Automatic Generation of Warp-Level Primitives and Atomic Instruction for Fast and Portable Parallel Reduction on GPU

Simon Garcia De Gonzalo and Sitao Huang (University of Illinois at Urbana–Champaign), Juan Gomez-Luna (Swiss Federal Institute of Technology(ETH) Zurich), Simon Hammond (Sandia National Laboratories), Onur Mutlu (Swiss Federal Institute of Technology (ETH) Zurich), Wen-mei Hwu (University of Illinois at Urbana–Champaign)
Motivation

- HPC and Cloud is heavily dominated by Graphics Processing Units.

- GPU programming strategies:
  - APIs: CUDA, OpenCL
  - Libraries: Thrust, CUB
  - High-level frameworks: Kokkos, Raja, Tangram
  - DSLs: Halide, Lift, Pencil

- All of the above need to deal with GPU performance portability
  - Implementation of atomic operations
  - Evolving Instruction set architecture (ISA)

- Low-level architecture differences have a direct effect on what software algorithm is optimal for a particular problem.
Tangram

- Kernel Synthesis framework for Performance Portable Programming
  - Provide representations for SIMD utilization (Vector primitive)
  - Provide an architectural hierarchy model to guide composition
  - Exhaustive space exploration, tuning parameters
  - Targets multiple backend
The Problem: Parallel Reduction

• Fundamental building block
  • Histogram
  • Scan
  • and many more

• Performance is heavily depends on hand-written code that takes advantage of the latest hardware improvements
Current Approaches

- Library developers:
  - Constantly adapting and upgrading
  - Backward compatible by preserving previous implementations
    - An ever expanding code-base

- Kokkos and Raja:
  - In-house or third-party library (Hand written)

- DSLs:
  - Abstraction for low level GPU instruction
  - Single source code different optimization (Tiling, unrolling, etc..)
  - Lack support for atomics operation on different memory spaces (Global vs Shared)
Contributions

- Addition of new Tangram extensions and AST transformations

- Automatic generation of:
  - Global memory atomics
  - Shared memory atomics
  - Warp-shuffle instruction

- Compare against other approaches:
  - Hand-written (Nvidia CUB)
  - High-Level frameworks (Kokkos)
  - OpenMP 4.0
Tangram: Parallel Reduction

(a) Atomic autonomous codelet

```c
int sum(const Array<1,int> in) {
    unsigned len = in.Size();
    int accum = 0;
    for(unsigned i=0; i < len; i += in.Stride())
        accum += in[i];
    return accum;
}
```

(b) Compound codelet

```c
int sum(const Array<1,int> in) {
    __tunable unsigned p;
    unsigned len = in.Size();
    unsigned tile = (len+p-1)/p;
    Sequence start();
    Sequence end();
    Sequence inc();
    Map map(sum, partition(in, p, start, inc, end));
    map.atomicAdd();
    return sum(map);
}
```

(c) Atomic cooperative codelet

```c
int sum(const Array<1,int> in) {
    Vector vthread();
    __shared int partial[vthread.MaxSize()];
    __shared int tmp[in.Size()];
    int val = 0;
    val += (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
    tmp[vthread.ThreadId()] = val;
    for(int offset = vthread.MaxSize(); offset > 0; offset /= 2)
        val += (vthread.LaneId() + offset < vthread.Size()) ?
            tmp[vthread.ThreadId()+offset] : 0;
    return val;
}
```

Tangram Reduction Codelets
Distribution

Tiled  Strided

Compute

Scalar  Parallel...

Architecture Hierarchy

Grid

Tiled

Block

Kernel 2

Thread

Scalar

Parallel...

Tiled  Parallel

Scalar
Distribution
Tiled
Strided

Compute
Tiled
Scalar
Parallel

Architecture Hierarchy
Grid
Tiled

Block
Tiled
Strided

Thread
Scalar
Parallel

Parallel...
GPU ISA and Microarchitecture Support

- **Fermi (2010)**
  - Fast Global memory Atomics
  - Warp Shuffle instructions

- **Kepler (2012)**
  - Fast Share memory atomics

- **Maxwell (2014)**
  - Wider type support for atomics operation
  - Scope for atomics

- **Pascal (2016)**
(a) Atomic autonomous codelet

```cpp
1._codelet
2    int sum(const Array<1,int>& in) {  
3        unsigned len = in.size();
4        int accu = 0;
5        for(unsigned i=0; i < len; in.stride()) {  
6            accu += in[i];
7        }
8        return accu;
9    }
```

(b) Compound codelet

```cpp
1._codelet
2    int sum(const Array<1,int>& in) {  
3        unsigned len = in.size();
4        unsigned tile = (len+p-1)/p;
5        Sequence start(...);
6        Sequence end(...);
7        Sequence inc(...);
8        Map map(sum, partition(in, p, start, inc, end));
9    return sum(map);
10  }
```

(c) Atomic cooperative codelet

```cpp
1._codelet
2    int sum(const Array<1,int>& in) {  
3        unsigned p = vthread.maxsize();
4        Sequence start(...);
5        Sequence end(...);
6        Sequence inc(...);
7        Map map(sum, partition(in, p, start, inc, end));
8        map.atomicAdd();
9    return sum(map);
10   }
```

Tangram Reduction Codelets
Global Memory Atomics

```cpp
__codelet
int sum(const Array<1, int> in) {
    __tunable unsigned p;
    unsigned len = in.size();
    unsigned tile = (len+p-1)/p;
    Sequence start(...);
    Sequence end(...);
    Sequence inc(...);
    Map map(sum, partition(in, p, start, inc, end));
    map.atomicAdd();
    return sum(map);
}
```

(b) Compound codelet
Global Memory Atomics

```
void sum(const Array<1,int>& in) {
    unsigned len = in.Size();
    unsigned tile = (len+p-1)/p;
    Sequence start(...);
    Sequence end(...);
    Map map(sum, partition(in, p, start, inc, end));
    map.atomicAdd();
    return sum(map);
}
```

(b) Compound codelet

Listing 1. Standard Version
Global Memory Atomics

```c
__codelet
int sum(const Array<1,int>& in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  Sequence start(…);
  Sequence end(…);
  Sequence inc(…);
  Map map(sum,
         partition(in,
                    p,
                    start,
                    inc,
                    end));
  map.atomicAdd();
  return sum(map);
}
```

```
void Reduce.Thread(int* in, int* input, …){
  int p = blockDim.x;
  if (threadIdx.x == 0)
    map = new map();
  if (threadIdx.x == 0)
    map = new map();
  Reduce.Thread(map, input, …);
  if (threadIdx.x == 0)
    RETURN(map, …);
  __shared_ int map; atomcAdd();
}
```

**Listing 1. Standard Version**

**Listing 2. Global Atomics**

GPU ISA and Microarchitecture Support

- **Fermi (2010)**
  - Fast Global memory Atomics
  - Warp Shuffle instructions

- **Kepler (2012)**
  - Fast Share memory atomics

- **Maxwell (2014)**
  - Wider type support for atomics operation
  - Scope for atomics

- **Pascal (2016)**
(a) Atomic autonomous codelet

```c
int sum(const Array<1,int> in) {
    unsigned len = in.Size();
    int accum = 0;
    for(unsigned i=0; i < len; in.Stride()) {
        accum += in[i];
    }
    return accum;
}
```

(b) Compound codelet

```c
int sum(const Array<1,int> in) {
    unsigned len = in.Size();
    unsigned tile = (len+p-1)/p;
    Sequence start();
    Sequence inc();
    Map map(sum, partition(in, p, start, inc, end));
    return sum(map);
}
```

(c) Atomic cooperative codelet

```c
int sum(const Array<1,int> in) {
    Vector vthread();
    __shared int partial[vthread.MaxSize()];
    __shared int tmp[in.Size()];
    int val = 0;
    val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
    tmp[vthread.ThreadId()] = val;
    for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
        val = (vthread.LaneId() + offset < vthread.Size()) ?
        tmp[vthread.ThreadId()+offset] : 0;
        tmp[vthread.ThreadId()] = val;
    }
    if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
        if(vthread.LaneId() == 0)
            partial[vthread.VectorId()] = val;
        if(vthread.VectorId() == 0){
            val = (vthread.ThreadId() < (in.Size() / vthread.MaxSize())) ?
            partial[vthread.LaneId()] : 0;
            for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
                val = (vthread.LaneId() + offset < vthread.Size()) ?
                partial[vthread.ThreadId()+offset] : 0;
                partial[vthread.ThreadId()] = val;
            }
        }
    }
    return val;
}
```

Tangram Reduction Codelets
Shared Atomics

(a) Atomic for single accumulator

(b) Atomic for partial results
```c
__codelet __coop__ tag(shared_V2)
int sum(const Array<1,int>& in) {
Vector vthread();
  __shared atomicAdd int partial
  __shared int tmp[in.Size()];
int val = 0;
  val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
  tmp[vthread.ThreadId()] = val;
  for(int offset = vthread.MaxSize() / 2; offset > 0; offset /= 2) {
    val = (vthread.ThreadId() + offset) < vthread.Size() ?
      val + tmp[vthread.ThreadId() + offset] : 0;
    tmp[vthread.ThreadId()] = val;
  }
  if(in.Size() <= vthread.MaxSize() && in.Size() / vthread.MaxSize() > 0) {
    partial = val;
    if(vthread.VectorId() == 0)
      val = partial;
  }
  return val;
}
```

Listing 3. Shared atoms code for Figure 3(b)
GPU ISA and Microarchitecture Support

- **Fermi (2010)**
  - Fast Global memory Atomics
  - Warp Shuffle instructions

- **Kepler (2012)**
  - Fast Share memory atomics

- **Maxwell (2014)**
  - Wider type support for atomics operation
  - Scope for atomics

- **Pascal (2016)**
(a) Atomic autonomous codelet

```c
    __codelet
    int sum(const Array<1,int>& in) {
        unsigned len = in.Size();
        int accu = 0;
        for(unsigned i=0; i < len; i++) {
            accu += in[i];
        }
        return accu;
    }
```

(b) Compound codelet

```c
    __codelet
    int sum(const Array<1,int>& in) {
        unsigned len = in.Size();
        unsigned tile = (len+p-1)/p;
        Sequence start(...);
        Sequence end(...);
        Sequence inc(...);
        Map map(sum, partition(in, p, start, inc, end));
        return sum(map);
    }
```

(c) Atomic cooperative codelet

```c
    __codelet
    int sum(const Array<1,int>& in) {
        Vector vthread();
        __shared int partial[vthread.MaxSize()];
        __shared int tmp[in.Size()];
        int val = 0;
        val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
        tmp[vthread.ThreadId()] = val;
        for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
            val += (vthread.LaneId() + offset < vthread.Size()) ?
                   tmp[vthread.ThreadId()+offset] : 0;
            tmp[vthread.ThreadId()] = val;
        }
        if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
            if(vthread.LaneId() == 0)
                partial[vthread.VectorId()] = val;
            if(vthread.VectorId() == 0){
                val = vthread.ThreadId() < (in.Size() / vthread.MaxSize()) ?
                        partial[vthread.LaneId()] : 0;
                for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
                    val += (vthread.LaneId() + offset < vthread.Size()) ?
                           partial[vthread.ThreadId()+offset] : 0;
                }
                partial[vthread.ThreadId()] = val;
            }
        }
        return val;
    }
```

Tangram Reduction Codelets
Warpshuffle Instructions
Warp Shuffle Instructions

**Listing 4. Shuffle Output Code for Figure 1(c)**

```c
__global__
void ReduceBlockInt(int *in, int *out, int *SourceSize, int *ObjectSize)
{
    int threadIdx, blockDim_x, blockDim_y, blockDim_z;
    __shared__ int partial[blockDim_x * blockDim_y * blockDim_z];
    int *in = (int *) SourceSize;
    int *out = (int *) ObjectSize;
    int val = 0;
    int offset = blockDim_x / 2;
    int threadIdx = threadIdx.x;
    int laneId = threadIdx % blockDim_x;

    for (int i = threadIdx.x; i < blockDim_x; i += blockDim_x)
    {
        partial[i] = val;
    }

    val = threadIdx % blockDim_x;
    for (int i = threadIdx.x; i < blockDim_x; i += blockDim_x)
    {
        partial[i] = val;
    }

    for (int i = threadIdx.x; i < blockDim_x; i += blockDim_x)
    {
        partial[i] = val;
    }
```

1. Reading from __shared__ array, values reduced into a local accumulator
2. For loop bounds based on __base__ primitive and iterator decreases by a constant every iteration
3. Shared array index is a function of `VectorThreadId()` and for loop iterator
4. Accumulator value written to __same shared array
5. Accumulator value written to index that is only a function of `VectorThreadId`

---

**Warp Shuffle Instructions**

```c
__codelet
t sum(const Array<1,int> in) {
    Vector vthread();
    __shared__ int partial[vthread.MaxSize()];
    int val = 0;
    val += (vthread.ThreadId() + in.nnz()) ? in[vthread.ThreadId()] : 0;
    for (int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
        val += (vthread.LaneId() + offset < vthread.Size()) ?
            tmp[vthread.ThreadId()+offset] : 0;
        tmp[vthread.ThreadId()] = val;
    }
    return val;
}
```

1. Reading from __shared__ array, values reduced into a local accumulator
2. For loop bounds based on __base__ primitive and iterator decreases by a constant every iteration
3. Shared array index is a function of `VectorThreadId()` and for loop iterator
4. Accumulator value written to __same shared array
5. Accumulator value written to index that is only a function of `VectorThreadId`

---

**Listing 4. Shuffle Output Code for Figure 1(c)**
Search Space

Over 85 different versions possible!
Experimental Setup

3 GPU architectures
- Kepler (Titan, Blue Waters)
- Maxwell (Cloud)
- Pascal (Piz Daint)

Compare results against
- NVIDIA CUB 1.8.0
- Kokkos GPU backend
- OpenMP 4.0 reduce pragma
  - Dual-socket 8-core 3.5GHz POWER8+
  - gcc 5.4.0
Kepler Results

Graph showing speedup over CUB for different array sizes (Number of 32-bit Elements) ranging from 64 to 268,635,456. The graph includes various codelets and variants such as:
- Tangram Fig.6(p)
- Tangram Fig.6(m)
- Tangram Fig.6(b)
- Tangram Fig.6(e)
- Kokoos (GPU)
- OpenMP (CPU)
- CUB baseline (GPU)
Maxwell Results

![Graph and Diagram]

- **Speedup over CUB**
- **Array Size (Number of 32-bit Elements)**
- **Tangram Fig. 6(n)**
- **Tangram Fig. 6(p)**
- **Tangram Fig. 6(k)**
- **Tangram Fig. 6(c)**
- **Tangram Fig. 6(a)**
- **Kokkos (GPU)**
- **OpenMP (CPU)**
- **CUB baseline (GPU)**

**Codelets and Variants**
- Tile Distribute (Figure 1(b))
- Stride Distribute (Figure 1(b))
- Global Atomic Tile Distribute
- Global Atomic Stride Distribute
- Cooperative (Figure 1(c))
- Cooperative + Shuffle
- Shared Memory Atomic 1 (Figure 1(a))
- Shared Memory Atomic 2 (Figure 1(b))
- Shared Memory Atomic 2 + Shuffle
- Scalar (Figure 1(a))
Pascal Results

![Graph showing speedup over CUB for different array sizes and codelet variants.](Image)

- **Codelets and Variants**
  - Tangram Fig.6(n)
  - Tangram Fig.6(p)
  - Tangram Fig.6(e)
  - Kokkos (GPU)
  - OpenMP (CPU)
  - CUB baseline (GPU)

**Array Size (Number of 32-bit Elements)**

- 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304, 16777216, 67108864, 268435456

**Grids and Blocks**
- Grid
- Block
- Thread

**Variants**
- (a)
- (b)
- (c)
- (d)
- (e)
Future work

- **Volta**
  - 8 TCU units per SM are available
    - Reduction can be expressed as $M \times M$:
      - HPCaML’19 talk

- **CUDA dynamic parallelism**

- **Multi-GPU**
Conclusion

Introduce new high-level API and AST transformation

– Automatic generation of:
  • Warp-shuffle instruction
  • Global memory atomics
  • Shared memory atomics

Implementation of Parallel Reduction

– Compare against CPU OMP, Kokkos, and CUB Library
  • Up to 7.8X speedup (2X on average) over CUB