 = n*3;
    int nb = (n3+szblock-1)/szblock;
    unsigned int* key;
    unsigned int* sorted;
    int* templ1;
    int* templ2;
    cudaMalloc((void**)&key, n3*sizeof(unsigned int));
    cudaMalloc((void**)&sorted, n3*sizeof(unsigned int));
    cudaMalloc((void**)&templ1, n3*sizeof(int));
    cudaMalloc((void**)&templ2, n3*sizeof(int));
    before_sort<<<<nb, szblock>>>(key, input, n3);
    //call the CUDPP sort
    __global__ void d
    before_sort(unsigned int* key, int* input, int n3){
        int id = blockIdx.x*szblock + threadIdx.d.x.x;
        if(id < n3){ //Data Packing
            key[id] = (input[id] << 16) + id/3;
        }
        scanconfig.sortConfig = CUDPP_SORT_RADI_X;
        scanconfig.sortAlgorithm = CUDPP_SCAN_FORWARD;
        scanconfig.exclusivity = CUDPP_SCAN_EXCLUSIVE;
        scanconfig.maxNumElements = n3;
    }
```
1-Ring Neighborhood Finding
CUDA (using CUDA++)

```c
int temp2;
CUDAmalloc((void**)&key, n3*sizeof(unsigned int));
CUDAmalloc((void**)&temp1, n3*sizeof(int));
CUDAmalloc((void**)&temp2, n3*sizeof(int));
before_sort<<nb, szblock>>(key, input, n3);
{
    // call the CUDPP sort
    CUDPPSortConfig sortconfig;
    CUDPPScanConfig scanconfig;
    sortconfig.nunElements = n3;
    sortconfig.datatype = CUDPP_UI NT;
    sortconfig.sortAlgorithm = CUDPP_SORT_RADIUS;
    scanconfig.direction = CUDPP_SCAN_FORWARD;
    scanconfig.exclusivity = CUDPP_SCAN_EXCLUSIVE;
    scanconfig.maxNunElements = n3;
    scanconfig.maxNunRows = 1;
    scanconfig.datatype = CUDPP_UI NT;
    scanconfig.op = CUDPP_ADD;

cudppInitializationScan(&scanconfig);
SortConfig.scanConfig = &scanconfig;
cudppSort(sorted, key, temp1, temp2, &sortconfig, 0);
cudppFinalizeScan(sortconfig.scanConfig);
}

after_sort<<nb, szblock>>(face, temp1, sorted, n3);
make_head<<nb, szblock>>(head, temp1, n3);
cudaFree(temp2); cudaFree(temp1);
cudaFree(sorted); cudaFree(key);```
1-Ring Neighborhood Finding
CUDA

```c
__global__ void after_sort(int* face, int* owner, unsigned int* sorted, int n3) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id < n3) { // Data unpacking
        int k = sorted(id);
        // key[id] = (input[id] << 16) + id/3;
        face[id] = (k<<0xffff);
        owner[id] = (k>>16u);
    }
}

__global__ void make_head(int* head, int* owner, int n3) {
    int id = blockIdx.x * blockDim.x * blockDim.y + threadIdx.x;
    if(id < n3) {
        int v = owner(id);
        if(id==0 || v != owner[id-1])
            head[v] = id;
    }
}
```

| Own | v1 | v1 | v1 | v1 | v1 | v2 | v2 | v2 | ...
|-----|----|----|----|----|----|----|----|----|-----
| face| f1 | f2 | f3 | f4 | f5 | f6 | f4 | f3 | f7 | ...

17
1-Ring Neighborhood Finding

CUDA (using CUDPP)

```c
int main()
{
    int temp2;
    cudaMalloc((void**)&key, n3*sizeof(unsigned int));
    cudaMalloc((void**)&temp1, n3*sizeof(int));
    cudaMalloc((void**)&temp2, n3*sizeof(int));
    before_sort<<<nb, szblock>>>(key, input, n3);
{
    // call the CUDPP sort
    CUDPPSortConfig sortconfig;
    CUDPPScanConfig scanconfig;
    sortconfig.numElements = n3;
    sortconfig.datatype = CUDPP_UI NT;
    sortconfig.sortAlgorithm = CUDPP_SORT_RADIX;
    scanconfig.direction = CUDPP_SCAN_FORWARD;
    scanconfig.exclusivity = CUDPP_SCAN_EXCLUSIVE;
    scanconfig.maxNumElements = n3;
    scanconfig.maxNumRows = 1;
    scanconfig.datatype = CUDPP_UI NT;
    scanconfig.op = CUDPP_ADD;
    cudppInitializationScan(&scanconfig);
    Sortconfig.scanConfig = &scanconfig;
    cudppSort(sorted, key, temp1, temp2, &sortconfig, 0);
    cudppFinalizeScan(sortconfig.scanConfig);
}
after_sort<<<nb, szblock>>>(face, temp1, sorted, n3);
make_head<<<nb, szblock>>>(head, temp1, n3);
    cudaFree(temp2);      cudaFree(temp1);
    cudaFree(sorted);      cudaFree(key);
}
```
CUDA or Cg Drawbacks

- Several kernels in mind
  - What should be passed in/out (texture format)
  - Merge/split kernels

- Explicit flow
  - Live time for local variables
  - Data management (reduce/reuse texture)

- Data packing
  - Optimization (hard to maintain)
  - Deficiencies in CUDPP (sort by object?) (extra kernel?)
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
Basic BSGP Constructs

- spawn and barrier
- requires (CPU)
  - Only executed once
- put, get
  - Communications
- fork, kill
  - Thread manipulation
- Par
Thread Communication: put, get

- thread.put(r, p, v)
  - Set v’s value to p in next super step to thread with ID r.

![Diagram of the BSP Model]

- Super Step1
  - C1
  - L + B * d1

- Super Step2
  - C2
  - L + B * d2

- Processor/Memory
- Processor/Memory
- Network (Bandwidth + Latency)
- BSP Model
Thread Communication: put, get

- thread.get(r, v):
  - get v’s value in previous super-step from thread with ID r.

Example:

```c
findFaces(int* face, int* head, int* input, int n) {
  spawn(n*3) {
    id = thread.rank;
    v = input[id];  // vertex id
    f = id/3;  // face id
    thread.sortby(v);
    // allocate a temp list
    require
      owner = dtenppnew[n] int;
    new id = thread.rank;
    owner[new id] = v;
    barrier;
    if (new id == 0 || owner[new id - 1] != v)
      head[v] = new id;
  }
}
```
Thread Manipulation: fork, kill

 Done using multi-superstep emulation

 child_id = thread.fork(k)
   Kill itself
   Creak k threads and return children’s relative order from 0~k-1
   Children inherit everything from their parent

 thread.kill(condition)
   Kill itself when condition is true
Thread Manipulation: fork, kill

- The Input and output

```c
float* getNumbers(int* start, int* end, int n) {
    float* ret = NULL;
    spawn(n) {
        id = thread.rank;
        s = start[id];
        e = end[id];
        child_id = thread.fork(e-s+1);
        loc = s + child_id;
        c_pre = charAt(loc-1);
        c_cur = charAt(loc);
        // survive if previous char is not a number but I am
        thread.kill(isDigit(c_pre) || !isDigit(c_cur));
        require
            ret = dnew[thread.size] float;
        ret[thread.rank] = parseNumber(loc);
    }
    return ret;
}
```
Thread Manipulation: fork, kill

```c
float* getNumbers(int* start, int* end, int n) {
    float* ret = NULL;
    spawn(n) {
        id = thread.rank;
        s = start[id];
        e = end[id];
        child_id = thread.fork(e-s+1);
        loc = s + child_id;
        c_pre = charAt(loc-1);
        c_cur = charAt(loc);
        // survive if previous char is not a digit but I am
        thread.kill(isDigit(c_pre) || !isDigit(c_cur));
        require
            ret = dnew[thread.size] float;
        ret[thread.rank] = parseNumber(loc);
    }
    return ret;
}
```
Reducing Barriers: par
Reducing Barrier: par

```c
deriv(float* dpdq, float* p, float* q, int n) {
    spawn(n) {
        par {
            a = dydx(p);
            b = dydx(q);
        }
        dpdq[thread.rank] = a / b;
    }
}
```

Original

Step 1 → Step 2 → Step 3 → Step 4

Par

Step 1 → Step 3 → Step 2 → Step 4
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
Compilation

- Build BSP on top of GPU (using CUDA)
  - Design of compiler (skip details here)

- Implement barriers
  - Context saving code

- Generate efficient/compact stream code
  - Analysis of data dependency
Context Saving

\[ \text{spawn}(3) \{ \]
\[ \quad x = \text{task}_A(); \]
\[ \quad \text{barrier}; \]
\[ \quad y = \text{task}_B(x); \]
\[ \} \]

1. Create temp memory for variables
2. Save old_id (shuffle after barrier)
3. Retrieve variables using old_id
Minimizing Temporary Stream

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Utilized</td>
<td>2</td>
<td>3,4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Do we need 4 memory spaces here?
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
Result 1 - Ray Tracing

- [Popov et al. 2007] Stackless KD-Tree Traversal for High Performance GPU Ray Tracing
  - 1 Kernel to do [ray-tracing + shading], low *occupancy*
  - CUDA re-implement (reduce register usage)
    - 50% vs. 33% occupancy, 2x~3x faster
  - BSGP: 3 Kernels : [eye rays], [reflection/refraction], [shadow]

<table>
<thead>
<tr>
<th>Code Complexity</th>
<th>CUDA</th>
<th>BSGP</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Lines</td>
<td>815</td>
<td>475</td>
<td>42%</td>
</tr>
<tr>
<td>Code Bytes</td>
<td>19.0K</td>
<td>12.1K</td>
<td>36%</td>
</tr>
<tr>
<td>#Kernels</td>
<td>10</td>
<td>3</td>
<td>70%</td>
</tr>
</tbody>
</table>
Occupancy

I Definition
   I # active threads / Max. # supported threads

I Where does it come from?
   I Fixed amount of registers and memory
   I Lower occupancy means longer total execution time

I Implication
   I Shorter kernel, use less variables
### Result 1 - Ray-Tracing (cont’d)

<table>
<thead>
<tr>
<th>Labors</th>
<th>CUDA</th>
<th>BSGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing</td>
<td>2~3 Days</td>
<td>1</td>
</tr>
<tr>
<td>Debug/Opt</td>
<td>4~5</td>
<td>2~3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scene</th>
<th>Frames Per Second (FPS)</th>
<th>Memory (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDA</td>
<td>BSGP</td>
</tr>
<tr>
<td>Fairy Forest</td>
<td>9.73</td>
<td>9.97</td>
</tr>
<tr>
<td>Robots</td>
<td>3.36</td>
<td>3.42</td>
</tr>
<tr>
<td>Kitchen</td>
<td>4.00</td>
<td>4.61</td>
</tr>
</tbody>
</table>

- Fairy Forest 174K 1L
- Robots 72K 3L/1 B
- Kitchen 111K 1L/4 B
- 1024x1024
Result 2 - Particle-Based Fluid

- Compared with CUDA SDK implementation
- Only rewrite simulation module
- 1 hour coding with 154 lines
- Better kernels/stream management than un-optimize CUDA

<table>
<thead>
<tr>
<th>FPS</th>
<th>Lines of Code (+GUI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>BSGP</td>
</tr>
<tr>
<td>187</td>
<td>290</td>
</tr>
</tbody>
</table>
Result 3 - X3D Parser

- Next Generation for VRML
- GPU-based has advantage
  - Data flow / Bandwidth
- Difficult on CUDA
  - Incremental testing
  - Temporary stream Management
  - Requires lots of modules
- Fair comparison?

<table>
<thead>
<tr>
<th>Scene</th>
<th>$T_{total}$ (ms)</th>
<th>$T_{I/O}$ (ms)</th>
<th>$T_{Parse}$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flux</td>
<td>BSGP</td>
<td>Flux</td>
</tr>
<tr>
<td>Paladin</td>
<td>2948</td>
<td>183</td>
<td>132</td>
</tr>
<tr>
<td>Scene</td>
<td>836</td>
<td>609</td>
<td>586</td>
</tr>
</tbody>
</table>
Result 4 - Adaptive Tessellation

1. View culling
   - thread.kill
2. View-dependent factor
3. Compute output size
4. Allocate memory
5. Tessellate triangles
   - thread.fork

<table>
<thead>
<tr>
<th>View</th>
<th>w/ out Management</th>
<th>w/ Management</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{\text{tess}}$(ms)</td>
<td>FPS</td>
</tr>
<tr>
<td>Side</td>
<td>43.9</td>
<td>21</td>
</tr>
<tr>
<td>Top</td>
<td>5</td>
<td>144</td>
</tr>
</tbody>
</table>
Outline

- Background / Introduction
- Example / Comparison with CUDA
- Basic Construct / Syntax
- Compilation Issues
- Results
- Conclusion and Future Work
Conclusion and Future Work

- Conclusion
  - 1st GPU BSP Model
  - Program is easy to read/write/maintain
    - Automatic data management
  - Better performance
  - Incremental development

- Future Work
  - Debugger
  - AMD/ATI’s CTM (Close to the Metal)
The End, Thank You

- Questions?