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)

ILLINOIS

ETH zürich

Sandia National Laboratories

Motivation

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

- GPU programing 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 operaton on different memory spaces (Global vs Shared)

Contributions

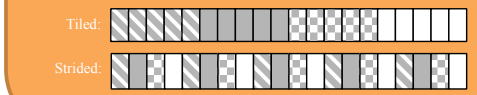
- Addition of new Tangram extensions and AST transformations
- Automatic generation of:
  - Global memory atomics
  - Shared memory atomics
  - Warp-shuffle instruction
- Compare against other approaches:
  - Hand-written (Nvidia CUB)
  - High-Level frameworks (Kokkos)
  - OpenMP 4.0

Tangram: Parallel Reduction

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   unsigned len = in.Size();
4   int accum = 0;
5   for(unsigned i=0; i < len; in.Stride()) {
6     accum += in[i];
7   }
8   return accum;
9  }
```

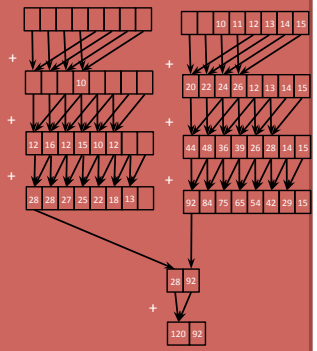
(a) Atomic autonomous codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3    __tunable unsigned p;
4   unsigned len = in.Size();
5   unsigned tile = (len+p-1)/p;
6   Sequence start(…);
7   Sequence end(…);
8   Sequence inc(…)
9   Map map( sum, partition(in, p, start, inc, end));
10
11  return sum(map);
12 }
```



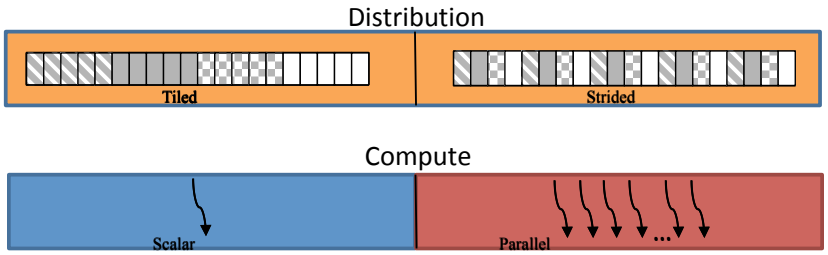
(b) Compound codelet

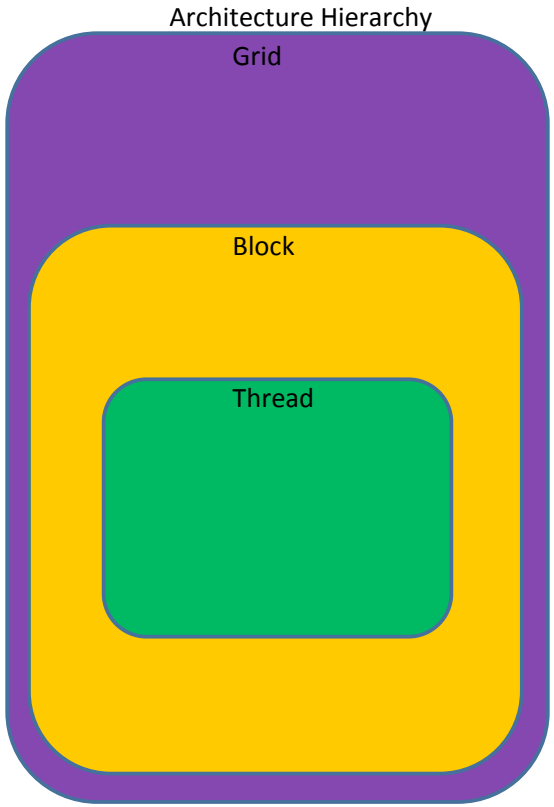
```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared int partial[vthread.MaxSize()];
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11       tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial[vthread.VectorId()] = val;
17    if(vthread.VectorId() == 0){
18      val = (vthread.ThreadId() <= (in.Size() / vthread.MaxSize())) ?
19       partial[vthread.LaneId()] : 0;
20      for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
21        val += (vthread.LaneId() + offset < vthread.Size()) ?
22       partial[vthread.ThreadId()+offset] : 0;
23        partial[vthread.ThreadId()] = val;
24      }
25    }
26  }
27  return val;
28 }
```

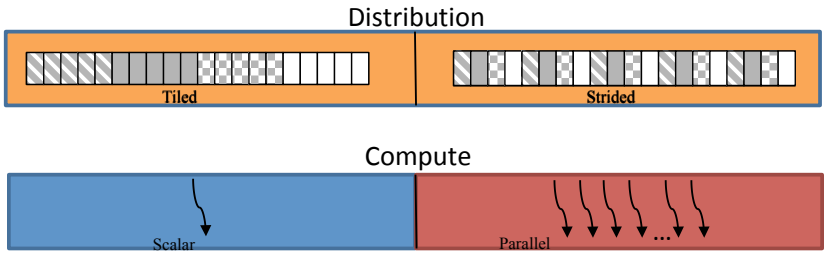


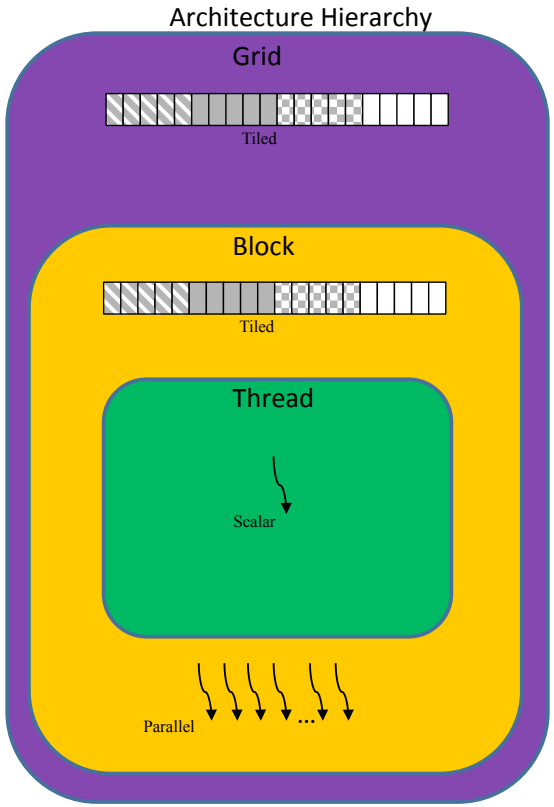
(c) Atomic cooperative codelet

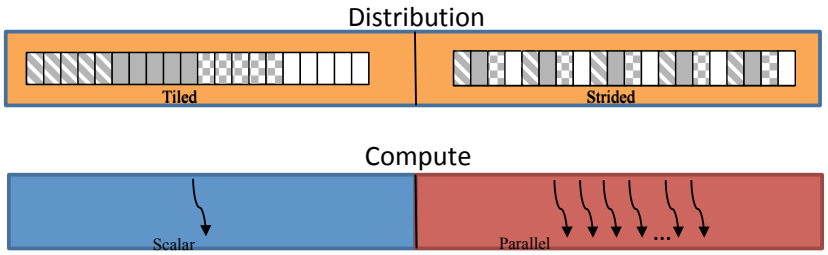
Tangram Reduction Codelets

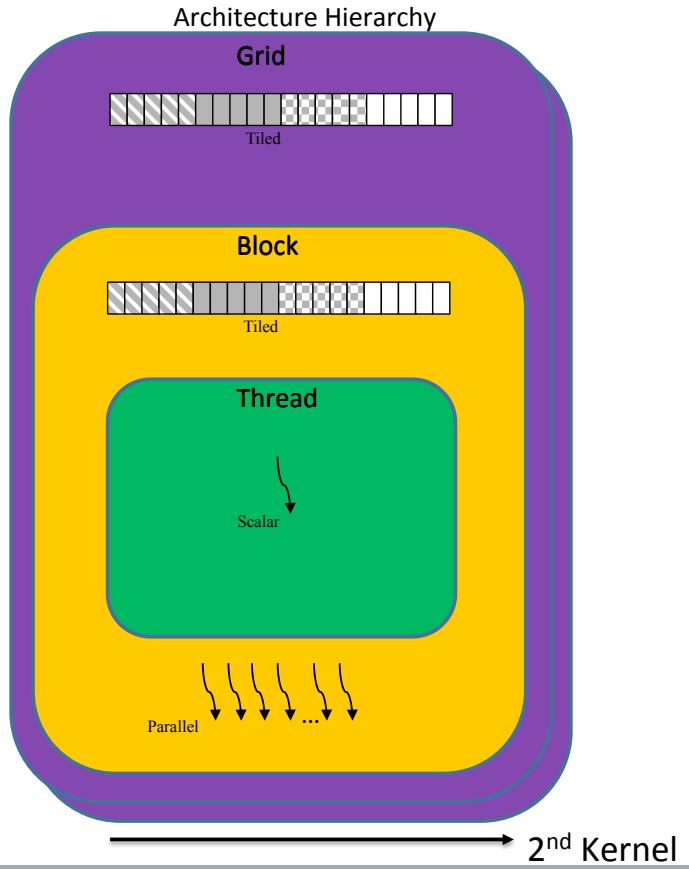

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...

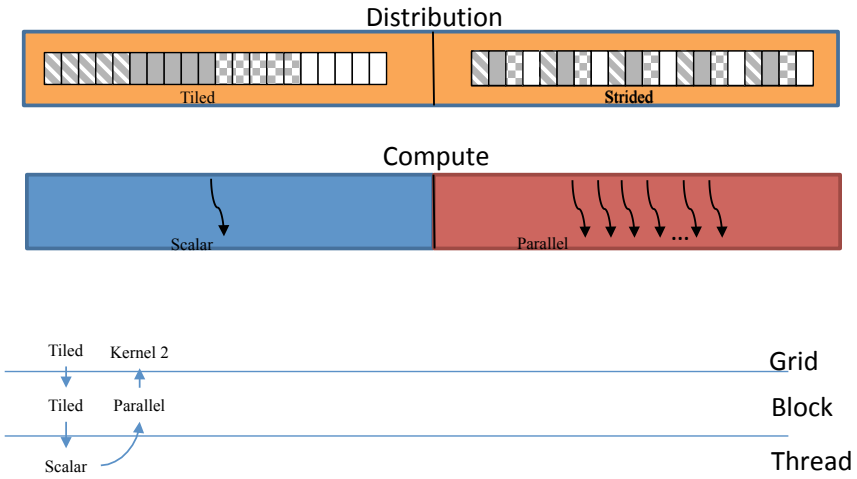

Architecture Hierarchy
Grid
Block
Thread

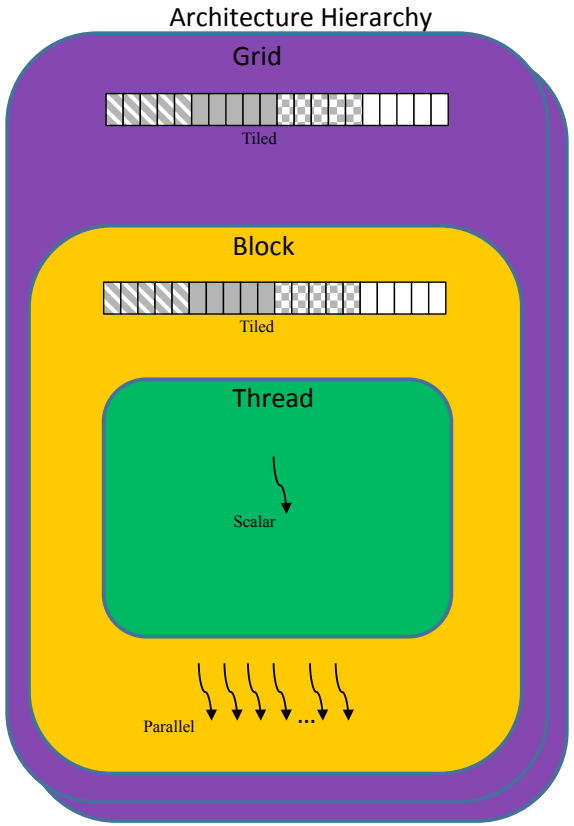

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...

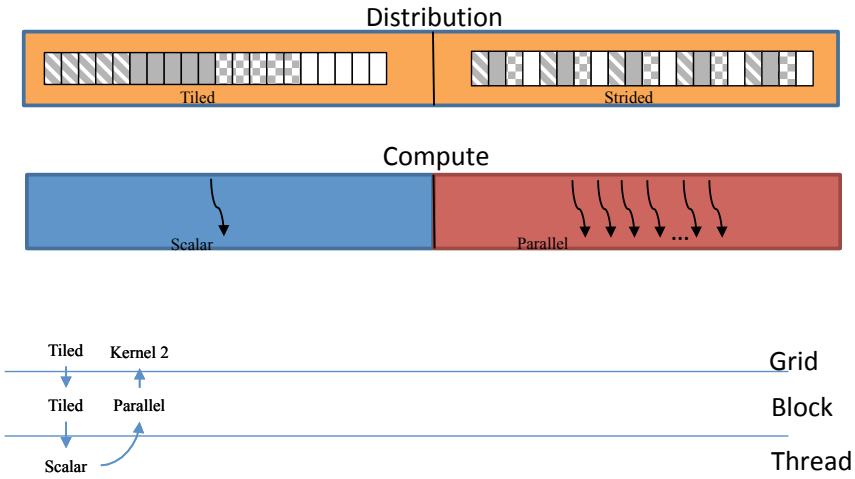

Architecture Hierarchy
Grid
Tiled
Block
Tiled
Thread
Scalar
Parallel
...

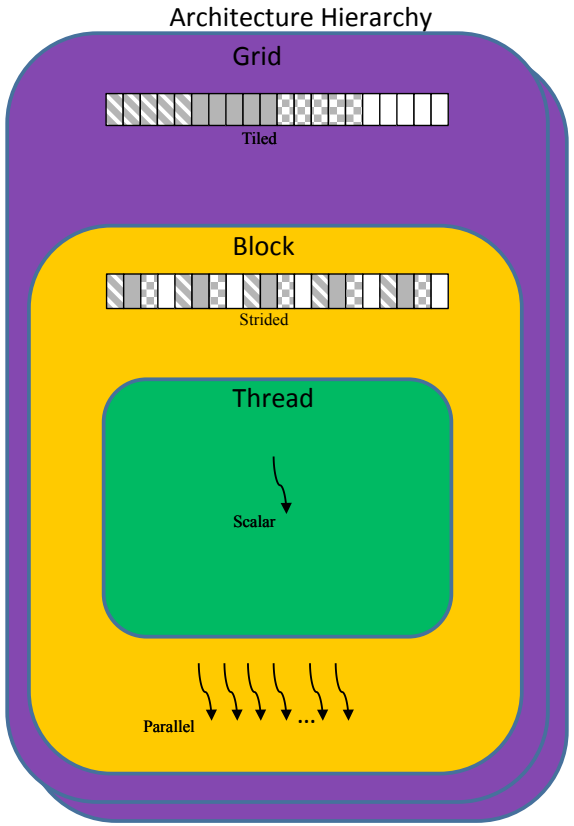

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...

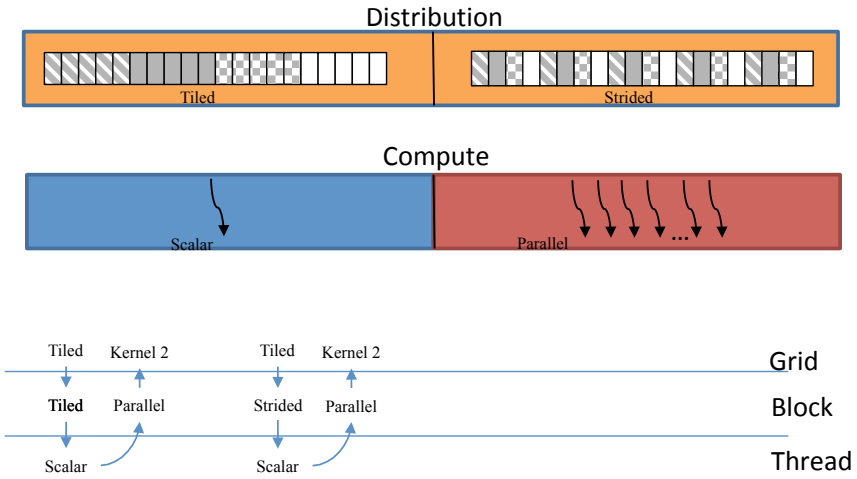

Architecture Hierarchy
Grid
Tiled
Block
Tiled
Thread
Scalar
Parallel
...
2nd Kernel

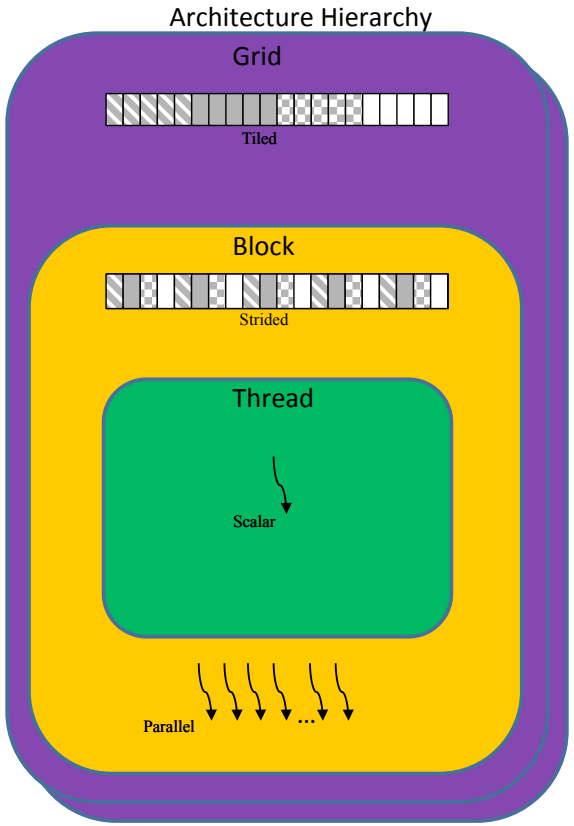

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...
Tiled
Kernel 2
Tiled
Parallel
Scalar
Grid
Block
Thread

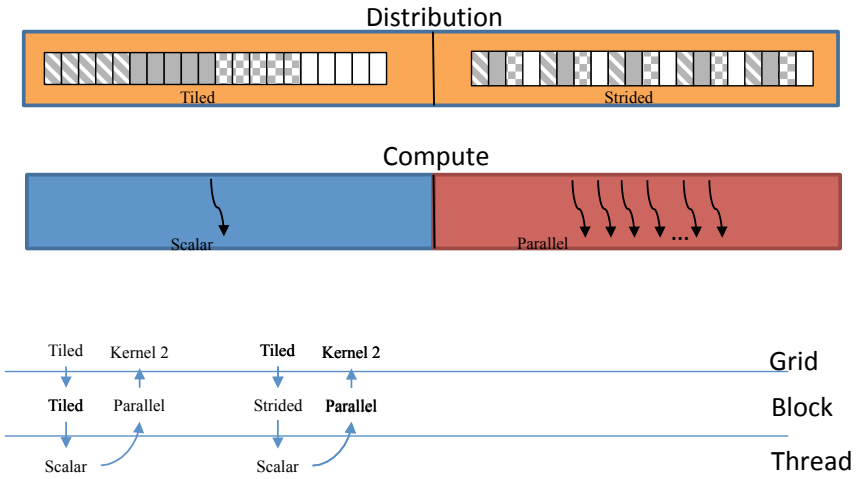

Architecture Hierarchy
Grid
Tiled
Block
Tiled
Thread
Scalar
Parallel
...

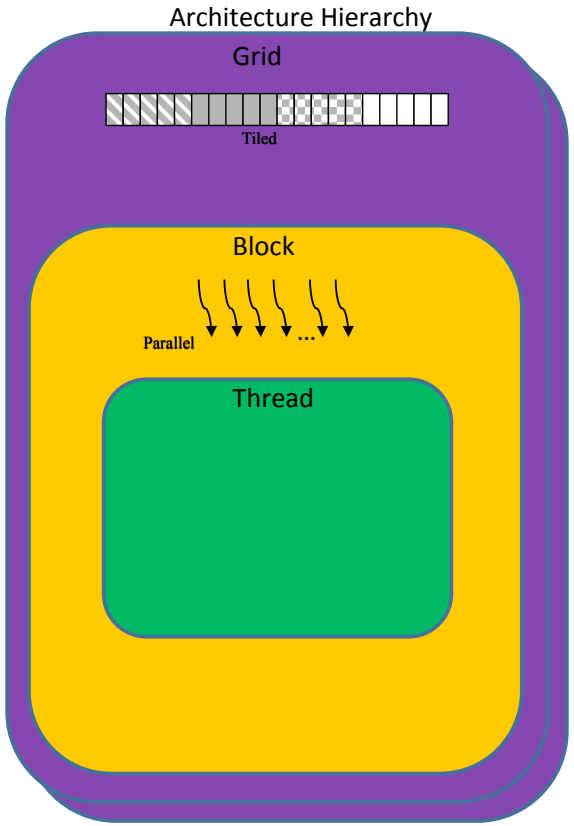

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...
Tiled
Kernel 2
Tiled
Parallel
Scalar
Grid
Block
Thread

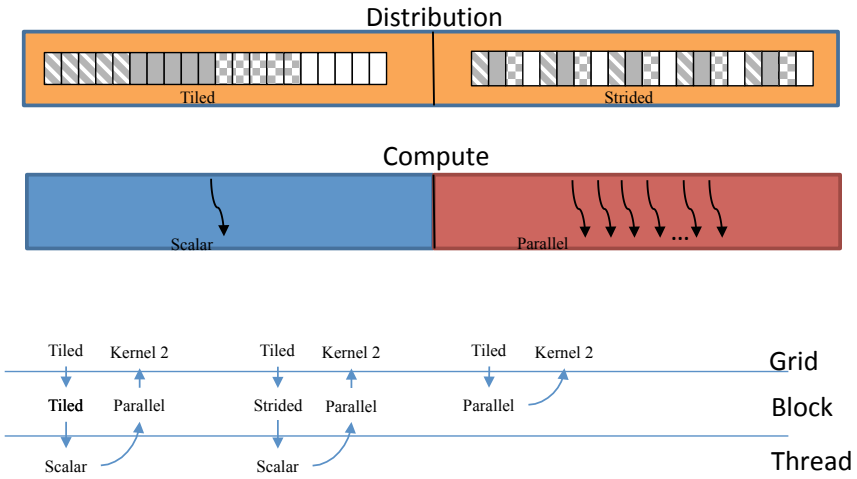

Architecture Hierarchy
Grid
Tiled
Block
Strided
Thread
Scalar
Parallel
...

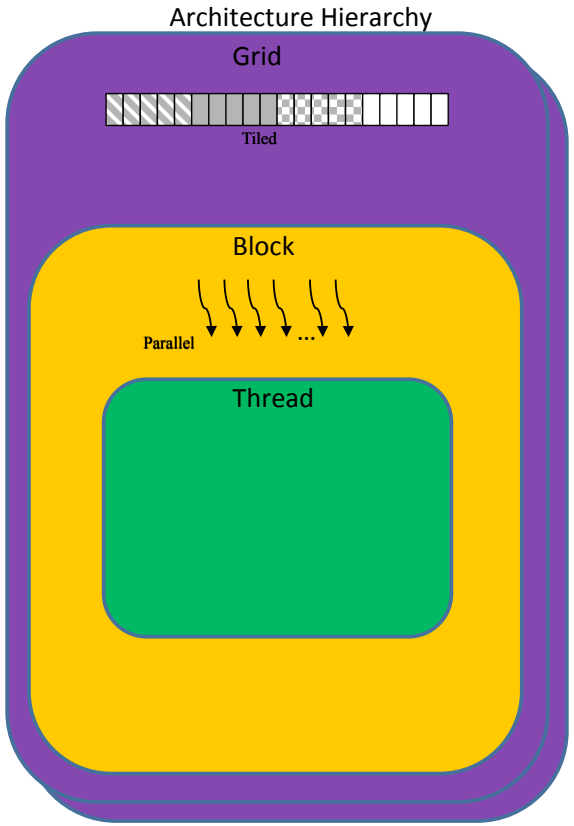

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...
Tiled
Kernel 2
Tiled
Kernel 2
Tiled
Parallel
Strided
Parallel
Scalar
Scalar
Grid
Block
Thread


Architecture Hierarchy
Grid
Tiled
Block
Strided
Thread
Scalar
Parallel
...

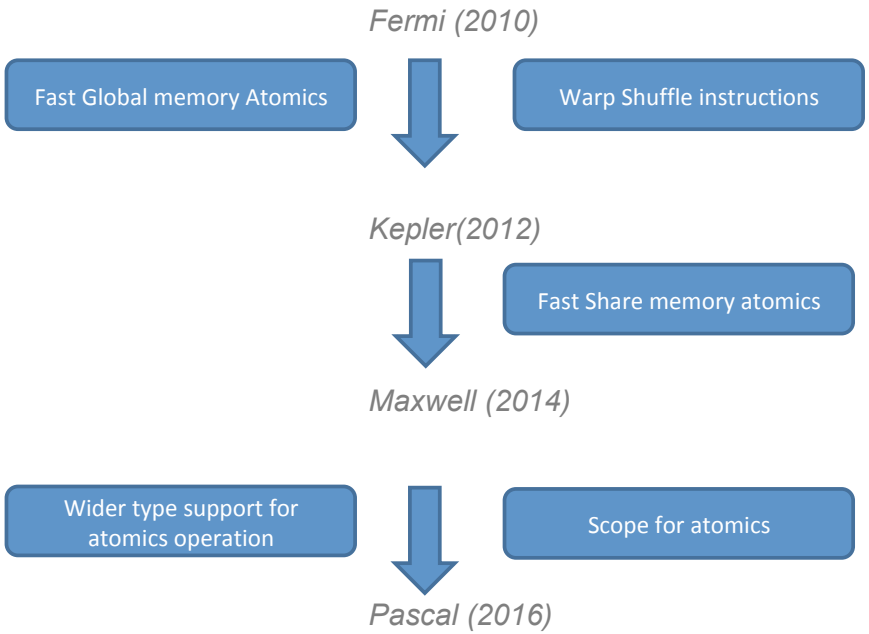

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...
Tiled
Kernel 2
Tiled
Kernel 2
Tiled
Parallel
Strided
Parallel
Scalar
Scalar
Grid
Block
Thread


Architecture Hierarchy
Grid
Tiled
Block
Parallel
...
Thread


Distribution
Tiled
Strided
Compute
Scalar
Parallel
...
Tiled
Kernel 2
Tiled
Kernel 2
Tiled
Kernel 2
Tiled
Parallel
Strided
Parallel
Parallel
Scalar
Scalar
Grid
Block
Thread


Architecture Hierarchy
Grid
Tiled
Block
Parallel
...
Thread

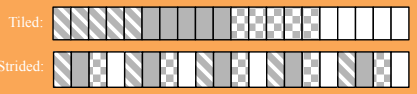
GPU ISA and Microarchitecture Support



```
1  __codelet
2  int sum(const Array<1,int> in) {
3   unsigned len = in.Size();
4   int accum = 0;
5   for(unsigned i=0; i < len; in.Stride()) {
6     accum += in[i];
7   }
8   return accum;
9  }
```

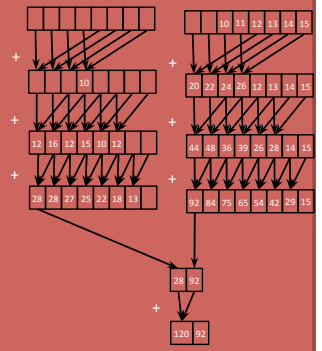
(a) Atomic autonomous codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   __tunable unsigned p;
4   unsigned len = in.Size();
5   unsigned tile = (len+p-1)/p;
6   Sequence start(…);
7   Sequence end(…);
8   Sequence inc(…)
9   Map map( sum, partition(in, p, start, inc, end));
10
11  return sum(map);
12 }
```



(b) Compound codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared int partial[vthread.MaxSize()];
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11       tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial[vthread.VectorId()] = val;
17    if(vthread.VectorId() == 0){
18      val = (vthread.ThreadId() <= (in.Size() / vthread.MaxSize())) ?
19       partial[vthread.LaneId()] : 0;
20      for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
21        val += (vthread.LaneId() + offset < vthread.Size()) ?
22       partial[vthread.ThreadId()+offset] : 0;
23        partial[vthread.ThreadId()] = val;
24      }
25    }
26  }
27  return val;
28 }
```



(c) Atomic cooperative codelet

Tangram Reduction Codelets

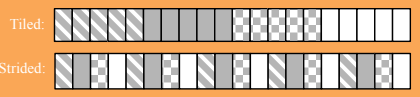
Global Memory Atomics

```
1  __codelet
2  int sum(const Array<1,int> in) {
3    __tunable unsigned p;
4    unsigned len = in.Size();
5    unsigned tile = (len+p-1)/p;
6    Sequence start(…);
7    Sequence end(…);
8    Sequence inc(…)
9    Map map( sum, partition(in, p, start, inc, end));
10
11   return sum(map);
12 }
```

(b) Compound codelet

Global Memory Atomics

```
__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));

  return sum(map);
}
```



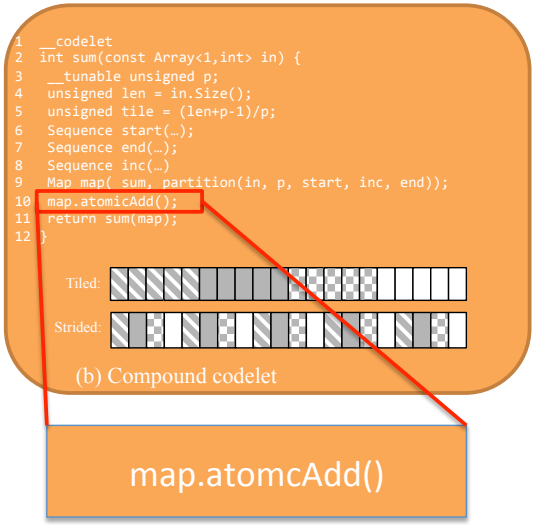
(b) Compound codelet

Thread
Block
Grid

```
__inline__ __device__
void Reduce_Thread(int *Return,
                   int *input_x,
                   ...){

    Return[threadIdx.x] = ...;
}

__global__
void Reduce_Block(int *Return,
                  int *input_x,
                  ...){
    int p = blockDim.x;
    ...
    __shared__ int *map_return;
    if (threadIdx.x == 0)
      map_return = new int[p];
    __syncthreads();
    Reduce_Thread(map_return,
                  input_x + ,
                  ...);
    ...
    if (threadIdx.x == 0)
      Return[blockIdx.x] = ...;
}

template
<unsigned int TGM_TEMPLATE_0>
int Reduce_Grid(int *input_x,
                ...){
  int p = TGM_TEMPLATE_0;
  ...
  int *map_return_block;
  cudaMalloc(&map_return_block,
             (p)*sizeof(int));
  ...
  Reduce_Block<<<p,...>>
               (map_return_block,
                input_x,
                ...);
}
```

Listing 1. Standard Version

Global Memory Atomics



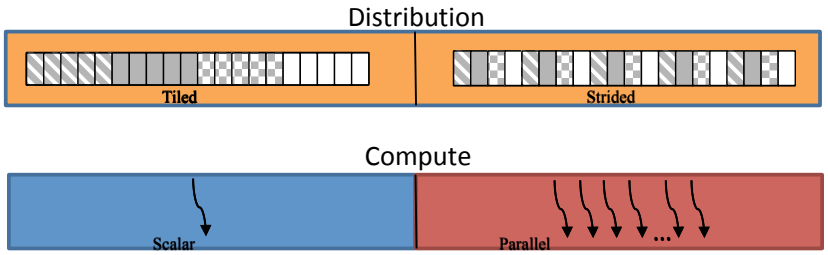
Thread
Block
Grid

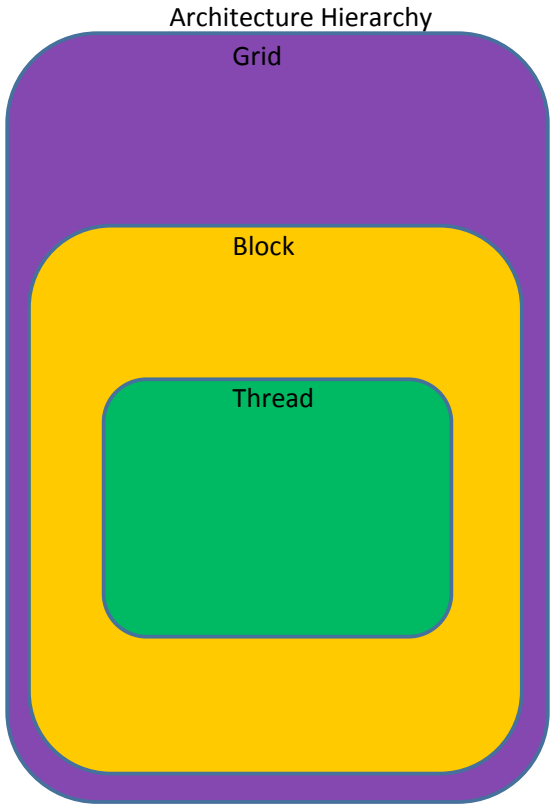
```
__inline__ __device__
void Reduce_Thread(int *Return,
                   int *input_x,
                   ...){
    Return[threadIdx.x] = ...;
}

__global__
void Reduce_Block(int *Return,
                  int *input_x,
                  ...){
    int p = blockDim.x;
    ...
    __shared__ int *map_return;
    if (threadIdx.x == 0)
      map_return = new int[p];
    __syncthreads();
    Reduce_Thread(map_return,
                  input_x + ,
                  ...);
    ...
    if (threadIdx.x == 0)
      Return[blockIdx.x] = ...;
}

template
<unsigned int TGM_TEMPLATE_0>
int Reduce_Grid(int *input_x,
                ...){
   int p = TGM_TEMPLATE_0;
   ...
   int *map_return_block;
   cudaMalloc(&map_return_block,
              (p)*sizeof(int));
   ...
   Reduce_Block<<<p,...>>
               (map_return_block,
                input_x,
                ...);
}
```

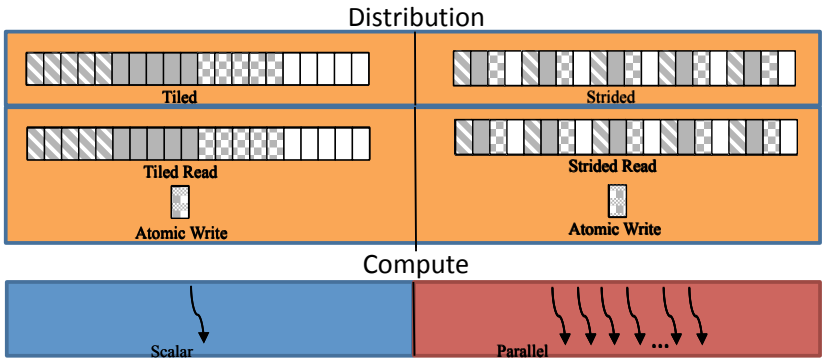
Listing 1. Standard Version

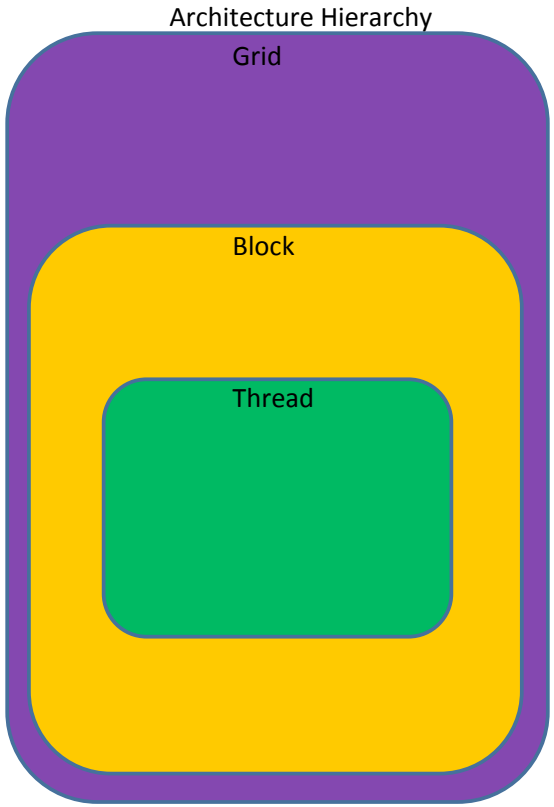
```
__inline__ __device__
void Reduce_Thread(int *Return,
                   int *input_x,
                   ...){
    atomicAdd_block(Return, ...);
}

__global__
void Reduce_Block(int *Return,
                  int *input_x,
                  ...){
    int p = blockDim.x;
    ...
    __shared__ int *map_return;
    if (threadIdx.x == 0)
      map_return = new int[1];
    __syncthreads();
    Reduce_Thread(map_return,
                  input_x + ,
                  ...);
    ...
    if (threadIdx.x == 0)
      atomicAdd(Return, ...);
}

template
<unsigned int TGM_TEMPLATE_0>
int Reduce_Grid(int *input_x,
                ...){
   int p = TGM_TEMPLATE_0;
   ...
   int *map_return_block;
   cudaMalloc(&map_return_block,
              sizeof(int));
   ...
   Reduce_Block<<<p,...>>
               (map_return_block,
                input_x,
                ...);
}
```

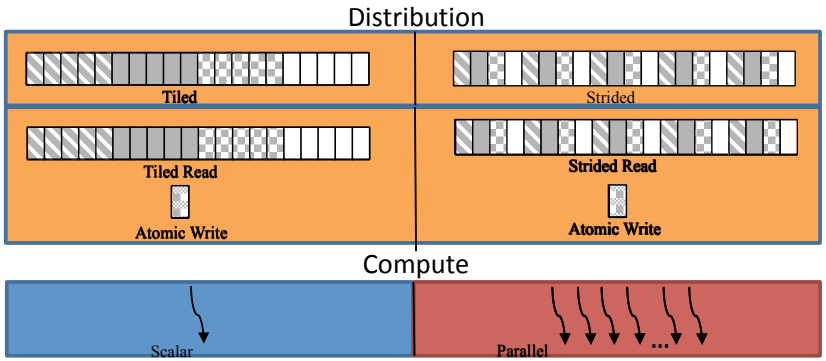
Listing 2. Global Atomics

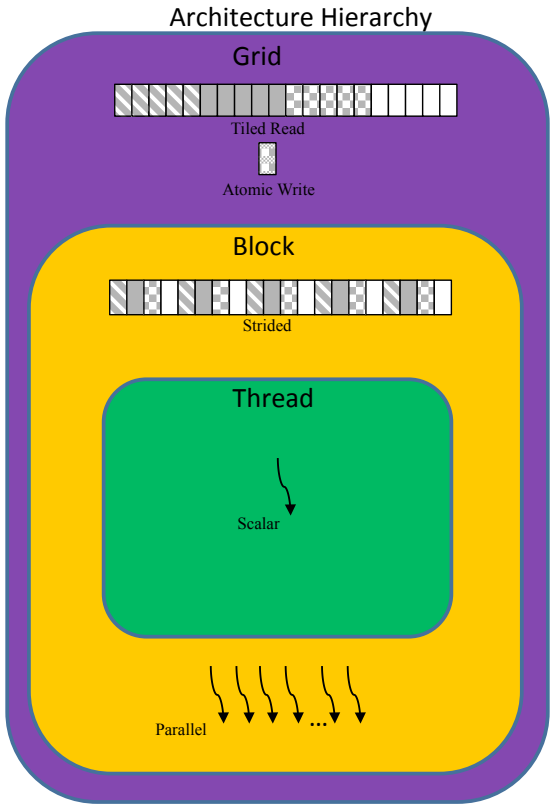

Distribution
Tiled
Strided
Compute
Scalar
Parallel
...

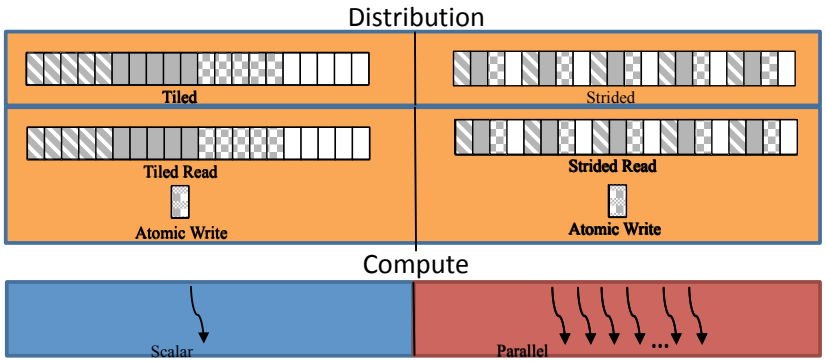

Architecture Hierarchy
Grid
Block
Thread

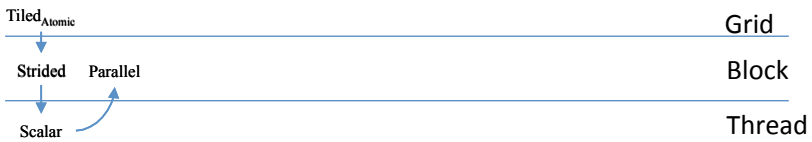

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...

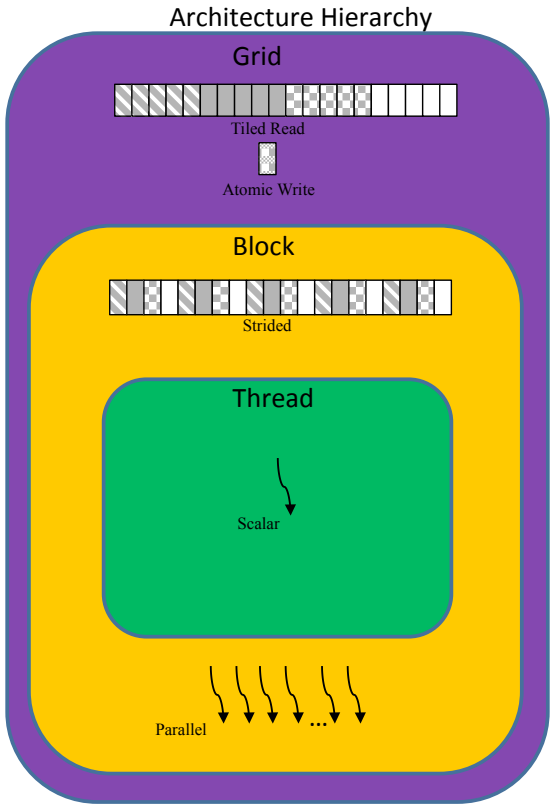

Architecture Hierarchy
Grid
Block
Thread

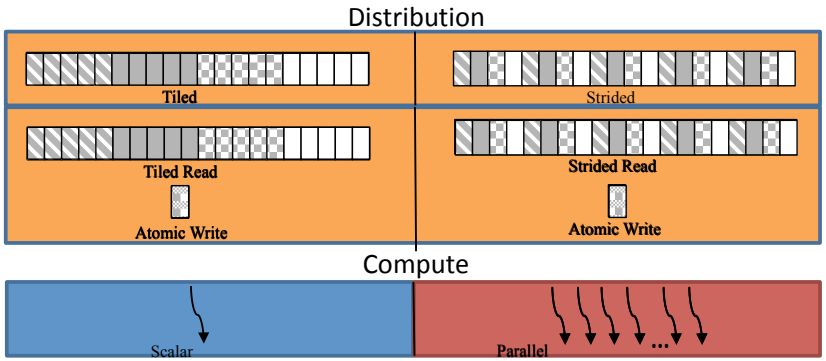

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...

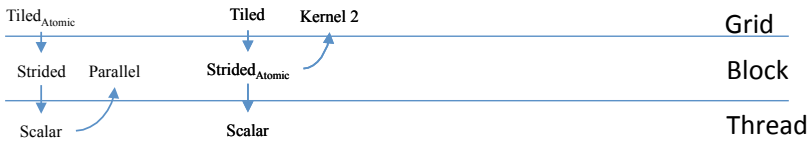

Architecture Hierarchy
Grid
Tiled Read
Atomic Write
Block
Strided
Thread
Scalar
Parallel
...

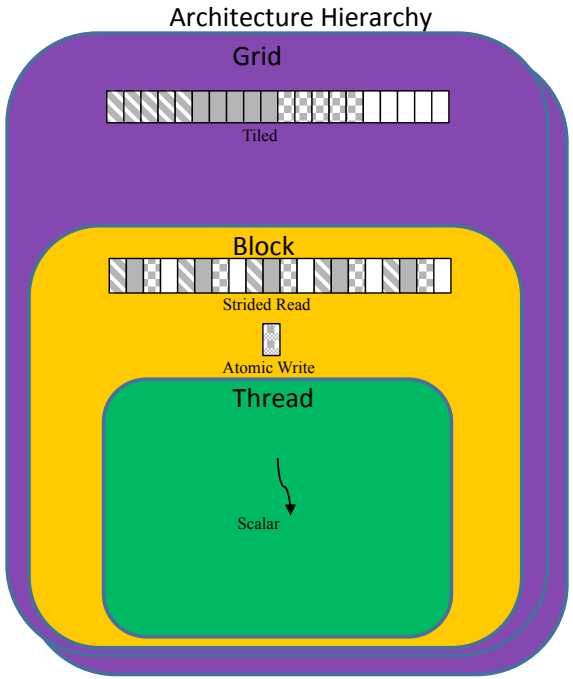

Distribution
Tiled
Strided
Tiled Read
Atomic Write
Strided Read
Atomic Write
Compute
Scalar
Parallel
...

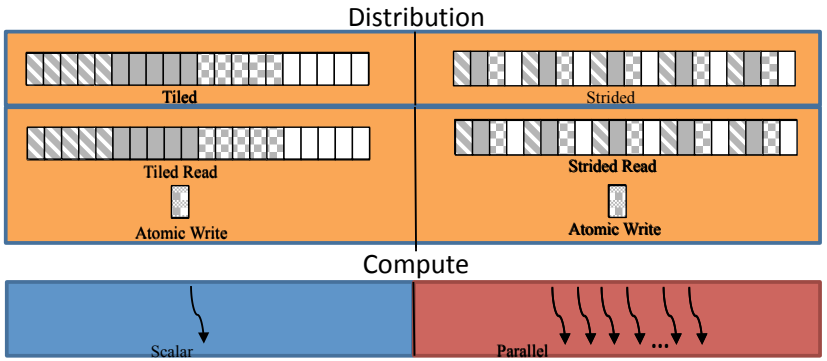

Tiled Atomic
Strided
Parallel
Scalar
Grid
Block
Thread

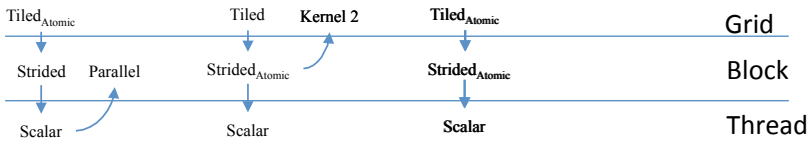

Architecture Hierarchy
Grid
Tiled Read
Atomic Write
Block
Strided
Thread
Scalar
Parallel
...

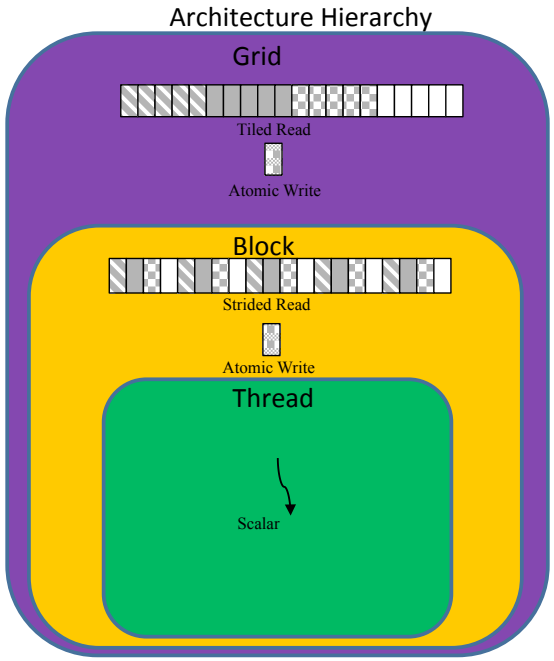

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel

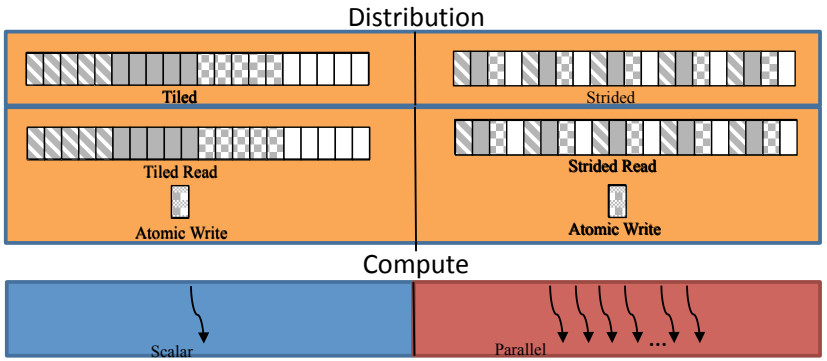

Tiled$_{Atomic}$
Tiled
Kernel 2
Strided
Parallel
Strided$_{Atomic}$
Scalar
Scalar
Grid
Block
Thread

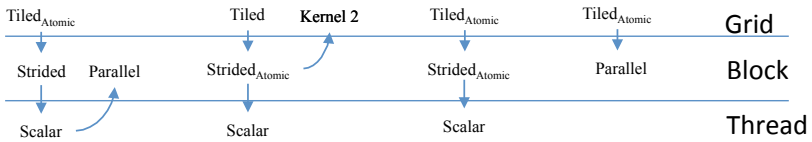

Architecture Hierarchy
Grid
Tiled
Block
Strided Read
Atomic Write
Thread
Scalar

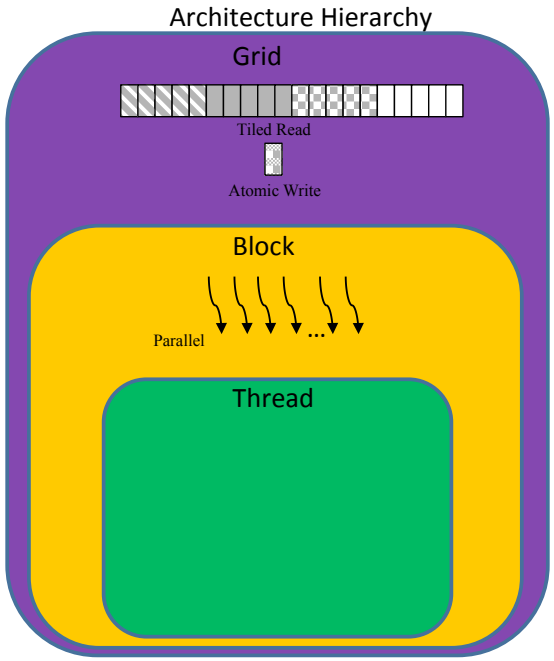

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel


Tiled$_{Atomic}$
Tiled
Kernel 2
Tiled$_{Atomic}$
Strided
Parallel
Strided$_{Atomic}$
Strided$_{Atomic}$
Scalar
Scalar
Scalar
Grid
Block
Thread

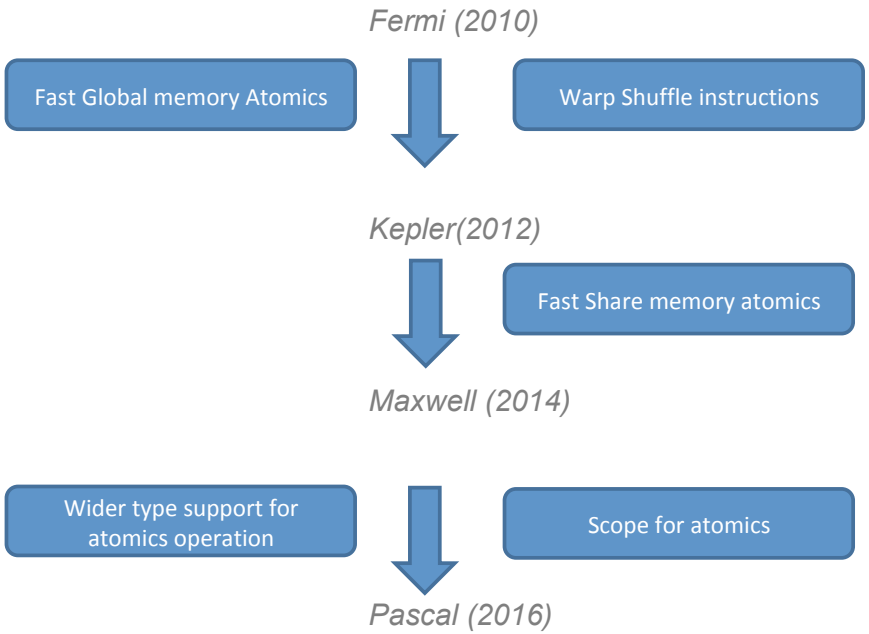

Architecture Hierarchy
Grid
Tiled Read
Atomic Write
Block
Strided Read
Atomic Write
Thread
Scalar


Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...


Tiled$_{Atomic}$
Tiled
Kernel 2
Tiled$_{Atomic}$
Tiled$_{Atomic}$
Grid
Strided
Parallel
Strided$_{Atomic}$
Strided$_{Atomic}$
Parallel
Block
Scalar
Scalar
Scalar
Thread


Architecture Hierarchy
Grid
Tiled Read
Atomic Write
Block
Parallel
...
Thread

GPU ISA and Microarchitecture Support



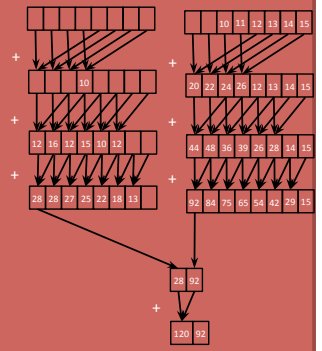
```
1  __codelet
2  int sum(const Array<1,int> in) {
3   unsigned len = in.Size();
4   int accum = 0;
5   for(unsigned i=0; i < len; in.Stride()) {
6     accum += in[i];
7   }
8   return accum;
9  }
```

(a) Atomic autonomous codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   __tunable unsigned p;
4   unsigned len = in.Size();
5   unsigned tile = (len+p-1)/p;
6   Sequence start(…);
7   Sequence end(…);
8   Sequence inc(…)
9   Map map( sum, partition(in, p, start, inc, end));
10
11  return sum(map);
12 }
```

(b) Compound codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared int partial[vthread.MaxSize()];
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11       tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial[vthread.VectorId()] = val;
17    if(vthread.VectorId() == 0){
18      val = (vthread.ThreadId() <= (in.Size() / vthread.MaxSize())) ?
19       partial[vthread.LaneId()] : 0;
20      for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
21        val += (vthread.LaneId() + offset < vthread.Size()) ?
22       partial[vthread.ThreadId()+offset] : 0;
23        partial[vthread.ThreadId()] = val;
24      }
25    }
26  }
27  return val;
28 }
```

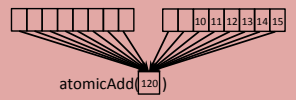


(c) Atomic cooperative codelet

Tangram Reduction Codelets

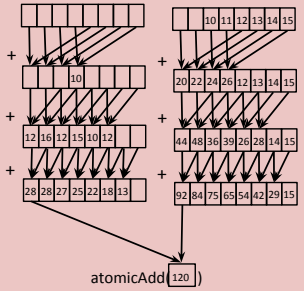
Shared Atomics

```
1  __codelet __coop __tag(shared_V1)
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared _atomicAdd int tmp;
5   int val = 0;
6   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
7   tmp = val;
8   return tmp;
9  }
```



(a)Atomic for single accumulator

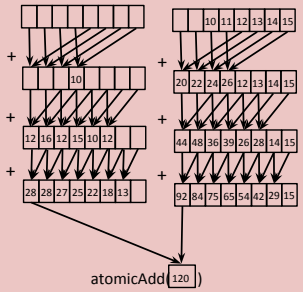
```
1  __codelet __coop __tag(shared_V2)
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared _atomicAdd int partial;
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ? tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial = val;
17    if(vthread.VectorId() == 0)
18      val = partial
19  }
20  return val;
21 }
```



(b) Atomic for partial results

Shared Atomics

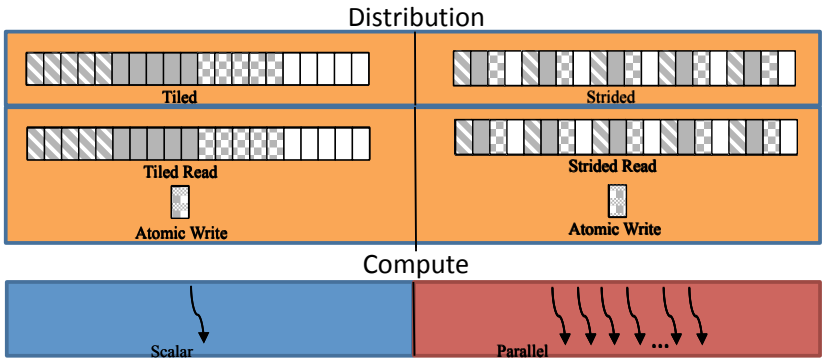
```
1  __codelet __coop __tag(shared_V2)
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4     shared  atomicAdd int partial;
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10   val += (vthread.LaneId() + offset < vthread.Size()) ?
11         tmp[vthread.ThreadId()+offset] : 0;
12   tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial = val;
17    if(vthread.VectorId() == 0)
18      val = partial
19  }
20  return val;
21 }
```

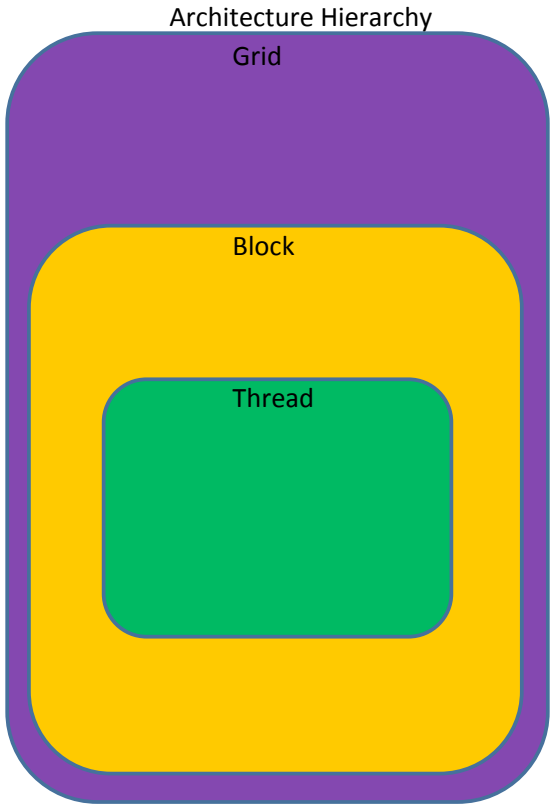


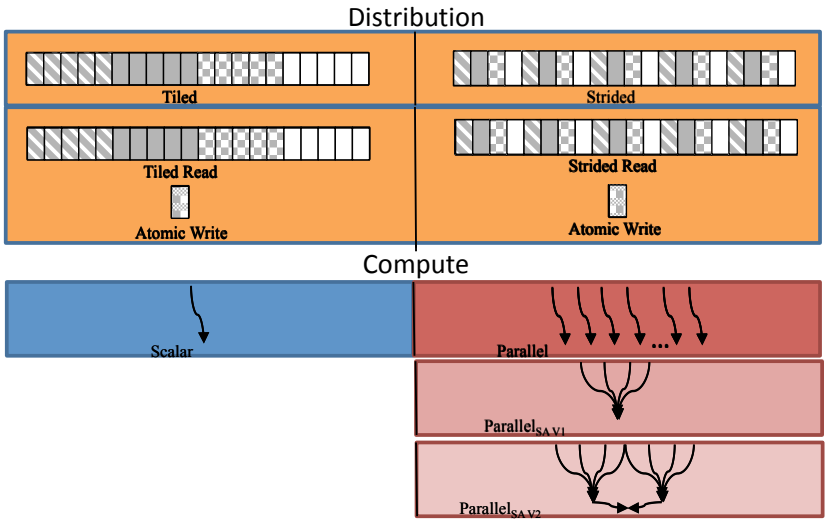
(b) Atomic for partial results

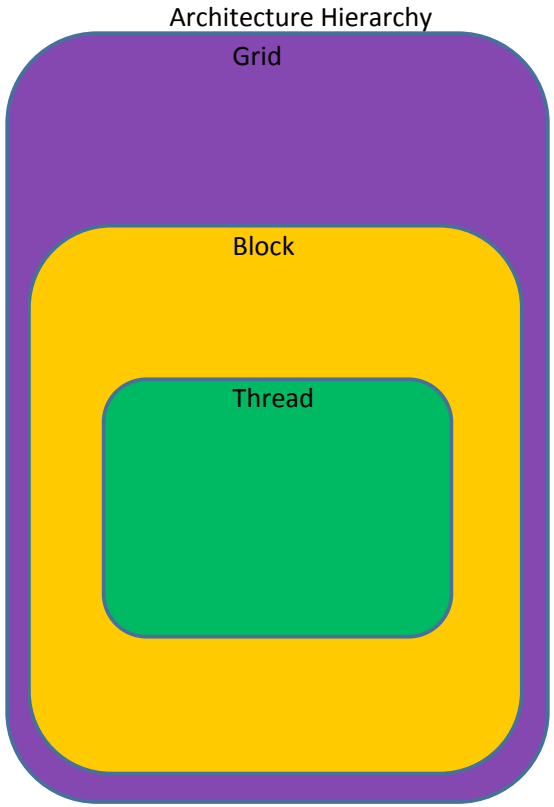
```
__global__
void Reduce_block(int *Return, int *input_x, int SourceSize,
                  int ObjectSize) {
  unsigned int blockID = blockIdx.x;
  __shared__ int partial;
  if(threadIdx.x == 0)
    partial = 0;
  __syncthreads();
  extern __shared__ int tmp[];
  tmp[threadIdx.x] = 0;
  __syncthreads();
  int val = 0;
  val = (((threadIdx.x < ObjectSize_31)) &&
        ((blockIdx.x * blockDim.x + threadIdx.x) < SourceSize_30))
        ? input_x[blockIdx.x * blockDim.x + threadIdx.x]
        : 0;
  tmp[threadIdx.x] = val;
  for (int offset = (32 / 2); (offset > 0); offset /= 2) {
    val += (((threadIdx.x % warpSize + offset_19) < warpSize))
           ? tmp[(threadIdx.x + offset)]
           : 0;
    tmp[threadIdx.x] = val;
    __syncthreads();
  }
  if (((ObjectSize != 32) && ((ObjectSize / 32) > 0))) {
    if ((threadIdx.x % warpSize == 0)) {
      atomicAdd(&partial, val);
      __syncthreads();
    }
    if ((threadIdx.x / warpSize == 0)) {
      val = partial;
    }
  }
  if (threadIdx.x == 0) {
    Return[blockID] = val;
  }
}
```

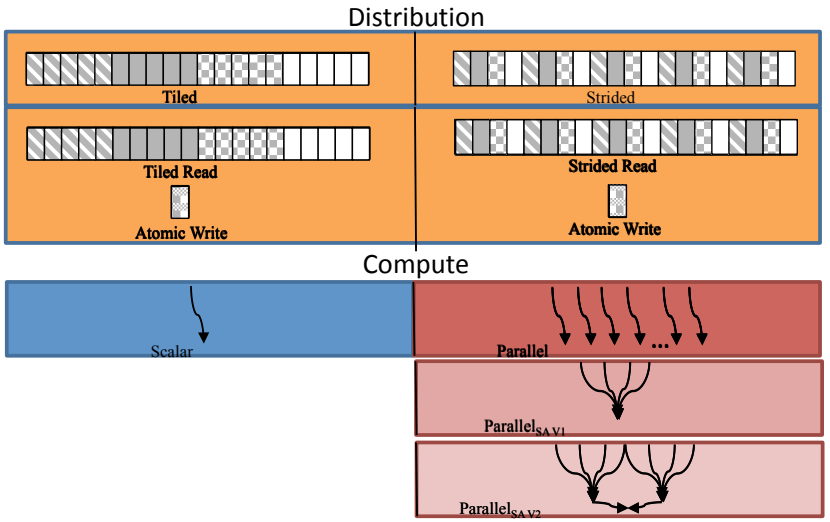
Listing 3. Shared atomics code for Figure 3(b)

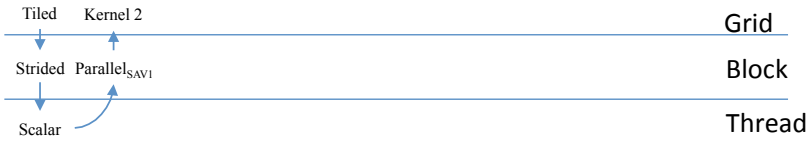

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...

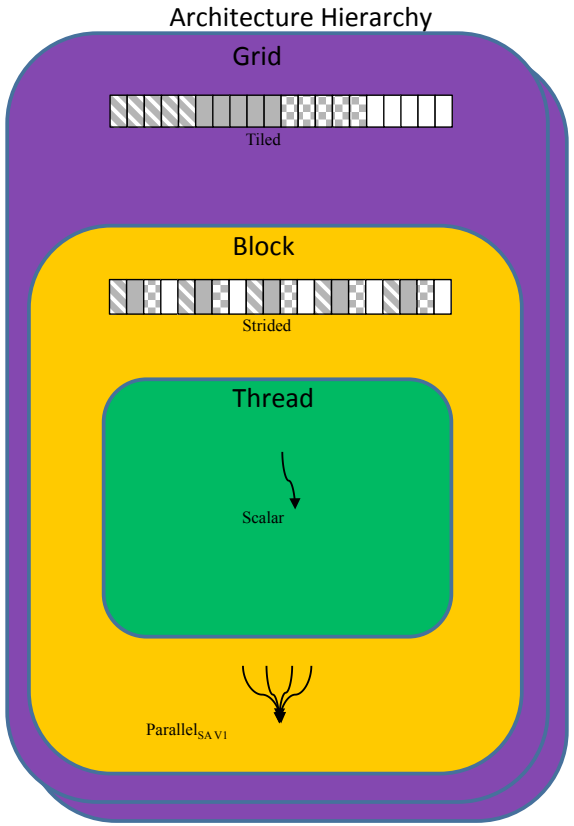

Architecture Hierarchy
Grid
Block
Thread

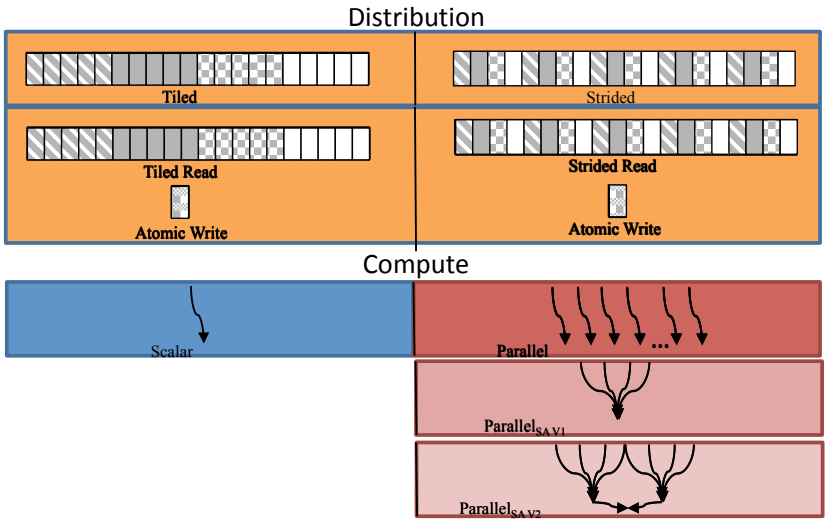

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$

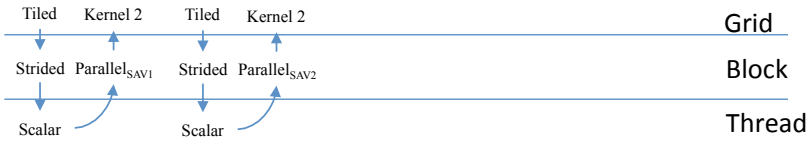

Architecture Hierarchy
Grid
Block
Thread

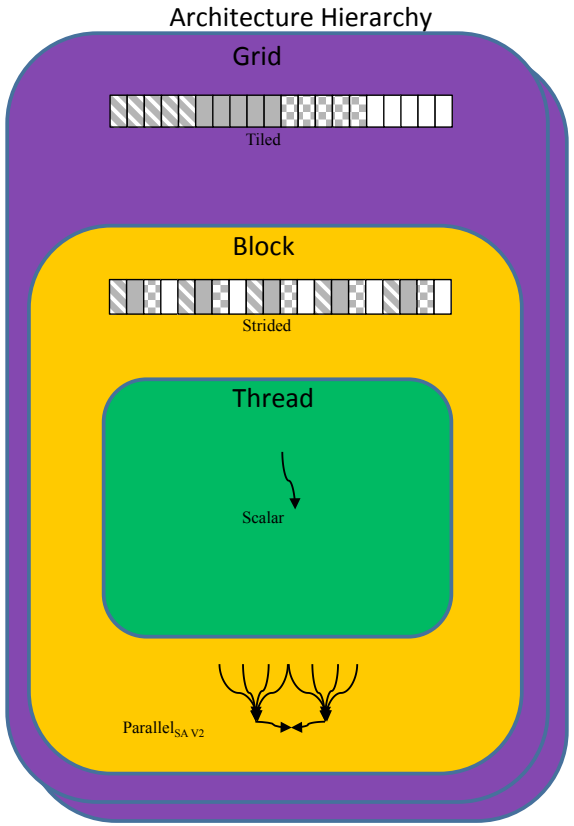

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$

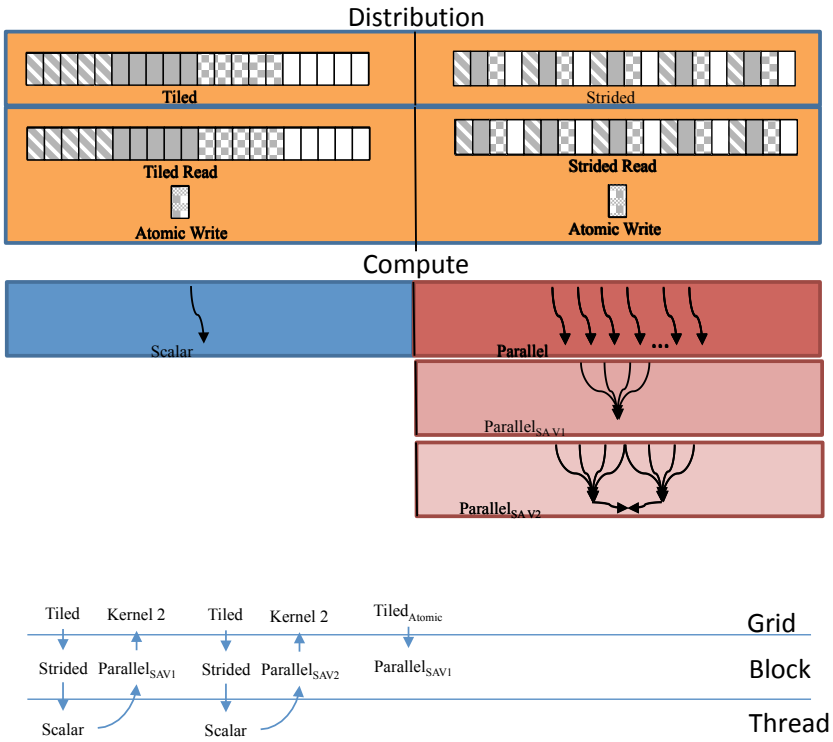

Tiled
Kernel 2
Strided
Parallel$_{SAV1}$
Scalar
Grid
Block
Thread

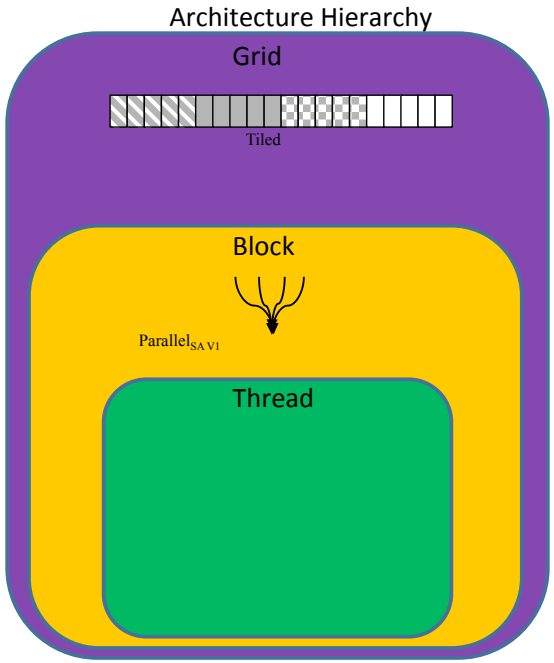

Architecture Hierarchy
Grid
Tiled
Block
Strided
Thread
Scalar
Parallel$_{SA\ V1}$

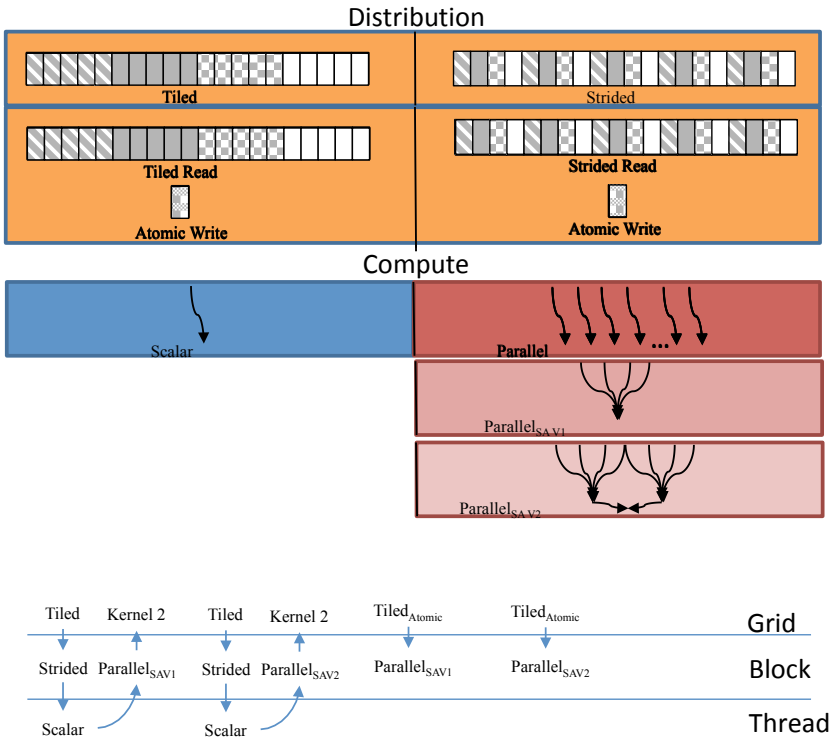

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$

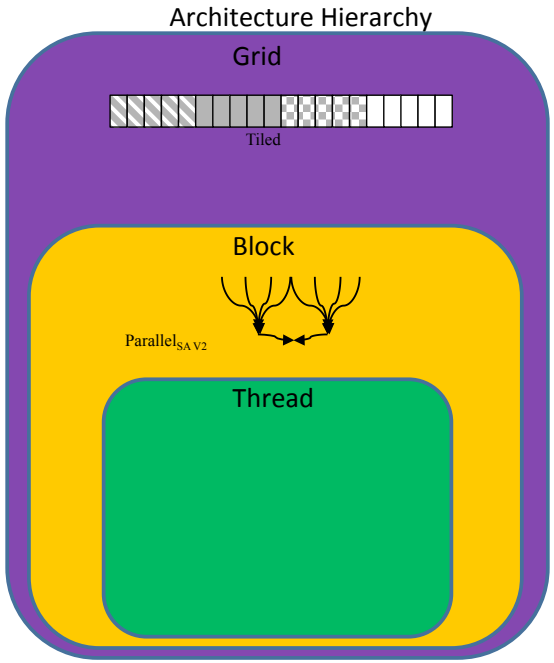

Tiled
Kernel 2
Tiled
Kernel 2
Strided
Parallel$_{SAV1}$
Strided
Parallel$_{SAV2}$
Scalar
Scalar
Grid
Block
Thread


Architecture Hierarchy
Grid
Tiled
Block
Strided
Thread
Scalar
Parallel$_{SA\ V2}$

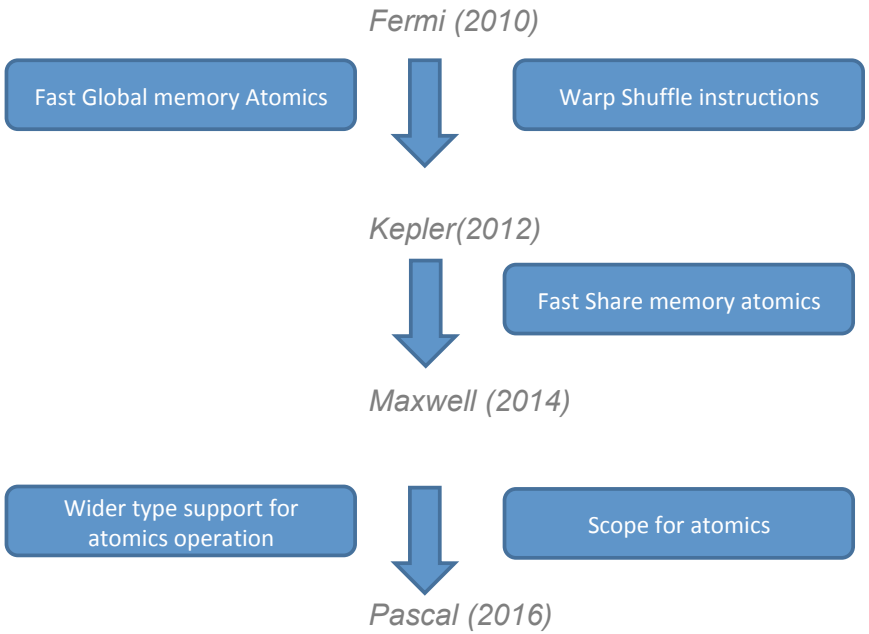

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$
Tiled
Kernel 2
Tiled
Kernel 2
Tiled$_{Atomic}$
Strided
Parallel$_{SAV1}$
Strided
Parallel$_{SAV2}$
Parallel$_{SAV1}$
Scalar
Scalar
Grid
Block
Thread


Architecture Hierarchy
Grid
Tiled
Block
Parallel$_{SA\ V1}$
Thread


Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$
Tiled
Kernel 2
Tiled
Kernel 2
Tiled$_{Atomic}$
Tiled$_{Atomic}$
Grid
Strided
Parallel$_{SAV1}$
Strided
Parallel$_{SAV2}$
Parallel$_{SAV1}$
Parallel$_{SAV2}$
Block
Scalar
Scalar
Thread


Architecture Hierarchy
Grid
Tiled
Block
Parallel$_{SA\ V2}$
Thread

GPU ISA and Microarchitecture Support



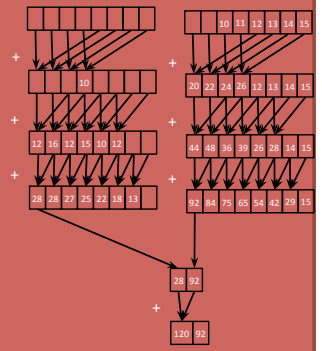
```
1  __codelet
2  int sum(const Array<1,int> in) {
3   unsigned len = in.Size();
4   int accum = 0;
5   for(unsigned i=0; i < len; in.Stride()) {
6     accum += in[i];
7   }
8   return accum;
9  }
```

(a) Atomic autonomous codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   __tunable unsigned p;
4   unsigned len = in.Size();
5   unsigned tile = (len+p-1)/p;
6   Sequence start(…);
7   Sequence end(…);
8   Sequence inc(…)
9   Map map( sum, partition(in, p, start, inc, end));
10
11  return sum(map);
12 }
```

(b) Compound codelet

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared int partial[vthread.MaxSize()];
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11       tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0)
16      partial[vthread.VectorId()] = val;
17    if(vthread.VectorId() == 0){
18      val = (vthread.ThreadId() <= (in.Size() / vthread.MaxSize())) ?
19        partial[vthread.LaneId()] : 0;
20      for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
21        val += (vthread.LaneId() + offset < vthread.Size()) ?
22       partial[vthread.ThreadId()+offset] : 0;
23        partial[vthread.ThreadId()] = val;
24      }
25    }
26  }
27  return val;
28 }
```



(c) Atomic cooperative codelet

Tangram Reduction Codelets

Warp Shuffle Instructions

```
__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;
}
```

(c) Atom

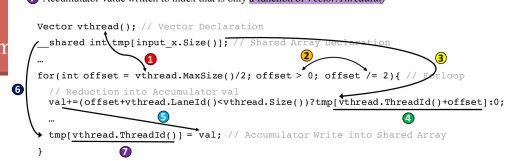
❶❷ *Forloop* bounds are based on *Vector<...>* primitive, and iterator decreases by a constant every iteration

❸ Reading from *__shared* array; values reduced into a local accumulator

❹ Shared array index is a function of *Vector.ThreadId()* and *forloop* iterator

❺❻ Accumulator value written to the same shared array

❼ Accumulator value written to index that is only a function of *Vector.ThreadId()*




```
__global__
void Reduce_block(int *Return, int *input_x, int SourceSize,
                  int ObjectSize) {
  unsigned int blockID = blockIdx.x;
  __shared__ int partial[32];
  if(threadIdx.x < 32)
    partial[threadIdx.x] = 0;
  __syncthreads();
  int val = 0;
  val = (((threadIdx.x < ObjectSize)) &&
           ((blockIdx.x * blockDim.x + threadIdx.x) < SourceSize))
              ? input_x[blockIdx.x * blockDim.x + threadIdx.x]
              : 0;
  for (int offset = (32 / 2); (offset > 0); offset /= 2){
    val += __shfl_down_sync(val, offset, 32);
  }
  if (((ObjectSize != 32) && ((ObjectSize / 32) > 0))){
    if ((threadIdx.x % warpSize == 0)) {
      partial[threadIdx.x / warpSize] = val;
    }
    __syncthreads();
    if ((threadIdx.x / warpSize == 0)) {
      val = ((threadIdx.x <= ((ObjectSize / 32))))
                ? partial[threadIdx.x % warpSize]
                : 0;
      for (int offset = (32 / 2); (offset > 0); offset /= 2){
        val += __shfl_down_sync(val, offset, 32);
      }
    }
  }
  if (threadIdx.x == 0){
    Return[blockID] = val;
  }
}
```

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

Warp Shuffle Instructions

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread();
4   __shared int partial[vthread.MaxSize()];
5   __shared int tmp[in.Size()];
6   int val = 0;
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8   tmp[vthread.ThreadId()] = val;
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11       tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
```

1 2 *Forloop* bounds are based on *Vector<…>* primitive, and iterator decreases by a constant every iteration

3 Reading from *__shared* array; values reduced into a local accumulator

4 Shared array index is a function of *Vector.ThreadId()* and *forloop* iterator

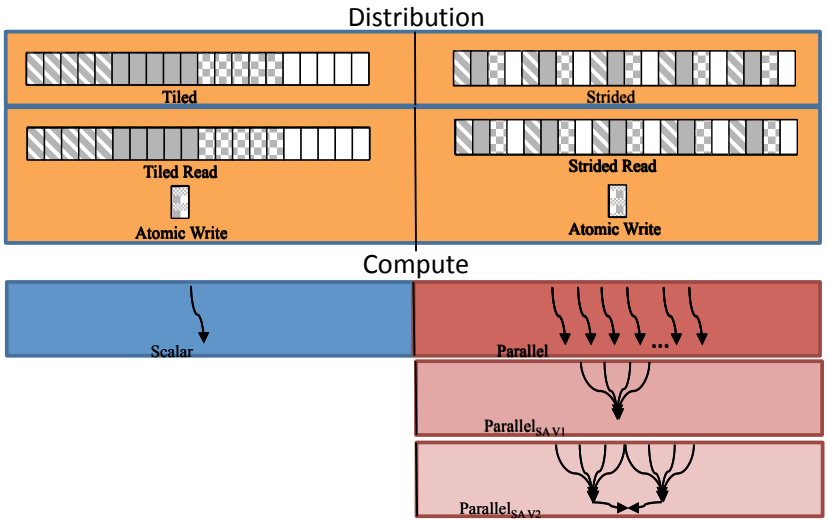
5 6 Accumulator value written to the same shared array

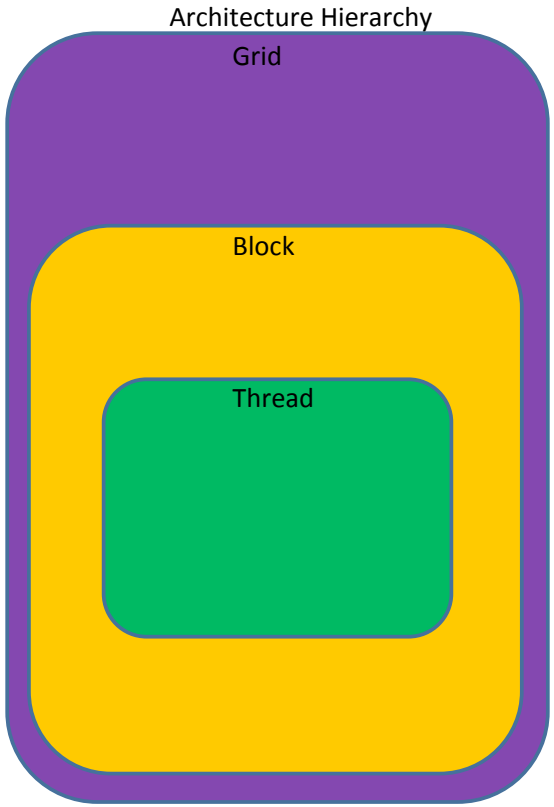
7 Accumulator value written to index that is only a function of *Vector.ThreadId()*

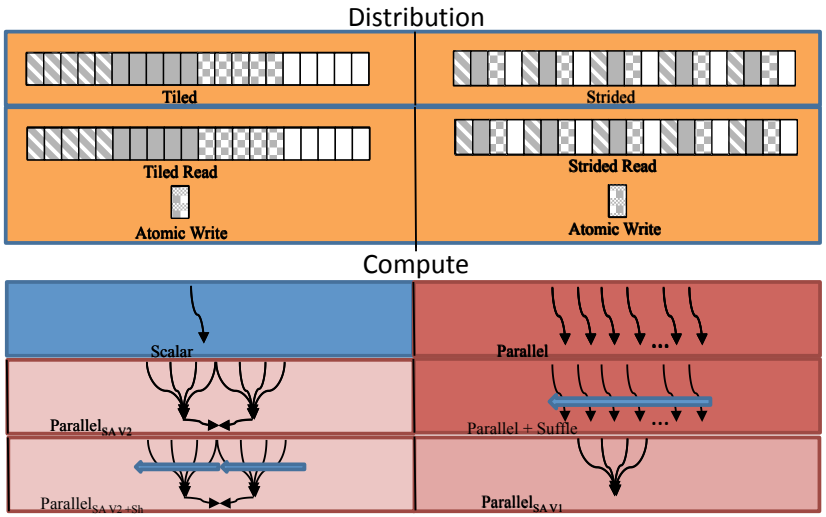
```
Vector vthread(); // Vector Declaration
__shared int tmp[input_x.Size()]; // Shared Array Declaration
…
for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){ // Forloop
  // Reduction into Accumulator val
  val+=(offset+vthread.LaneId()<vthread.Size())?tmp[vthread.ThreadId()+offset]:0;
  …
  tmp[vthread.ThreadId()] = val; // Accumulator Write into Shared Array
}
```

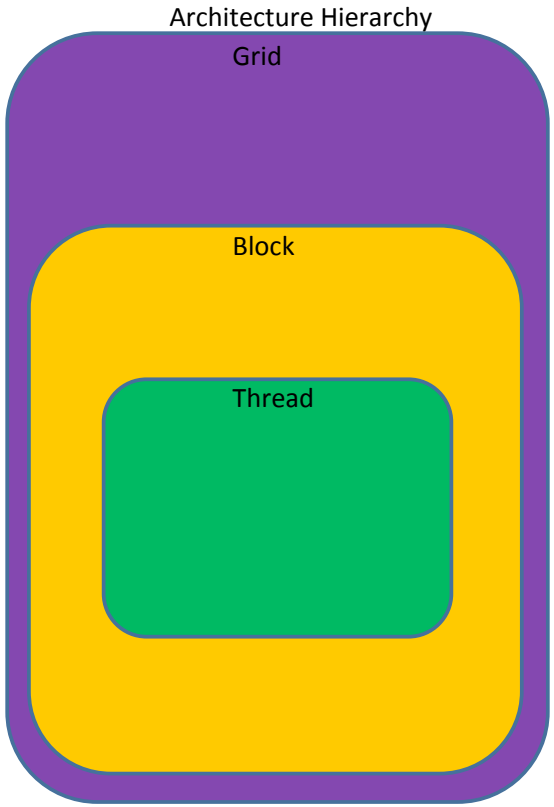
```
1  __global__
2  void Reduce_block(int *Return, int *input_x, int SourceSize,
3                    int ObjectSize) {
4    unsigned int blockID = blockIdx.x;
5    __shared__ int partial[32];
6    if(threadIdx.x < 32)
7      partial[threadIdx.x] = 0;
8    __syncthreads();
9    int val = 0;
10   val = (((threadIdx.x < ObjectSize)) &&
11            ((blockIdx.x * blockDim.x + threadIdx.x) < SourceSize))
12              ? input_x[blockIdx.x * blockDim.x + threadIdx.x]
13              : 0;
14   for (int offset = (32 / 2); (offset > 0); offset /= 2){
15     val += __shfl_down_sync(val, offset, 32);
16   }
17   if (((ObjectSize != 32) && ((ObjectSize / 32) > 0))){
18     if ((threadIdx.x % warpSize == 0)) {
19       partial[threadIdx.x / warpSize] = val;
20     }
21     __syncthreads();
22     if ((threadIdx.x / warpSize == 0)) {
23       val = ((threadIdx.x <= ((ObjectSize / 32))))
24                ? partial[threadIdx.x % warpSize]
25                : 0;
26       for (int offset = (32 / 2); (offset > 0); offset /= 2){
27         val += __shfl_down_sync(val, offset, 32);
28       }
29     }
30   }
31   if (threadIdx.x == 0){
32     Return[blockID] = val;
33   }
34 }
```

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

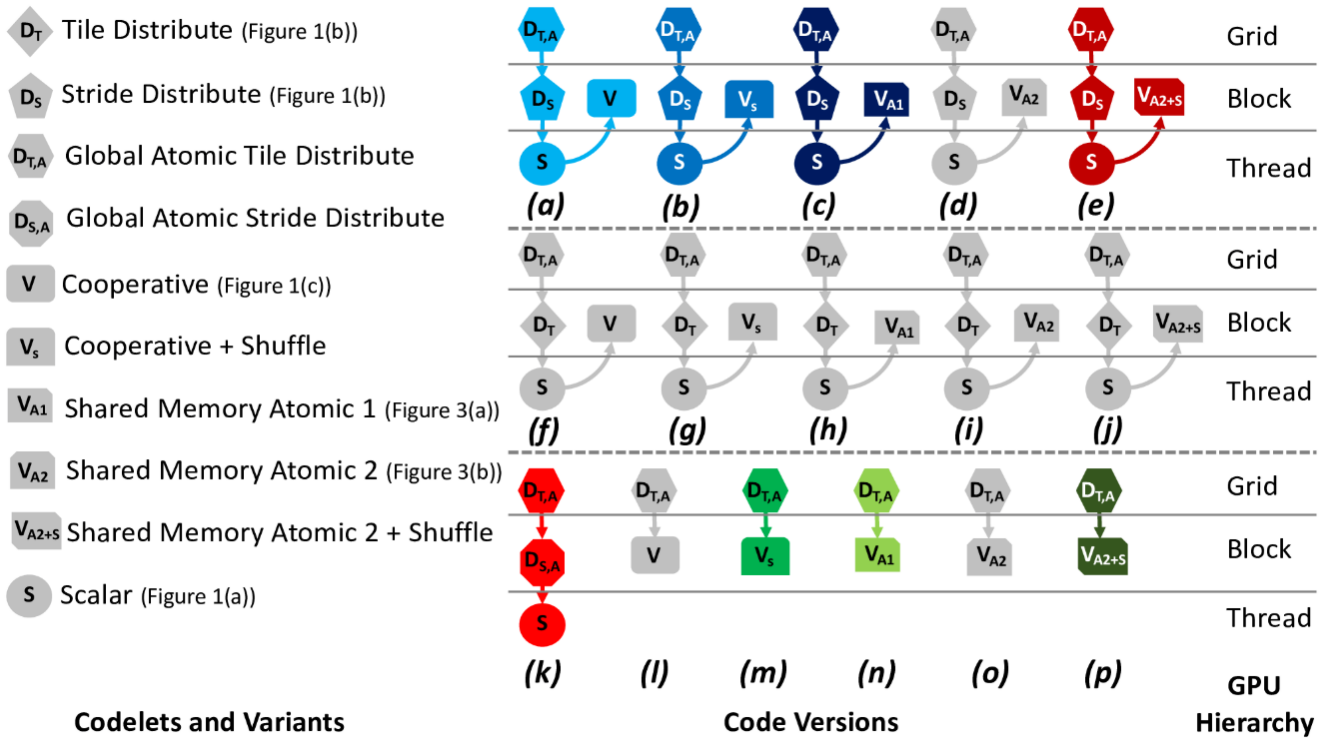

Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
...
Parallel$_{SA\ V1}$
Parallel$_{SA\ V2}$


Architecture Hierarchy
Grid
Block
Thread


Distribution
Tiled
Strided
Tiled Read
Strided Read
Atomic Write
Atomic Write
Compute
Scalar
Parallel
Parallel + Suffle


Architecture Hierarchy
Grid
Block
Thread

Search Space



Over 85 different versions possible!

Experimental Setup

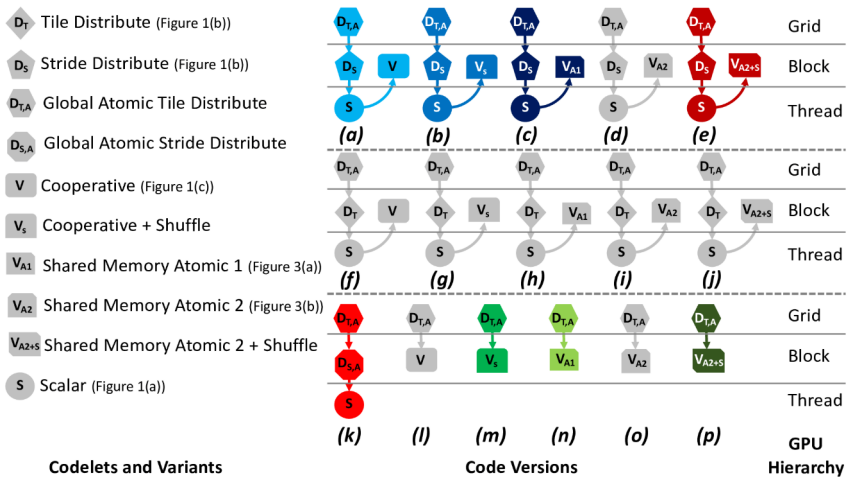
3 GPU architectures

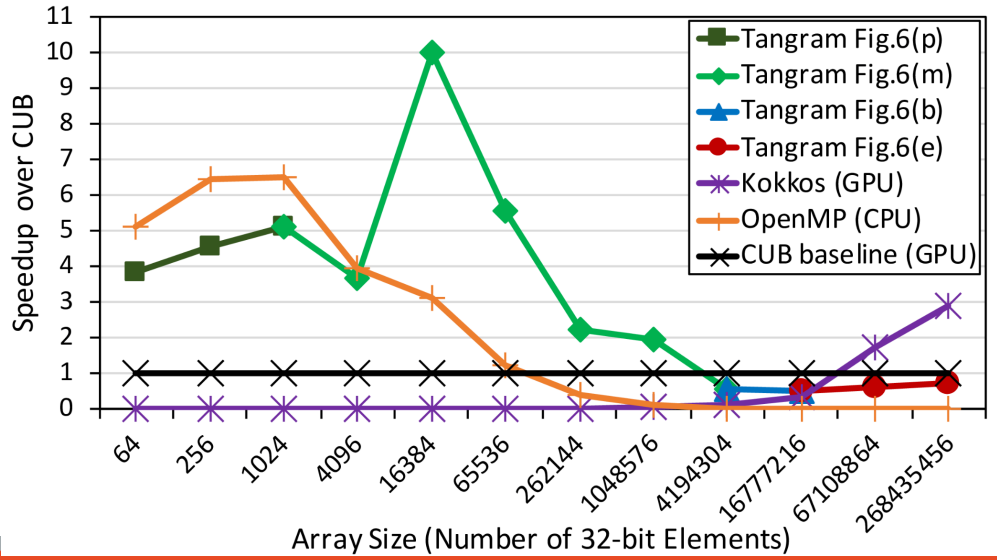
- Kepler(Titan, Blue Waters)
- Maxwell (Cloud)
- Pascal(Piz Daint)

Compare results against

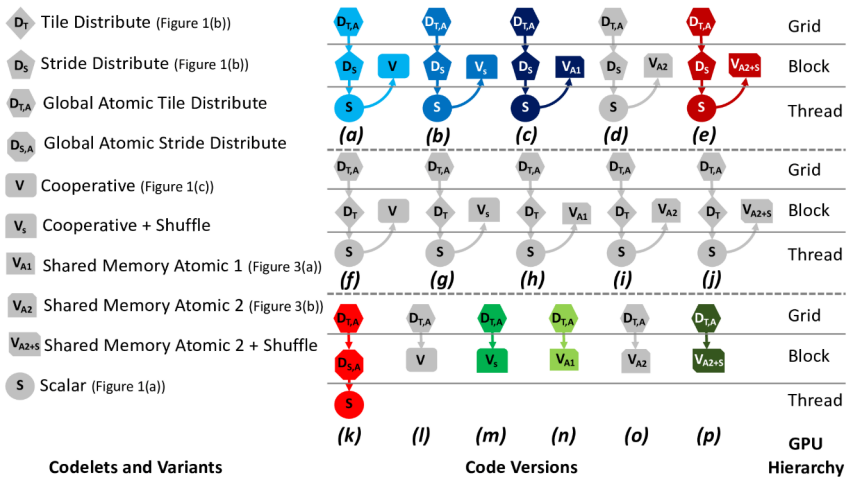
- NVIDIA CUB 1.8.0
- Kokkos GPU backend
- OpemMP 4.0 reduce pragma
  - Dual-socket 8-core 3.5GHz POWER8+
  - gcc 5.4.0

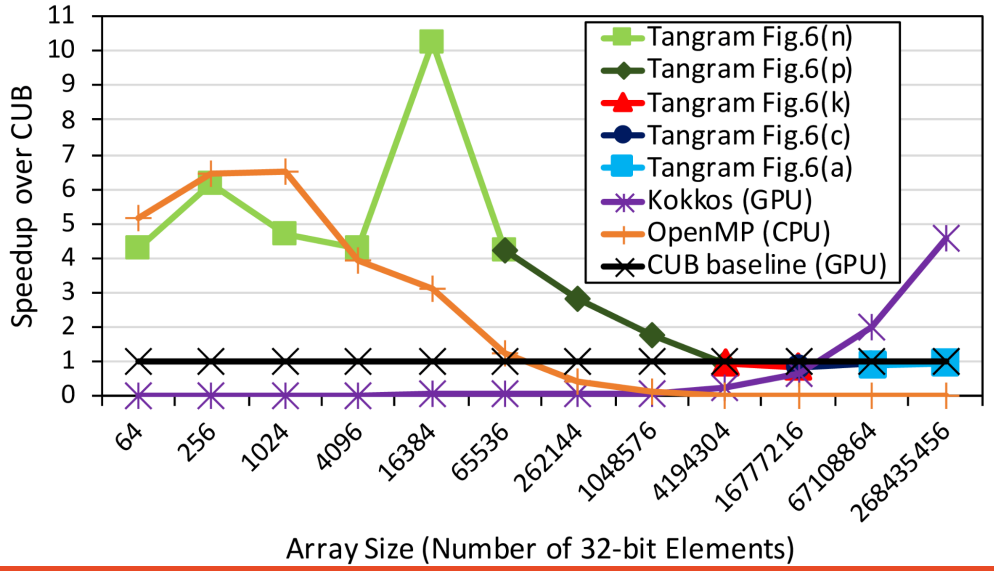
Kepler Results



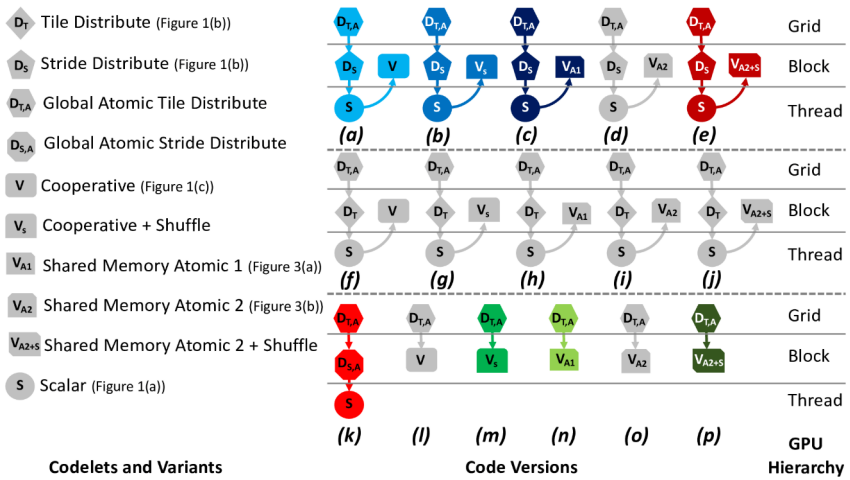


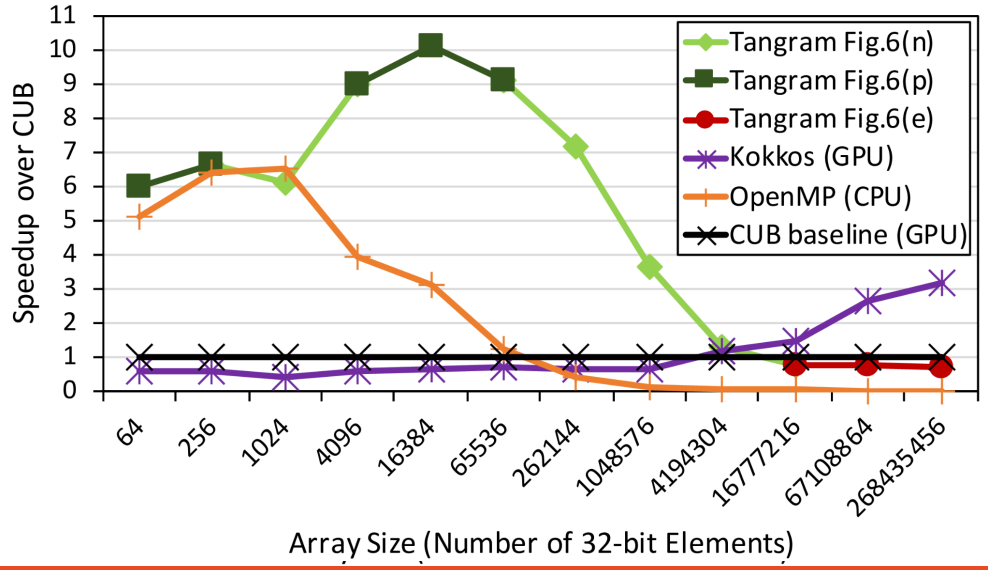
Maxwell Results





Pascal Results





Future work

- Volta
  - 8 TCU units per SM are available
    - Reduction can be express as M x 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