Problems with Brook/CUDA/etc

- Supplies high performance, but makes GPU programming hard

```
Supplies high performance, but makes GPU programming hard

Programmer

CUDA/Brook/... Compiler

Kernel 1   Kernel 2

Kernel ...

Kernel n

Dataflow Management

GPU

```
Problems with Brook/CUDA/etc

• Supplies high performance, but makes GPU programming hard
  – Program readability and maintenance
    • Bundle independent processes to reduce temporary streams and kernel launches
  – Manual dataflow management
    • Recycle temporary streams
  – Inefficient code reuse
    • Primitives with broken integrity
What’s BSGP?

- A C-like GPU programming language
  - Like sequential programs

```
spawn(n){
    v = a(id);
    scan(v);
    barrier;
    p[id] = v;
}
```
BSGP Features

• Programmer specifies barriers, compiler automatically deduces supersteps.
BSGP Features

• Programmer specifies *barriers*, compiler automatically deduces *supersteps*

• Implicit data dependencies through local variables
**BSGP Features**

- Programmer specifies barriers, compiler automatically deduces supersteps
- Implicit data dependencies through local variables
- Allows collective operation
  - Parallel primitives are called as a whole in a single statement

```
Thread 0
  Rank 0
  x  2

Thread 1
  Rank 1
  x  4

Thread 2
  Rank 2
  x  3

2 4 3
+  +  +
0 2 6 9
scan(x)
```
Example: one-ring neighborhood

• Compute the one-ring neighboring triangles of each vertex of a triangular mesh

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$t_1, t_2, t_3, t_4, t_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_2$</td>
<td>$t_4, t_5, t_6, t_7, t_8, t_9$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>......</td>
</tr>
</tbody>
</table>
One-ring neighborhood: BSGP version

```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;
    }
}
```

- Sorting the triplicated triangles
- Compute each vertex’s head pointer
One-ring neighborhood: CUDA version

Kernels

```c
__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&0xfffff);
        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;
    }
}
```

Dataflow management

```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* 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;
        spdatatype = 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);
}
```
CUDA: explicit dataflow

```c
__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;
    }
}
```

```c
__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&0xffffffff);
        owner[rk]=(k>>16);
    }
}
```

```c
__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;
    }
}
```

```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* 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;
        scanf.config.direction = CUDPP_SCAN_FORWARD;
        scanf.config.exclusivity = CUDPP_SCAN_EXCLUSIVE;
        scanf.config.maxNumElements = n3;
        scanf.config.maxNumRows = 1;
        scanf.config.datatype = CUDPP_UINT;
        scanf.config.op = CUDPP_ADD;
        scanf.initializeScan(&scanfconfig);
        scanfconfig = &scanfconfig;
        cudppSort(sorted, key, temp1, temp2, &sp, 0);
        cudppFinalizeScan(sp, scanfConfig);
    }
    after_sort<<<ng,szblock>>>(pf,temp1,sorted,n3);
    make_head<<<ng,szblock>>>(hd,temp1,n3);
    cudaFree(temp2);
    cudaFree(temp1);
    cudaFree(sorted);
    cudaFree(key);
}
```
BSGP: implicit dataflow

```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;
    }
}
```
CUDA: inefficient code reuse

Kernels

```c
_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&0xfffff);
        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;
    }
}
```

Dataflow management

```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* 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);
}
```
CUDA: inefficient code reuse

```c
__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 & 0xfffff);
        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;
    }
}
```

Kernels

```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* 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 CUDA P 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);
    cudppFree(temp1);
}
```

local_sort(key) {
    ...
}

global merge steps
CUDA: inefficient code reuse

Kernels

```c
__global__ void
before_sort(unsigned int* key, int* ib, int n3){
    int rk=blockIdx.xyszblock+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.xyszblock+threadIdx.x;
    if(rk<n3){
        int k=sorted[rk];
        pf[rk]=(k&0xffffffff);
        owner[rk]=(k>>16u);
    }
}

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

cudppSort

```c
local_sort(key){
    ...
}
```

global merge steps

```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 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);
        cudppScanConfig = &scanconfig;
        cudppSort(sorted, key, temp1, temp2, &sp, 0);
        cudppFinalizeScan(sp, scanconfig);
    }

    after_sort<<<ng,szblock>>>(pf,temp1,sorted,n3);
    makesafe<<<ng,szblock>>>(hd,temp1,n3);
    cudppFinalize(temp1);
```
CUDA: inefficient code reuse

---

**Kernels**

```c
__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&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;
        }
    }
}
```

---

```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* 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));
    //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.scanconfig = CUDPP_UINT;
    scanconfig.op = CUDPP_ADD;
    cudppInitializeScan(scanconfig);
    cudppSort(sorted, key, temp1, temp2, &sp, 0);
    cudppFinalizeScan(scanconfig);
    after_sort<<<ng,szblock>>>(pf,temp1,sorted,n3);
    cudaFree(key);
    cudaFree(sorted);
    cudaFree(temp1);
    cudaFree(temp2);
}
```
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: efficient code reuse

```c
findFaces (int* pf, int* hd, int* ib, int n) {
    spawn (n*3) {
        rk = thread.rank;
        f = rk/3;
        v = ib [rk];
        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: efficient code reuse

findFaces (int* pf, int* hd, int* ib, int n) {
    spawn(n*3) {
        rk = thread.rank;
        f = rk/3;
        v = ib[rk];
        local_sort(key);
        barrier;
        global merge steps
    }
}

BSGP: efficient code reuse

```c
findFaces(int* pf, int* hd, int* ib, int n){
  spawn(n+3){
    rk = thread.rank;
    f = rk/3;
    v = ib[rk];
    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;
  }
}

local_sort(key);
barrier;
global merge steps

Bundled into one kernel automatically
```
Why BSGP?

• Easy to read, write and maintain
• Similar or better performance than native languages
  – i.e., CUDA...
• Complex programs
  – i.e., X3D parser
BSGP Language Constructs

- spawn and barrier
- Insert CPU code: require
- Thread manipulation: fork and kill
- Communication: thread.get and thread.put
- Reducing barriers: par
- Parallel primitive operations, including reduce, scan and sort
BSGP Language Constructs

- `spawn` and `barrier`

- Insert CPU code: `require`

- Thread manipulation: `fork` and `kill`

- Communication: `thread.get` and `thread.put`

- Reducing barriers: `par`

- Parallel primitive operations, including `reduce`, `scan` and `sort`

```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

• **spawn** and **barrier**

• Insert CPU code: **require**

• Thread manipulation: **fork** and **kill**

• Communication: **thread.get** and **thread.put**

• Reducing barriers: **par**

• Parallel primitive operations, including **reduce**, **scan** and **sort**

```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

- **spawn** and **barrier**
- Insert CPU code: **require**

```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

- **spawn** and **barrier**
- Insert CPU code: **require**
- **Thread manipulation:** **fork** and **kill**
- **Communication:** `thread.get` and `thread.put`
- **Reducing barriers:** `par`

Parallel primitive operations, including `reduce`, `scan` and `sort`
BSGP Language Constructs

- **spawn** and **barrier**
- Insert CPU code:
- Thread manipulation:
- Communication: **thread.get** and **thread.put**
- Reducing barriers:
- Parallel primitive operations, including **reduce**, **scan** and **sort**

```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

- Reducing barriers: **par**
- Parallel primitive operations, including reduce, scan and sort
BSGP Language Constructs

- spawn and barrier
- Insert CPU code: require
- Thread manipulation: fork and kill
- Communication: thread.get and thread.put
- Reducing barriers: par
- Parallel primitive operations, including reduce, scan and sort
BSGP Compiler Design

- Emulate persistent thread context
  - Add context saving code
  - Only save values used across supersteps
- Minimize peak memory consumption
  - Using graph optimization in polynomial time
BSGP Compilation Algorithm

Inline function

Optimize

Liveness analysis

Final stream program

Generate kernels and kernel launching code

Generate temporary stream management

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

- Use SSA as intermediate form
- Compile each spawn block’s SSA form
- Generate kernels in CUDA assembly code
- Apply CUDA assembler to get binary code
- Insert binary code into CPU code as a constant array
- Generate object file/executable by a conventional CPU compiler
Sample Applications

- Recursive ray tracer
- Particle simulation
- X3D Parser
- Adaptive tessellation
Recursive Ray Tracer

- Both BSGP and CUDA are implemented and optimized by the same programmer
Recursive Ray Tracer

- 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>
Particle Simulation

- CUDA SDK demo
- Rewrote simulation module in BSGP, reused GUI code
Particle Simulation

- CUDA SDK demo
- Rewrote simulation module in BSGP, reused GUI code
- Simpler and faster

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>BSGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render fps</td>
<td>187</td>
<td>290</td>
</tr>
<tr>
<td>Module lines</td>
<td>-</td>
<td>154</td>
</tr>
<tr>
<td>Total lines</td>
<td>2113</td>
<td>1579</td>
</tr>
<tr>
<td>Coding time</td>
<td>-</td>
<td>1 hour</td>
</tr>
</tbody>
</table>

- Integration and sort preparation aren’t bundled
- Sort isn’t bundled with sort preparation
- Sort calls unbundled scan
X3D Parser

- BSGP implementation
  - Incremental development
  - 16 GPU functions, compiled into 82 kernels, 19k lines of assembly
  - 15x faster than CPU parser
- Extremely difficult in CUDA

An 7.03MB X3D scene
Loaded in 183ms
Adaptive Tessellation

- A displacement map based terrain renderer
Adaptive Tessellation

- Without thread manipulation
  - Parallelized over all input triangles
- With thread manipulation
  - Parallelized over output vertices using `thread.fork`

<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>$T_{\text{tess}}$</td>
<td>FPS</td>
<td>$T_{\text{tess}}$</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>

2x~10x speedup
Try BSGP Now!

- BSGP compiler, primitive library, editor and all example code

- http://www.kunzhou.net/#BSGP