CuPBoP: CUDA for Parallelized and Broad-range Processors

Ruobing Han (Georgia Institute of Technology)
Execute CUDA on non-NVIDIA devices
Software developers love CUDA
However, on the hardware side...
If CUDA can be executed on non-NVIDIA devices...

I: Other hardware vendors can benefit from the CUDA ecosystem.
II: Allow Single-Kernel-Multiple-Device on heterogeneous system.
III: Lower the Total cost of ownership (TCO).

....
Background: CUDA on NVIDIA GPUs
Related work:

CUDA source code
- host code
- kernel code

Clang frontend
- LLVM IR
- NVVM IR

NVPTX codegen
- PTX assembly
- fat binary

CUDA compilation pipeline

src2src translators:
- DPCT (Intel), hipify (AMD)

reverse-engineering:
- Ocelot
What we want?

Our solution: CuPBoP

CUDA source code

host code

kernel code

Clang frontend

LLVM IR

NVVM IR

NVPTX codegen

PTX assembly

fat binary

CUDA compilation pipeline

Insight:
Translate CUDA programs on LLVM/NVVM IR level.

Pros:
[1] open source for all parts.
Instead of translating CUDA to portable languages, we make CUDA a portable language.
CuPBoP framework

Implement CUDA APIs on non-NVIDIA devices.

Compilation part

- CUDA program
  - kernel code
  - host code
- Compile to NVVM IR
- Compile to LLVM IR
- Kernel transformation
- Host transformation
- linker
- Runtime library
- Executable file

Apply compilation transformations.
Implement CUDA APIs for non-NVIDIA devices.

By changing the link path, CUDA code can be executed on CPUs without modifications.

To support a new CUDA API, the developers only need to implement the new API in the library.
CuPBoP compilation

When the **targeted architectures are significantly different with NVIDIA GPUs**...

For example, CPUs are designed for multiple program multiple data (MPMD)

```c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    c[id] = a[id] + b[id];
}

int main() {
    ...
    vecAdd<<<64, 1024>>>(d_a, d_b, d_c, n);
    ...
}
```

[1] There are 64K CUDA threads. CPUs cannot support that many threads easily.

[2] The workload for each thread is too lightweight for CPU cores.
CuPBBoP compilation

Compilation transformations

```c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    c[id] = a[id] + b[id];
}

int main()
{
    ...
    vecAdd<<<64, 1024>>>(d_a, d_b, d_c, n);
    ...

    int blockIdx;
    int blockDim;
    void cpu_vecAdd(double *a, double *b, double *c, int n)
    {
        for(int threadIdx = 0; threadIdx < 1024; threadIdx++) {
            int id = blockIdx*blockDim+threadIdx;
            c[id] = a[id] + b[id];
        }
    }
```
SPMD->MPMD (related work)

Step 1:
Analyze the Parallel Region\[1\] (the regions between barriers that must be executed by all the threads before proceeding to the next region.)

SPMD->MPMD (related work)

Step2:
Wrap each Parallel Region with a single-layer for-loop.
Limitation

The previous works assume each barrier is reached by all threads. However, CUDA also has barriers for CUDA warps.

Only the first warp reaches the if-body.

The warp shuffle function contains an implicit warp-level barrier.

```c
int val = 1;
if (threadIdx.x < 32) {
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(-1, val, offset);
} 
```
Use nested-layer loops.
For inter/intra-warp loops.

```c
int shfl_arr[32]; int val[b_size];
bool flag[32];
for (int wid = 0; wid < b_size / 32; wid++) {
    for (int tx = 0; tx < 32; tx++)
        val[wid * 32 + tx] = 1;
    for (int tx = 0; tx < 32; tx++)
        flag[tx] = (wid * 32 + tx) < 32;
    // loop peeling
    if (flag[0]) {
        for (int offset = 16; offset > 0; offset /= 2) {
            for (int tx = 0; tx < 32; tx++)
                shfl_arr[tx] = val[wid * 32 + tx];
            for (int tx = 0; tx < 32; tx++)
                if (tx + offset < 32)
                    val[wid * 32 + tx] += shfl_arr[tx + offset];
        }
    }
}
```
# Experiments: hardware backend

 CuPBoP can execute CUDA on x86, AArch64, and RISC-V backends.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Compilation requirement</th>
<th>Runtime requirement</th>
<th>ISA support</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPC++</td>
<td>DPC++</td>
<td>DPC++</td>
<td>x86</td>
</tr>
<tr>
<td>HIP-CPU</td>
<td>C++17</td>
<td>TBB(&gt;=2020.1-2), pthreads</td>
<td>x86 AArch64 RISC-V</td>
</tr>
<tr>
<td>CuPBoP</td>
<td>LLVM</td>
<td>pthreads</td>
<td>x86 AArch64 RISC-V</td>
</tr>
</tbody>
</table>

Theoretically, CuPBoP supports all backends that LLVM supports.  

**Thanks to all of you! LLVM developers!**
## Experiments: benchmark coverage

<table>
<thead>
<tr>
<th></th>
<th>DPC++</th>
<th>HIP-CPU</th>
<th>CuPBoP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rodinia-GPU</td>
<td>56.5</td>
<td>56.5</td>
<td>73.9</td>
</tr>
<tr>
<td>Crystal</td>
<td>0</td>
<td>76.9</td>
<td>100</td>
</tr>
</tbody>
</table>

Texture memory and other features are required to achieve higher coverage.
CuPBoP is 7.83x/4.25x times faster than HIP-CPU on AArch64/RISC-V. CuPBoP is 2.28x faster than DPC++ and 3.36x faster than HIP-CPU on x86.

Optimizations:
- LLVM O3 optimization
- Lock free queue
- Lightweight kernel coalescing
Unsolved problems and future works

CUDA programs are close to the upper bound (green dots and curves)

The translated CPU programs are much below the bound (red/blue dots and curves).
Unsolved problems and future works

... 
uint32_t priv_hist[256];
int index = threadIdx.x;
while (index < num_pixels) {
    priv_hist[pixels[index]]++;
    index += blockDim.x;
}
... 

Good memory access pattern for NVIDIA GPUs!
Unsolved problems and future works

...  
uint32_t priv_hist[256];
int index = threadIdx.x;
while (index < num_pixels) {
    priv_hist[pixels[index]]++;
    index += blockDim.x;
}
...

Poor memory access pattern for CPUs

for(tid=0; tid<blockDim.x; tid++) {
    ...
    uint32_t priv_hist[256];
    int index = tid;
    while (index < num_pixels) {
        priv_hist[pixels[index]]++;
        index += blockDim.x;
    }
    ...
}
future works

VS
Thanks for your time!

CupBoP is an open-source project on Github. We welcome any kinds of contribution & feedback.