BSGP: Bulk-Synchronous GPU Programming

Kun Zhou

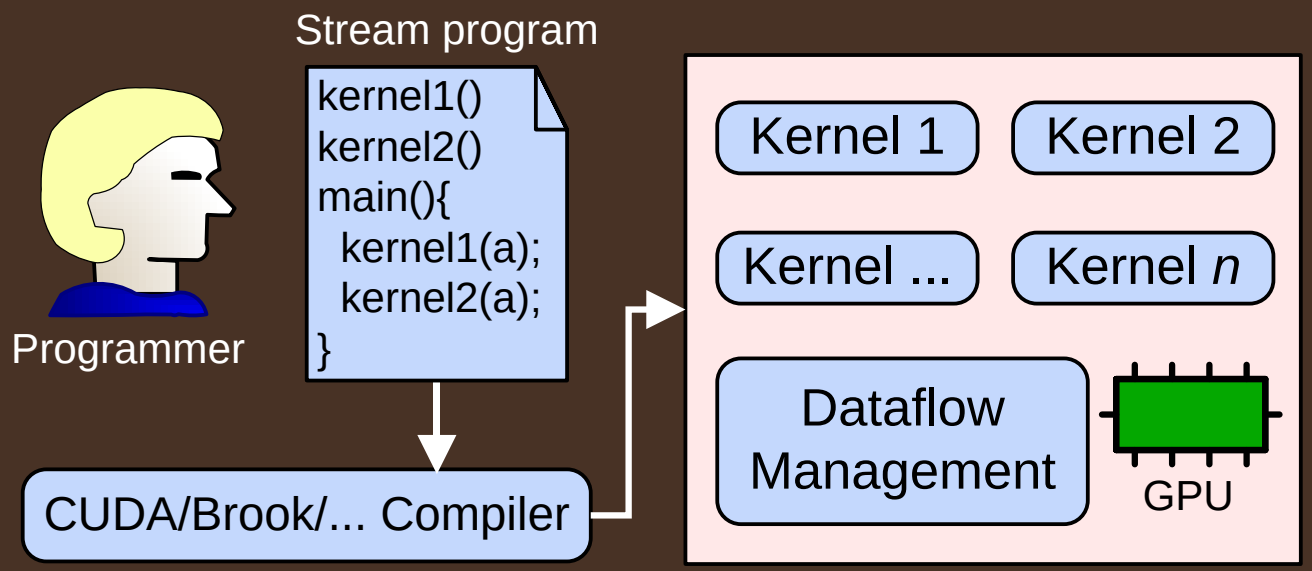
Zhejiang University

SIGGRAPHASIA2008

NEW HORIZONS

Problems with Brook/CUDA/etc

- Supplies high performance, but makes GPU programming hard

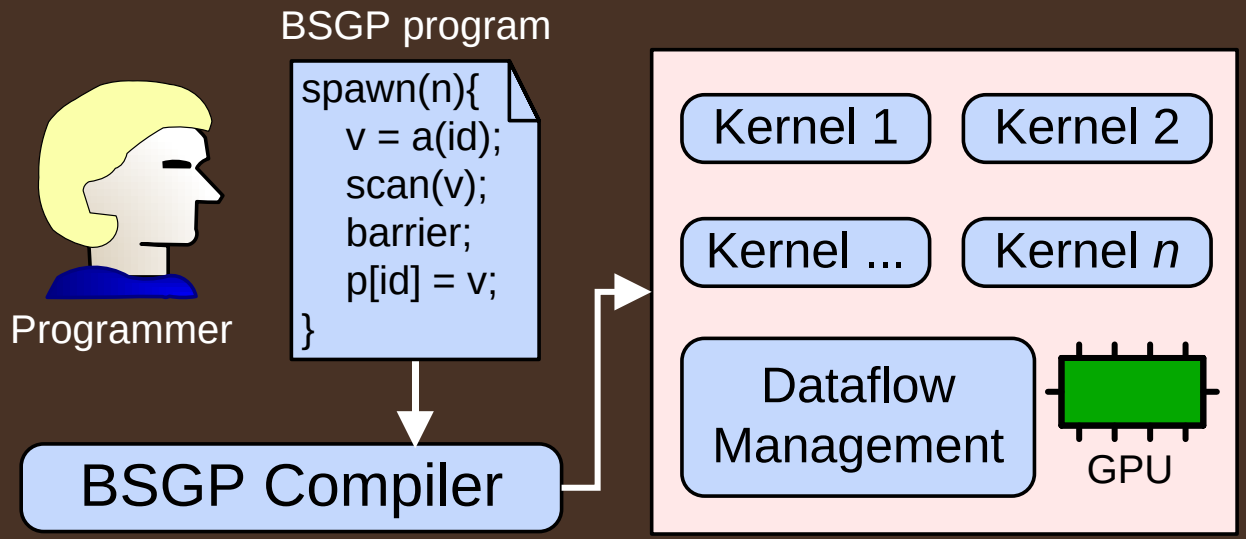


Problems with Brook/CUDA/etc

- 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

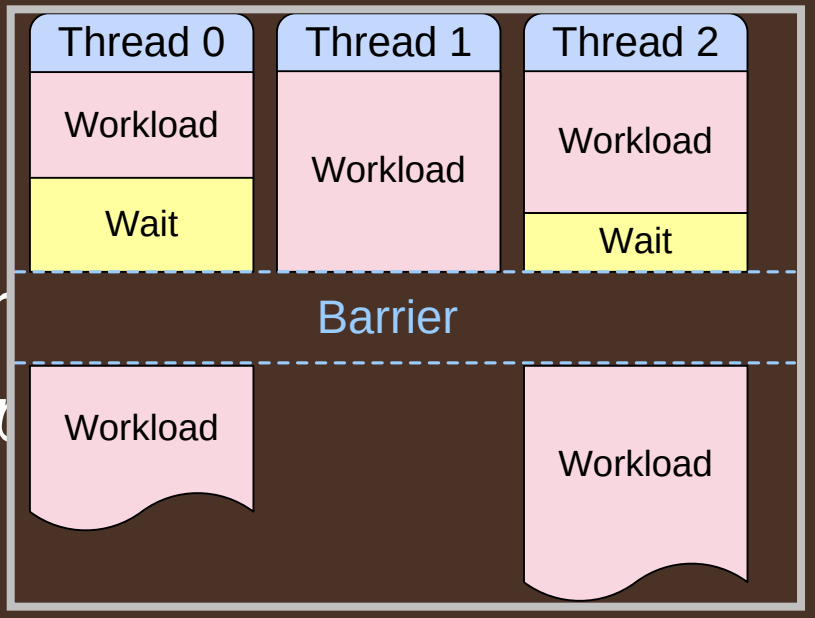
What's BSGP?

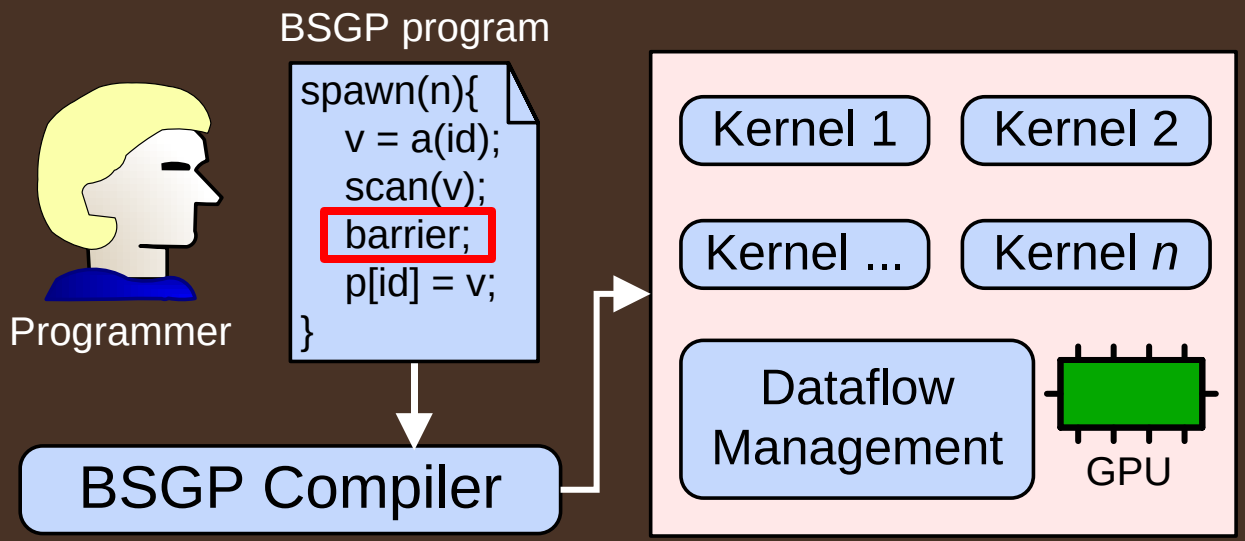
- A C-like GPU programming language
  - Like sequential programs



BSGP Features

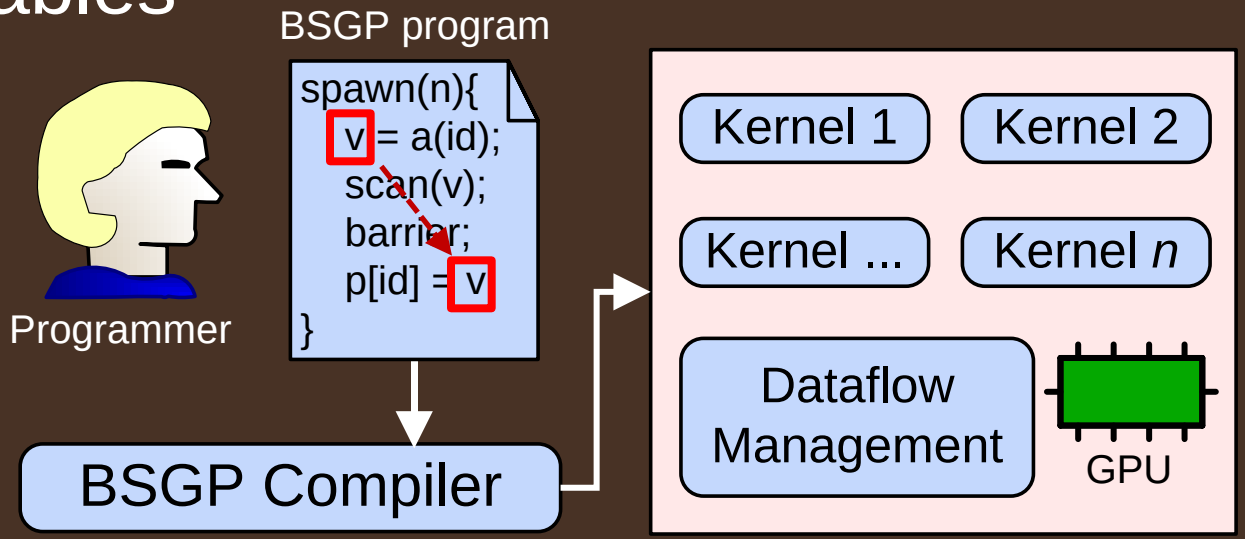
- Programmer specifies *bar*
  automatically deduces *sup*





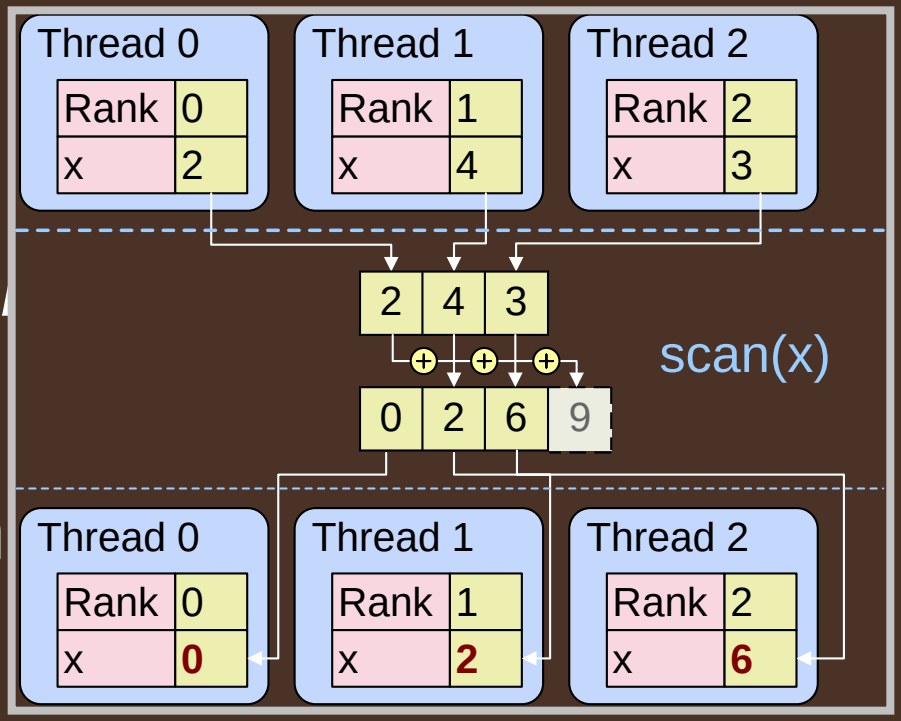
BSGP Features

- Programmer specifies *barriers*, compiler automatically deduces *supersteps*
- Implicit data dependencies through local variables



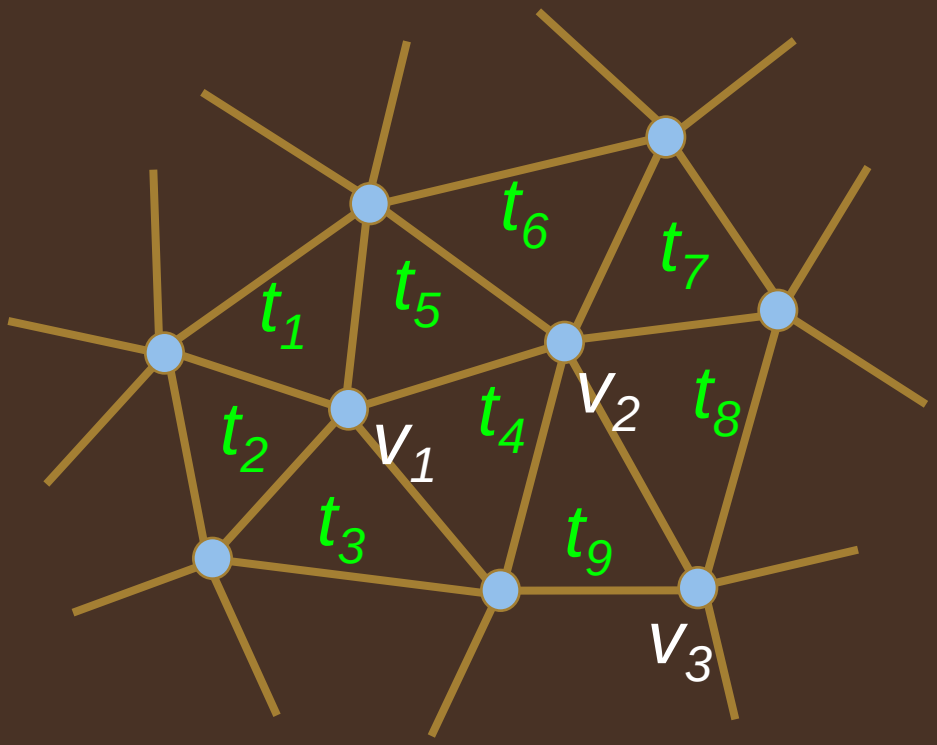
BSGP Features

- Programmer specifies A
  automatically deduces
- Implicit data dependen
  variables
- Allows *collective operation*
  - Parallel primitives are called as a whole in a single statement



Example: one-ring neighborhood

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



v_1	t_1 , t_2 , t_3 , t_4 , t_5
v_2	t_4 , t_5 , t_6 , t_7 , t_8 , t_9
v_3	……

One-ring neighborhood: BSGP version

```
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

```
__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;
    }
}
```

Kernels

```
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);
}
```

Dataflow management

CUDA: explicit dataflow

```
__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;
    }
}
```

Kernels

```
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);
}
```

Dataflow management

BSGP: implicit dataflow

```
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

```
__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;
    }
}
```

Kernels

```
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);
}
```

Dataflow management

CUDA: inefficient code reuse

```
__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;
    }
}
```

Kernels

```
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(temp1);
}
```

cudppSort

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

```
global merge steps
```

CUDA: inefficient code reuse

```
__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;
    }
}
```

Kernels

```
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(temp1);
}
```

cudppSort

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

```
global merge steps
```

CUDA: inefficient code reuse

```
__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;
    }
}
```

Kernels

```
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(temp1);
}
```

cudppSort

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

```
global merge steps
```

BSGP: efficient code reuse

```
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

```
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 lis
        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;
    }
}
```

```
thread.sortby
local_sort(key);
barrier;
global merge steps
```

BSGP: efficient code reuse

```
findFaces(int* pf, int* hd, int* ib, int n){
    spawn(n*3){
        rk = thread.rank;
        f = rk/3;                    //face id
        v = ib[rk];                  //
        thread.sortby(v);
        //allocate a temp lis
        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
```

BSGP: efficient code reuse

```
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 lis
        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;
    }
}
```

Bundled into one kernel automatically

```
local_sort(key);
barrier;
global merge steps
```

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**
- T… till
- C… nd t…
- R…
- P… uding re…

```
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**
- T… till
- C… nd t…
- R…
- P… uding re…

```
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**
- T… till
- C… nd t…
- R…
- P… uding re…

```
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**
- Con
  thr
- Red
- Para
  red

```
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;
}
```

ng

BSGP Language

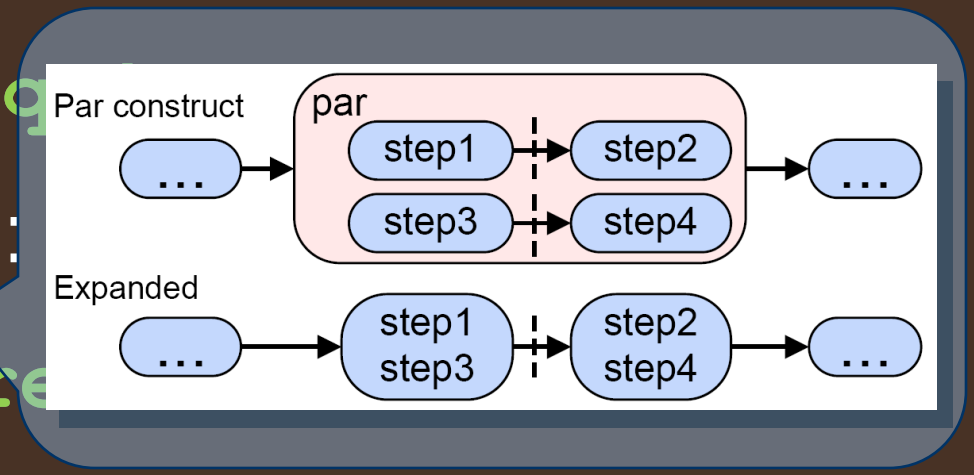
- **spawn** and **barr**
- Insert CPU code:
- Thread manipulat
- Communication: **thread.get** and **thread.put**
- Reducing barriers
- Parallel primitive **reduce**, **scan** a

```
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;
    }
}
```

```
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;
    }
}
```

BSGP Language Constructs

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



- Reducing barriers: **`par`**
- Parallel primitive operations, including **`reduce`**, **`scan`** and **`sort`**

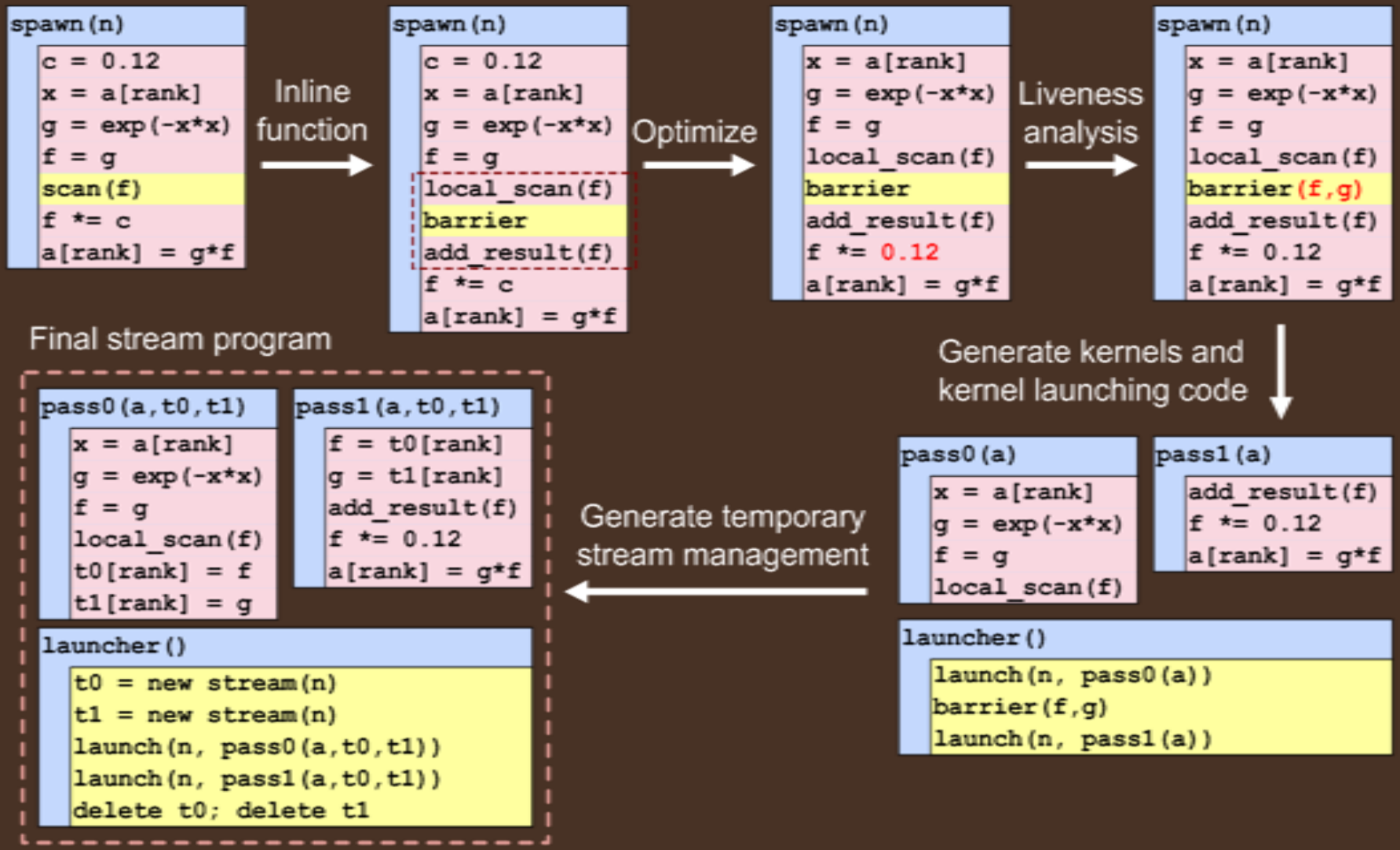
BSGP Language Constructs

- **`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

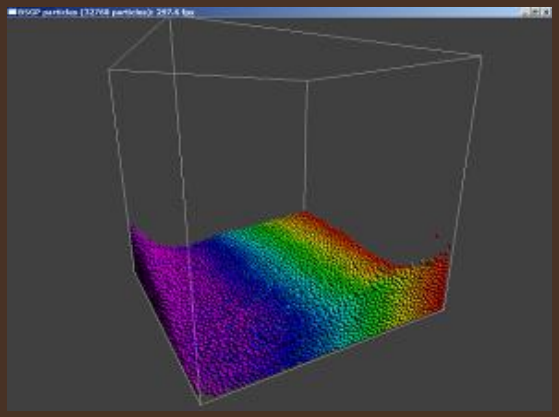


BSGP Compiler Implementation

- 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

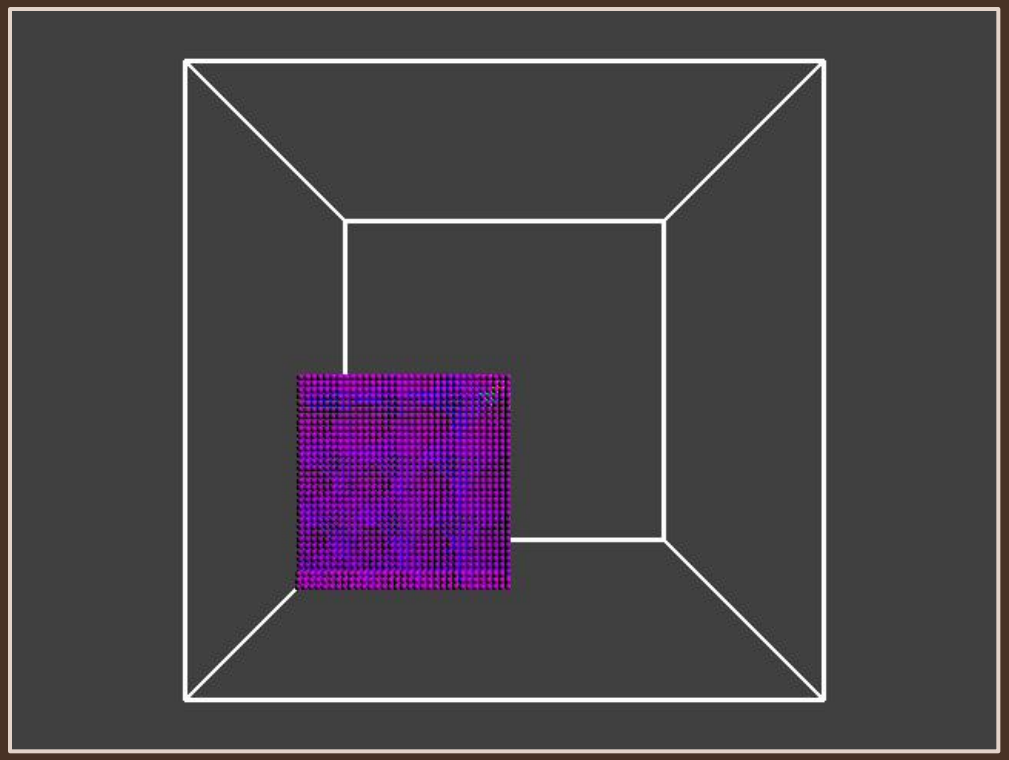
Recursive Ray Tracer

- Both BSGP and CUDA are Implemented and optimized by the same programmer
- Clear advantage in code complexity
- Similar performance and memory usage

	CUDA	BSGP
Render fps	4.00	4.61
Mem usage	144M	150M
Code lines	815	475
# GPU funcs	10	3
Coding days	2~3	1
Tuning days	4~5	2~3

Particle Simulation

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



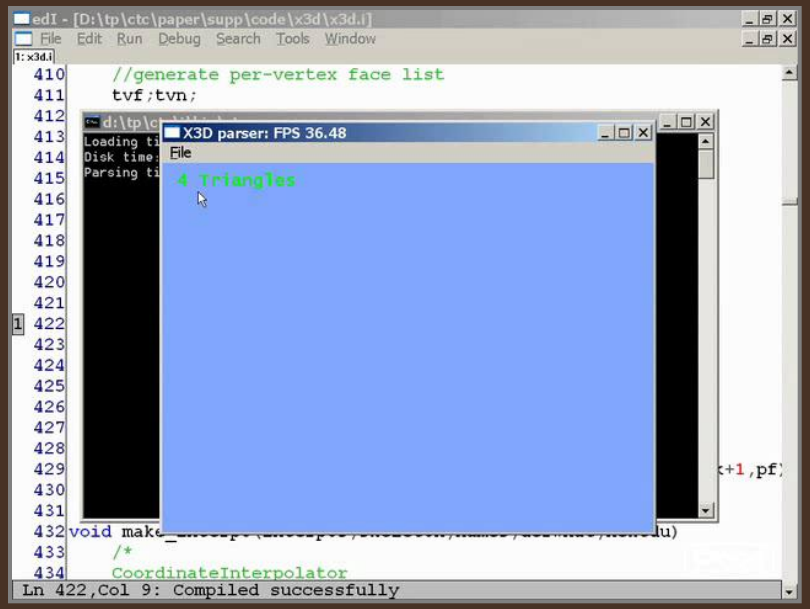
Particle Simulation

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

	CUDA	BSGP
Render fps	187	290
Module lines	-	154
Total lines	2113	1579
Coding time	-	1 hour

- Integration and sort preparation aren't bundled
- Sort isn't bundled with sort preparation
- Sort calls unbundled scan

X3D Parser



An 7.03MB X3D scene
Loaded in 183ms

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

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`

View	no thread man.		with thread man.		# vert output
	T_{tess}	FPS	T_{tess}	FPS	
Side	43.9ms	21.0	3.62ms	142	1.14M
Top	5.0ms	144	2.1ms	249	322k

2x~10x speedup

Try BSGP Now!

- BSGP compiler, primitive library, editor and all example code
- *http://www.kunzhou.net/#BSGP*