SPMDfy

A Transpiler From CUDA to ISPC
The Tool was actually inspired by hipify-clang which translates CUDA -> HIP.
Table of Contents

1. ISPC 101
2. Saxpy Example
3. Building the Transpiler
4. SPMDfy Pipeline
5. Motivation for CFG
6. Spmdfy Pipeline
7. Current Status and Future Work
8. Questions?
ISPC 101

• **ISPC** is a compiler for a **C-like language** for **SPMD on SIMD architectures**
• Currently it support x86/64, ARM neon, PTX, Xeon Phi.
Let’s Do Saxpy
int main(){
    int n = 1024;
    double a = 1.0, b = 2.0;
    vector<float> x(n), y(n);
    for(int i = 0; i < n; i++){
        y[i] = a * x[i] + b;
    }
}

Saxpy - CPU
int main()
{
    int n = 1024;
    double a = 1.0, b = 2.0;
    vector<float> x(n), y(n);
    for(int i = 0; i < n; i+=8){
        vec8 x8 = load(x[i], x[i + 8]);
        vec8 y8 = load(y[i], y[i + 8]);
        y8 = a * x8 + b;
        store(y[i], y8);
    }
}
Saxpy - SIMD Intrinsics and Auto vectorization
// saxpy.ispc
export void saxpy(uniform float y[], uniform float x[],
    uniform double a, uniform double b, uniform int n){
    foreach(i = 0...n){
        y[i] = a * x[i] + b;
    }
}

// main.cpp
int main(){
    int n = 1024;
    double a = 1.0, b = 2.0;
    vector<float> x(n), y(n);
    ispc::saxpy(x.data(), y.data(), a, b, n);
}
__global__ void saxpy(uniform float y[], uniform float x[], uniform double a, uniform double b, uniform int n){
    for(int i = threadIdx.x; i < n; i += blockDim.x){
        y[i] = a * x[i] + b;
    }
}

int main(){
    int n = 1024;
    double a = 1.0, b = 2.0;
    vector<float> x(n), y(n);
    saxpy<<<1, n>>>(x.data(), y.data(), a, b, n);
}

Saxpy - CUDA
ISPC Execution Model

Gang

Program Instance 1
Program Instance 2
Program Instance 3
Program Instance 4
ISPC Execution Model

Program Instance 1

Program Instance 2

Program Instance 3

Program Instance 4
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Program Instance 1</td>
<td>Program Instance 2</td>
<td>Program Instance 3</td>
<td>Program Instance 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ISPC Execution Model
ISPC Execution Model
Program Instance 1  Program Instance 2  Program Instance 3  Program Instance 4
ISPC Targets

- host
- sse2-i32x4
- sse2-i32x8
- sse4-i32x4
- sse4-i32x8
- sse4-i16x8
- sse4-i8x16
- avx1-i32x4
- avx1-i32x8
- avx1-i32x16
- avx1-i64x4
- **avx2-i32x8**
- avx2-i32x16 ...

** ISA : avx2 **
** Mask size: 32 bits **
** Gang size: 8 **
Typical ISPC Build phases

Program.ispc

Source.cu/cpp
Typical ISPC Build phases

- `Program.ispc`
- `Program.h`
- `Source.cu/cpp`
- `Program.o`
- `a.out`
Typical ISPC Build phases

1. Source.cu/cpp
2. Program.h
3. Program.ispc
4. Source.o
5. Program.o
6. a.out
How to build the transpiler?
Enter the dragon
Clang provides a rich AST representation.
It allows developers to query information about every token in the code.
Preserves source information.
This enables developers to build great tools like linter, static analyzers, formatting tools, source translators etc.
Ways to build the tool

- Clang provides access to its AST through the following interfaces
  - *libClang* - A stable C interface, also has python interface
  - *libTooling* - A less stable C++ interface (recommended for complex tools)
Ways to build the tool

- I use LibTooling
- There are different approaches to build the tool
  - **Clang Plugin** - shared libraries
  - **Clang Tool** - standalone tool
SPMDfy Pipeline
SPMDfy Pipeline

1. Frontend
2. CFG
3. CodeGen
SPMDfy Pipeline - Pass

1. Locate AST Nodes
2. Hoist Shared Memory
3. Insert ISPC Nodes
SPMDfy Pipeline - Pass

Frontend → Workspace

Locate AST Nodes → Pass 1 → Pass 2 → Pass 3 → CodeGen

Hoist Shared Memory

Insert ISPC Nodes
Motivation for CFG
__global__ void kernel() {
    if (tid < n) {
    }
}

Let’s take an example
__global__ void kernel()
{
    if(tid < n) {
    } // reconv
}

Reconvergence
```c
__global__ void kernel(int a) {
    if (tid < n) {
    } // reconv
    a++;
}
```
__global__ void kernel(int a) {
    if (tid < n) {
    } // reconv
    a++;
}
__global__ void kernel(int a) {

    if (tid < n) {
    } // reconv

ISPC_BLOCK_START

    a++;

}
Why `a++` is not included?

```c
__global__ void kernel(int a)

ISPC_BLOCK_START

    if(tid < n){
    }
    // reconv

ISPC_BLOCK_START

    a++;

}  // reconv
```
Why `a++` is not included?
Current Status and Future Direction
Current State of the tool

- Features of CUDA supported
  - Atomic function
  - Shared memory - both dynamic and static
  - Syncthreads - also with complex control flow
  - Some CUDA Math libraries
  - Device Functions
  - There also a tool to convert compilation database of nvcc to cuda clang
Future Work

- Inline of Device function
- Dataflow analysis of code
- To detect partial and complete control flow
- More C++ stuff - part of the work involved translating from C++ -> C
SPMDfy and Experiments

- https://github.com/schwarzschild-radius/spmdfy
Questions?