Automatic translation from CUDA to C++

Luca Atzori, Vincenzo Innocente, Felice Pantaleo, Danilo Piparo

31 August, 2015

Goals

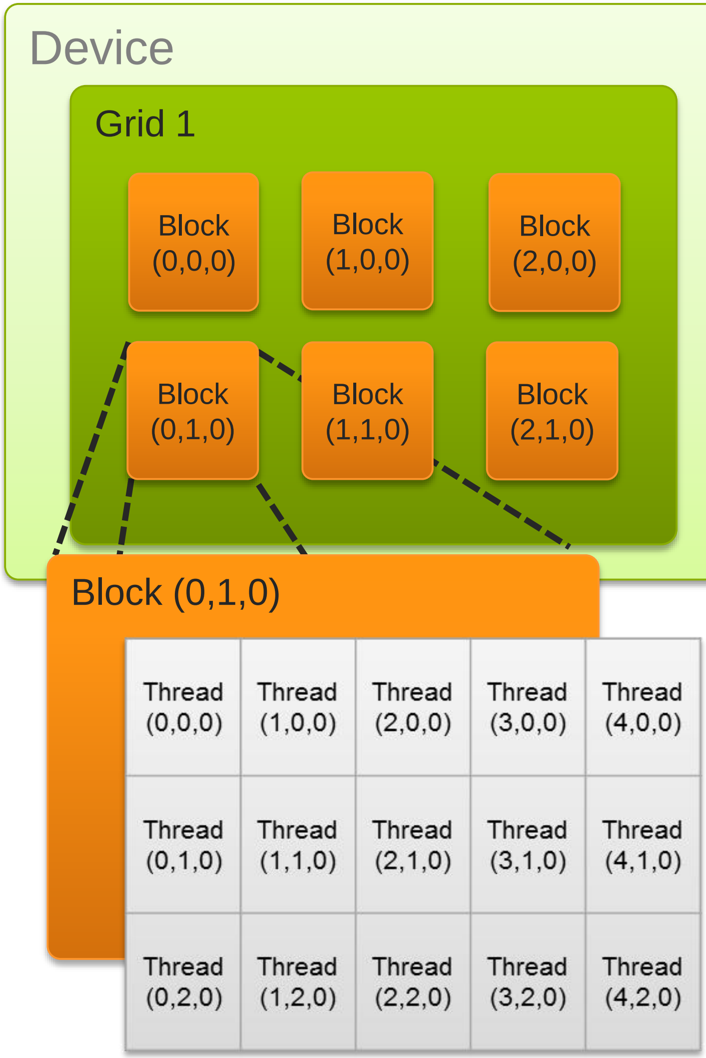
Running CUDA code on CPUs. Why?

Performance portability!

- A major challenge faced today by developers on heterogeneous high performance computers.
- We want to write the **best** possible code using the **best** possible frameworks and run it on the **best** possible hardware.
- Our code should run well even after the translation on a platform that does not needs to have an NVIDIA graphic card.
- CUDA is an effective data-parallel programming model for more than just GPU architectures?

CUDA Computational Model

- Data-parallel model of computation
  - Data split into a 1D, 2D, 3D grid of blocks
  - Each block can be 1D, 2D, 3D in shape
  - More than 512 threads/block
- Built-in variables:
  - `dim3 threadIdx`
  - `dim3 blockIdx`
  - `dim3 blockDim`
  - `dim3 gridDim`



Thread (0,0,0)	Thread (1,0,0)	Thread (2,0,0)	Thread (3,0,0)	Thread (4,0,0)
Thread (0,1,0)	Thread (1,1,0)	Thread (2,1,0)	Thread (3,1,0)	Thread (4,1,0)
Thread (0,2,0)	Thread (1,2,0)	Thread (2,2,0)	Thread (3,2,0)	Thread (4,2,0)

Mapping

The mapping of the computation is NOT straightforward.

- **Conceptually easiest implementation**: spawn a CPU thread for every GPU thread specified in the programming model.
- **Quite inefficient**:
  - mitigates the locality benefits
  - incurs a large amount of scheduling overhead

Mapping

Translating the CUDA program such that the mapping of programming constructs maintains the locality expressed in the programming model with existing operating system and hardware features.

- CUDA blocks execution is asynchronous
- Each CPU thread should be scheduled to a single core for locality
- Maintain the ordering semantics imposed by potential barrier synchronization points

CUDA	**C++**	
block	std::thread / Task	asynchronous
thread	sequential unrolled for loop (can be vectorized)	synchronous (barriers)

Source-to-source translation

Why don’t just find a way to compile it for x86?

- Because having the output source code would be nice!
- We would like to analyze the obtained code:
    - further optimizations
    - debugging
- We don’t want to focus only on x86

Working with ASTs

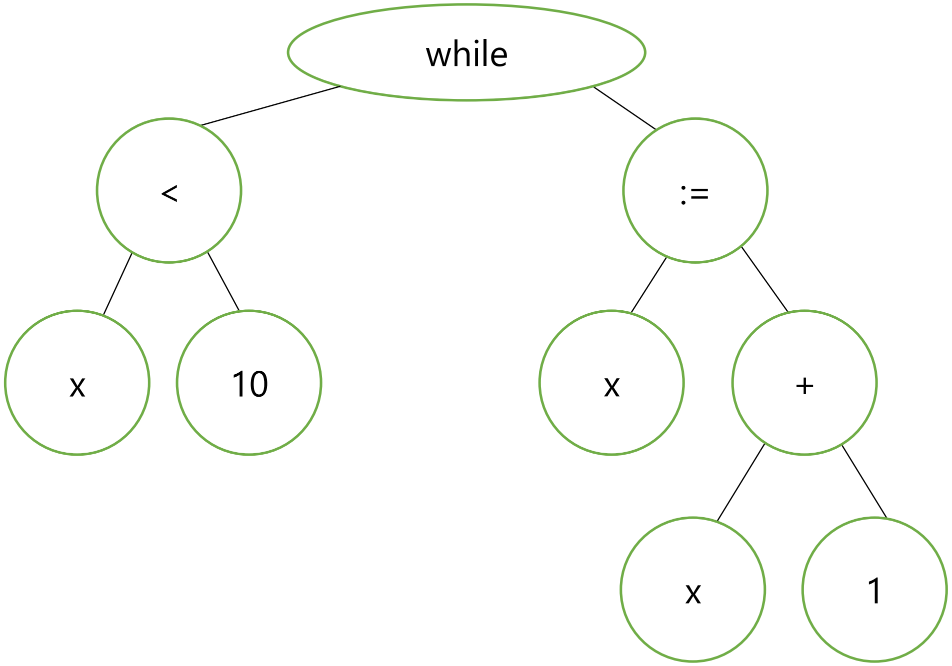
What is an Abstract Syntax Tree?

A tree representation of the abstract syntactic structure of source code written in a programming language.

Why regular expression tools aren't powerful enough?

Almost all programming languages are (deterministic) context free languages, a superset of regular languages.

`while (x < 10) { x := x + 1; }`



Clang

- Clang is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages. It uses LLVM as its back end.
- “Clang’s AST closely resembles both the written C++ code and the C++ standard.”
- This makes Clang’s AST a good fit for refactoring tools.

Clang handles CUDA syntax!

Vector addition: host translation

```
int main(){
    ...
sum<<<numBlocks, numThreadsPerBlock>>>(in_a, in_b, out_c);
    ...
}
```

```
int main() {
    ...
for(i = 0; i < numBlocks; i++)
        sum(in_a, in_b, out_c, i);
    ...
}
```

Vector addition: kernel translation

```
__global__ void sum(int* a, int* b, int* c){
    ...
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
    ...
}
```

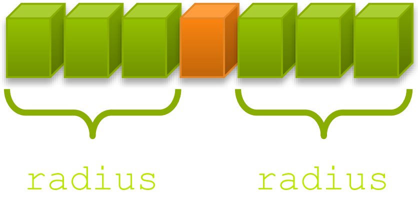
```
void sum(int* a, int* b, int* c, dim3 blockIdx, dim3 blockDim){
dim3 threadIdx;
    ...
//Thread_Loop
for(threadIdx.x.=0; threadIdx.x < blockDim.x; threadIdx.x++){
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
    ...
}
```

Not built-in variables anymore!

The loops enumerate the values of the previously implicit threadIdx

Example: Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering
- If radius is 3, then each output element is the sum of 7 input elements:

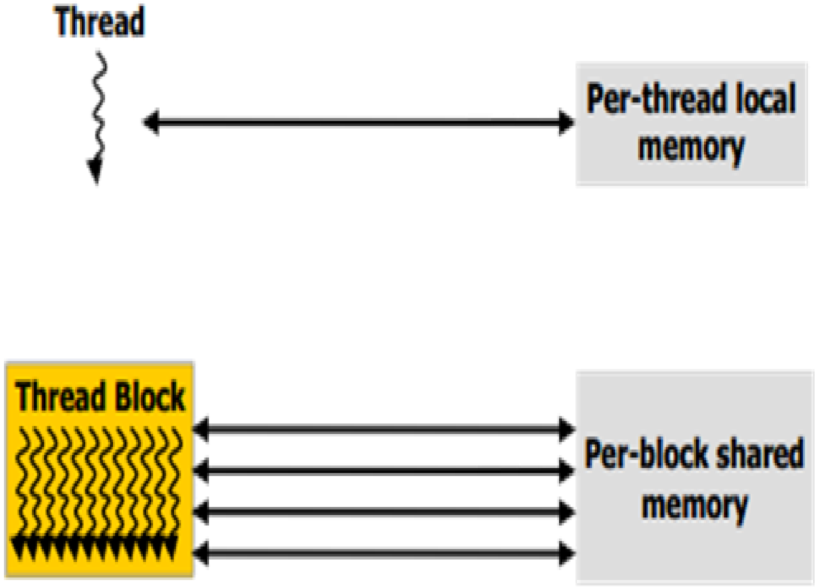


Sharing Data Between Threads

Terminology: within a block, threads share data via shared memory

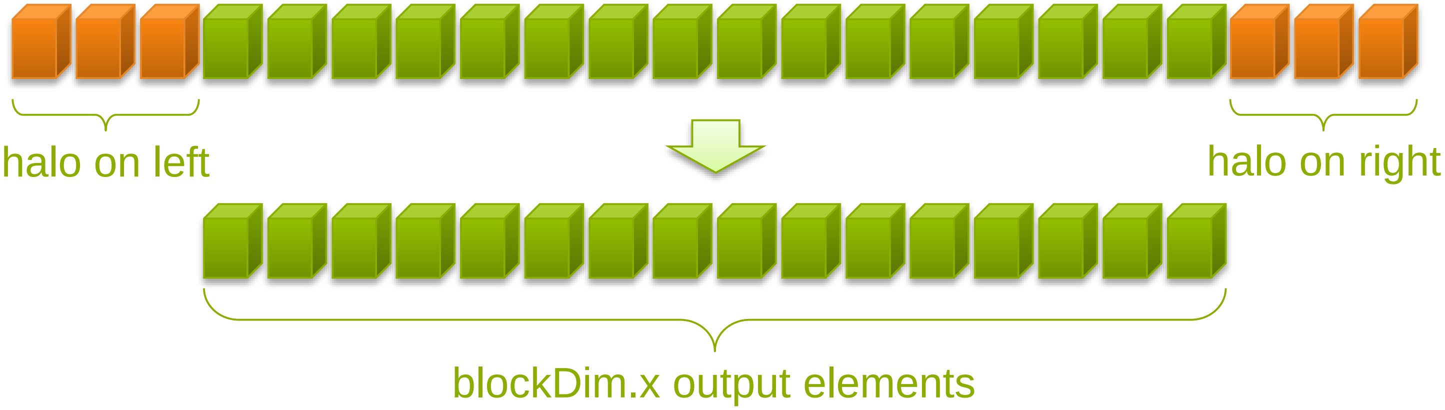
Declare using __shared__, allocated per block

Data is not visible to threads in other blocks



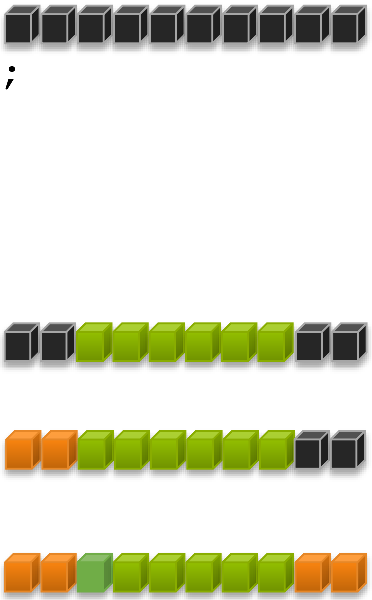
Implementing with shared memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory



Each block needs a halo of radius elements at each boundary

Stencil kernel



```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int global_id = threadIdx.x + blockIdx.x * blockDim.x;
  int local_id = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[local_id] = in[global_id];
  if (threadIdx.x < RADIUS) {
    temp[local_id - RADIUS] = in[global_id - RADIUS];
    temp[local_id + BLOCK_SIZE] =
      in[global_id + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
    __syncthreads();
```

Stencil kernel

```
 // Apply the stencil
    int result = 0;
    for(int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[local_id + offset];

    // Store the result
    out[global_id] = result;
}
```

Stencil: CUDA kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int global_id = threadIdx.x + blockIdx.x * blockDim.x;
    int local_id = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[local_id] = in[global_id];
    if (threadIdx.x < RADIUS) {
        temp[local_id – RADIUS] = in[global_id – RADIUS];
        temp[local_id + BLOCK_SIZE] = in[global_id + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for(int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[local_id + offset];

    // Store the result
    out[global_id] = result;
}
```

Stencil: kernel translation (1)

```
void stencil_1d(int *in, int *out, dim3 blockDim, dim3 blockIdx) {
    int temp[BLOCK_SIZE + 2 * RADIUS];
    int global_id;
    int local_id;
    THREAD_LOOP_BEGIN{
    global_id = threadIdx.x + blockIdx.x * blockDim.x;
    local_id = threadIdx.x + radius;
    // Read input elements into shared memory
    temp[local_id] = in[global_id];
    if (threadIdx.x < RADIUS) {
        temp[local_id – RADIUS] = in[global_id – RADIUS];
        temp[local_id + BLOCK_SIZE] = in[global_id + BLOCK_SIZE];
    }
    }THREAD_LOOP_END
    THREAD_LOOP_BEGIN{
    // Apply the stencil
    int result = 0;
    for(int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[local_id + offset];
    out[global_id] = result; // Store the result
    }THREAD_LOOP_END
}
```

A thread loop implicitly introduces a barrier synchronization among logical threads at its boundaries

Stencil: kernel translation (2)

```
void stencil_1d(int *in, int *out, dim3 blockDim, dim3 blockIdx) {
    int temp[BLOCK_SIZE + 2 * RADIUS];
    int global_id[];
    int local_id[];
    THREAD_LOOP_BEGIN{
    global_id[tid] = threadIdx.x + blockIdx.x * blockDim.x;
    local_id[tid] = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[local_id[tid]] = in[global_id[tid]];
    if (threadIdx.x < RADIUS) {
        temp[local_id[tid] - RADIUS] = in[global_id[tid] - RADIUS];
        temp[local_id[tid] + BLOCK_SIZE] = in[global_id[tid] + BLOCK_SIZE];
    }
    }THREAD_LOOP_END
    THREAD_LOOP_BEGIN{
    // Apply the stencil
    int result = 0;
    for(int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[local_id[tid] + offset];
    out[global_id[tid]] = result; // Store the result
    }THREAD_LOOP_END
}
```

We create an array of values for each local variable of the former CUDA threads.

Statements within thread loops access these arrays by loop index .

Benchmark

Used source code	Time (ms)	Slowdown wrt CUDA
CUDA[1]	3.41406	1
Translated TBB[2]	9.41103	2.76
Native sequential[3]	22.451	6.58
Native TBB[2]	14.129	4.14

1 – Not optimized CUDA code, memory copies included
2 - We wrapped the call to the function in a TBB parallel for
3 - Not optimized, naive implementation

Intel® Core™ i7-4771 CPU @ 3.50GHz
NVIDIA® Tesla K40 Kepler GK110B

Conclusion

- One of the most challenging aspects was obtaining the ASTs from CUDA source code
    - Solved including CUDA headers at compile time
    - Now we can match all CUDA syntax
- At the moment: kernel translation almost completed.
- Interesting results, but still a good amount of work to do.

Future work

- Handle CUDA API in the host code (i.e. cudaMallocManaged)
- Atomic operations in the kernel
- Adding vectorization pragmas

Thank you

GitHub https://github.com/HPC4HEP/CUDAtoCPP

References

Felice Pantaleo, “Introduction to GPU programming using CUDA”

(Backup) Example: __syncthread inside a while statement

```
while(j>0){

        foo();

        __syncthreads();

        bar();
}
```

```
bool newcond [];
thread_loop {
        newcond [tid] = (j[tid]>0);
}
LABEL: thread_loop{
        if(newcond[tid]) { foo(); }
}
thread_loop{
        if(newcond[tid] { bar(); }
}
thread_loop{
        newcond[tid] = (j[tid]>0);
        if(newcond[tid]) go = true;
}
if(go) gotoLABEL
```