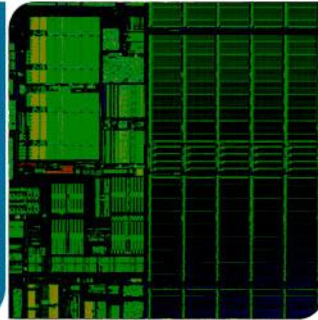
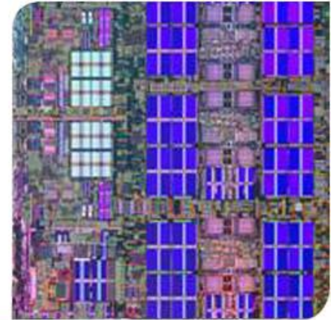
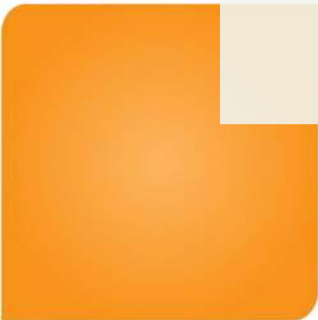
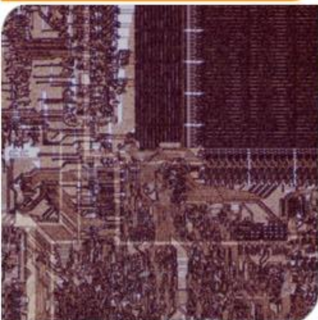



CuPBoP: CUDA for Parallelized and Broad-range Processors



Ruobing Han (Georgia Institute of Technology)



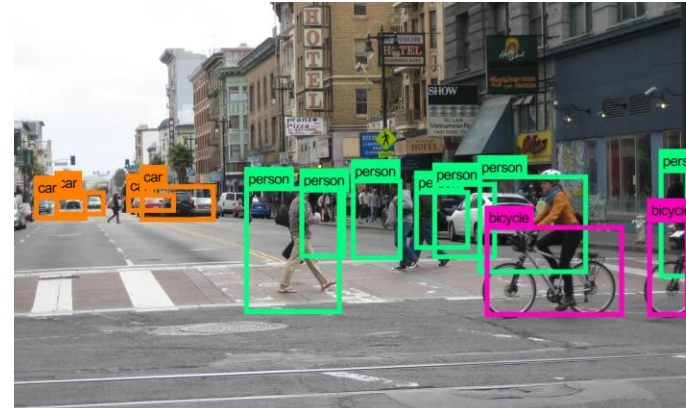
Georgia Tech

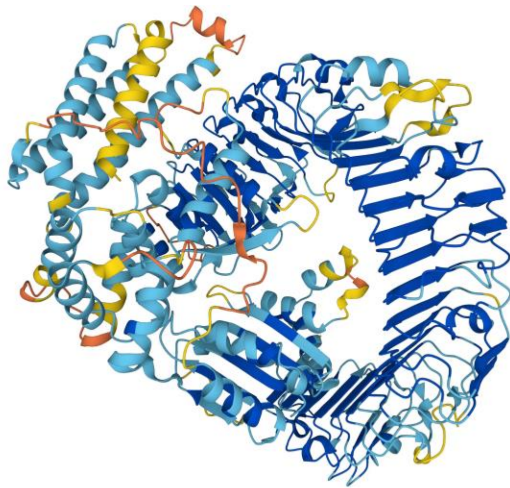
comparch

Execute CUDA on non-NVIDIA devices

Software developers love CUDA

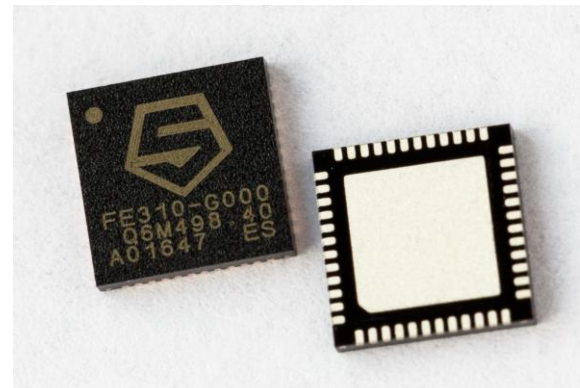
*





However, on the hardware side...

*



If CUDA can be executed on non-NVIDIA devices... *

I: Other hardware vendors can benefit from the CUDA ecosystem.

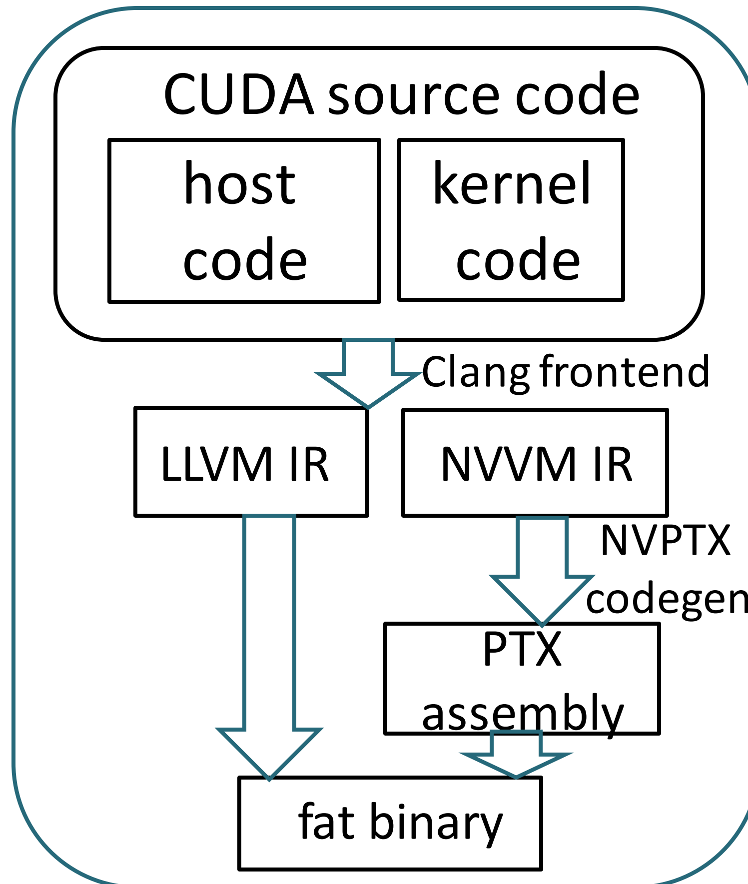
II: Allow Single-Kernel-Multiple-Device on heterogeneous system.

III: Lower the Total cost of ownership (TCO).

....

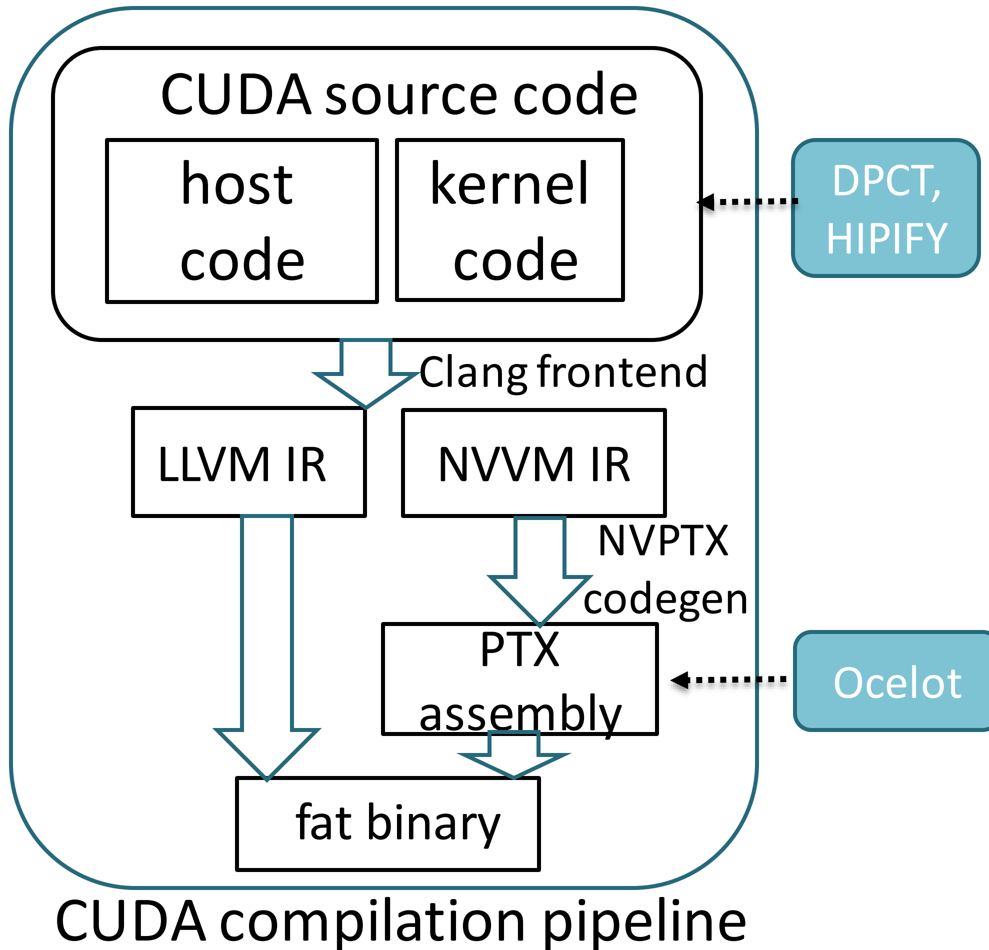
Background: CUDA on NVIDIA GPUs

*



Related work:

*



CUDA compilation pipeline

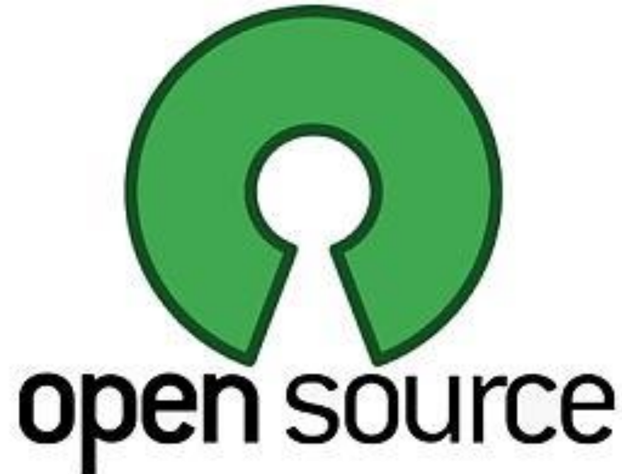
src2src translators:
DPCT (Intel), hipify(AMD)

reverse-engineering:
Ocelot

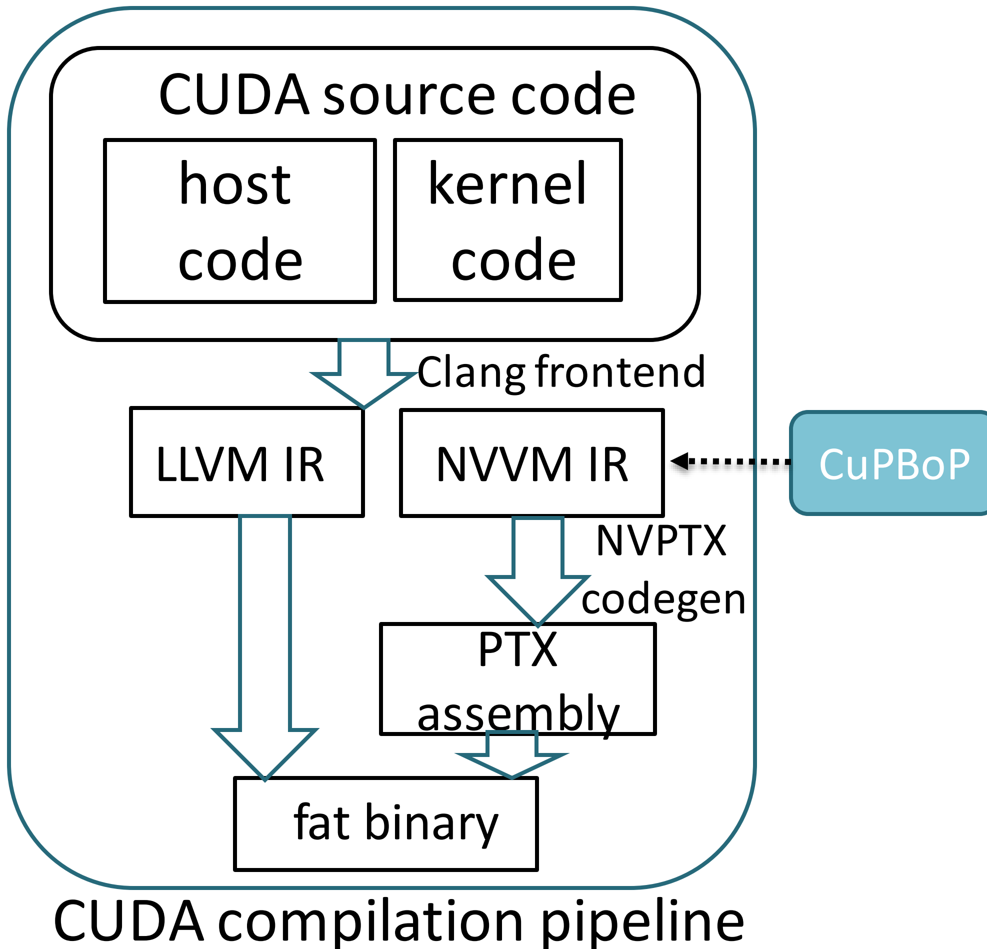
What we want?

*

[1] Get rid of manually modifications.
[2] Scalability to new CUDA updates.
[3] Only rely on open source toolchains.



Our solution: CuPBoP



CUDA compilation pipeline

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

Pros:
[1] open source for all parts.
[2] avoid complicated high-level languages.
[3] scalable to new CUDA features.

*

Instead of translating CUDA to portable languages, we make CUDA a portable language.

CuPBoP framework

Implement CUDA APIs on non-NVIDIA devices.

CUDA program
kernel code
host code

Compilation part
Compile to NVVM IR
Kernel transformation
Compile to LLVM IR
Host transformation

runtime part
Runtime library

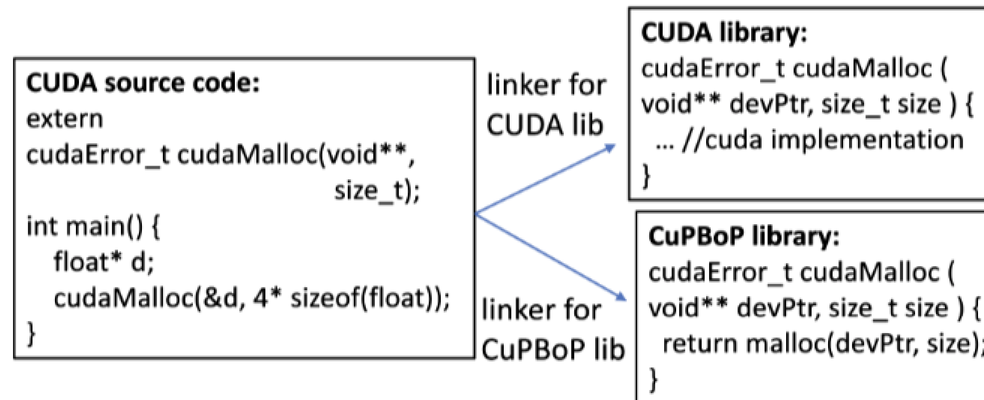
linker

Executable file

Apply compilation transformations.

CuPBoP runtime

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)

```
1   __global__ void vecAdd(double *a, double *b, double *c, int n)
2   {
3       int id = blockIdx.x*blockDim.x+threadIdx.x;
4       c[id] = a[id] + b[id];
5   }
6   int main() {
7       ...
8       vecAdd<<<64, 1024>>>(d_a, d_b, d_c, n);
9       ...
10  }
```

[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.

CuPBoP compilation

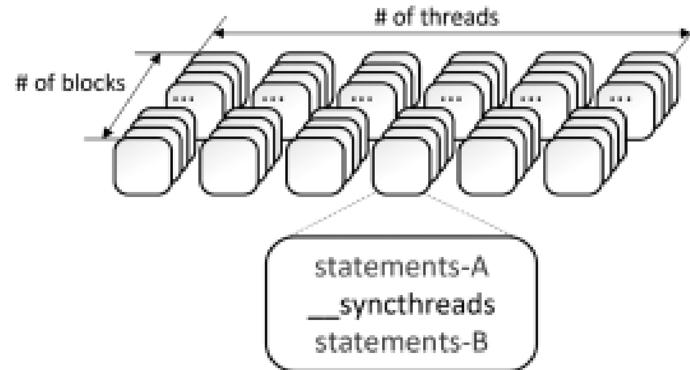

```
1   __global__ void vecAdd(double *a, double *b, double *c, int n)
2   {
3       int id = blockIdx.x*blockDim.x+threadIdx.x;
4       c[id] = a[id] + b[id];
5   }
6   int main() {
7       ...
8       vecAdd<<<64, 1024>>>(d_a, d_b, d_c, n);
9       ...
10  }
```

Compilation transformations

```
1   int blockIdx;
2   int blockDim;
3   void cpu_vecAdd(double *a, double *b, double *c, int n)
4   {
5       for(int threadIdx = 0; threadIdx < 1024; threadIdx++) {
6           int id = blockIdx*blockDim+threadIdx;
7           c[id] = a[id] + b[id];
8       }
9   }
```

[1] Only 64 CPU threads are required;
[2] The workloads are much heavier, which is friendly to MPMD model.

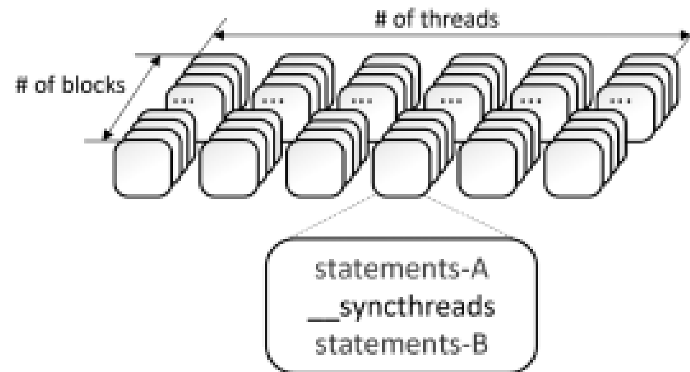
SPMD->MPMD (related work)



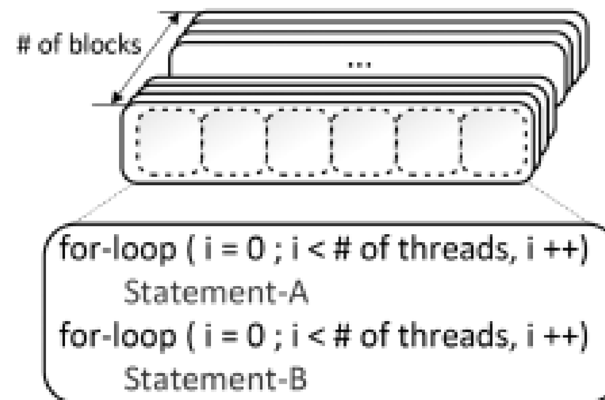
Step1:
Analyze the **Parallel Region**[1] (the regions between barriers that must be executed by all the threads before proceeding to the next region.)

[1] Jääskeläinen, Pekka, et al. "pocl: A performance-portable OpenCL implementation." *International Journal of Parallel Programming* 43.5 (2015): 752-785.

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.

```
1  int val = 1;
2  if (threadIdx.x < 32) {
3    for (int offset = 16; offset > 0; offset /= 2)
4        val += __shfl_down_sync(-1, val, offset);
5  }
```

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

CuPBoP solutions

Use nested-layer loops.
For inter/intra-warp loops.

inter-warp loop (line 3)

intra-warp loop (lines 4, 6, 11, 13)

```
1  int shfl_arr[32]; int val[b_size];
2  bool flag[32];
3  for (int wid = 0; wid < b_size / 32; wid++) {
4    for (int tx = 0; tx < 32; tx++)
5      val[wid * 32 + tx] = 1;
6    for (int tx = 0; tx < 32; tx++)
7      flag[tx] = (wid * 32 + tx) < 32;
8    // loop peeling
9    if (flag[0]) {
10     for (int offset = 16; offset > 0; offset /= 2) {
11       for (int tx = 0; tx < 32; tx++)
12         shfl_arr[tx] = val[wid * 32 + tx];
13       for (int tx = 0; tx < 32; tx++)
14         if (tx + offset < 32)
15           val[wid * 32 + tx] += shfl_arr[tx + offset];
16     }
17   }
18 }
```

Experiments: hardware backend

Framework	Compilation requirement	Runtime requirement	ISA support
DPC++	DPC++	DPC++	x86
HIP-CPU	C++17	TBB(>=2020.1-2), pthreads	x86 AArch64 RISC-V
CuPBoP	LLVM	pthreads	x86 AArch64 RISC-V

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

Theoretically, CuPBoP supports all backends that LLVM supports.

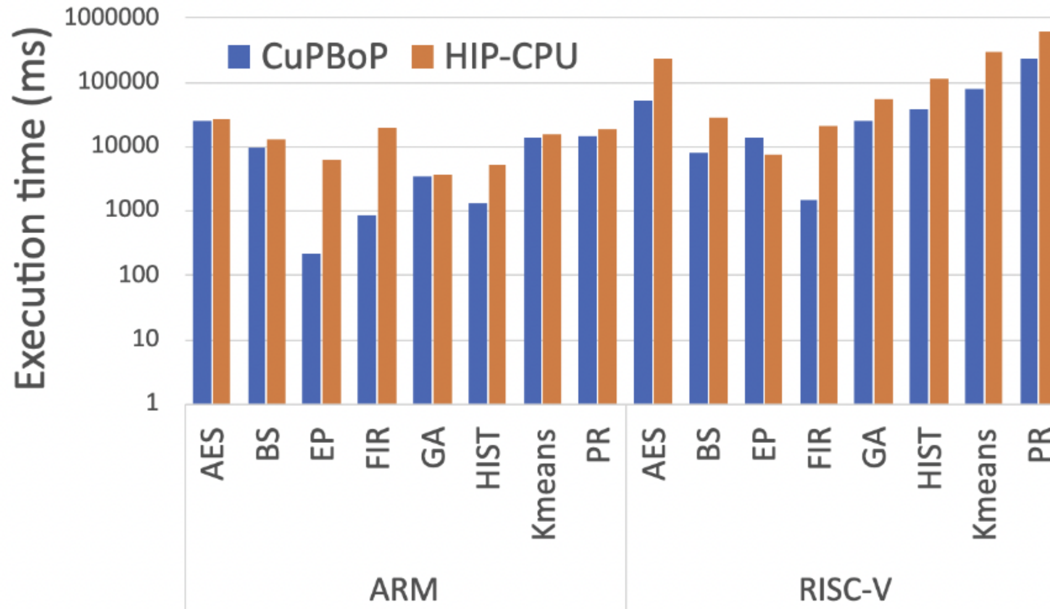
Thanks to all of you! LLVM developers!

Experiments: benchmark coverage

	DPC++	HIP-CPU	CuPBoP
Rodinia-GPU	56.5	56.5	73.9
Crystal	0	76.9	100

Texture memory and other features are required to achieve higher coverage.

Experiments: performance

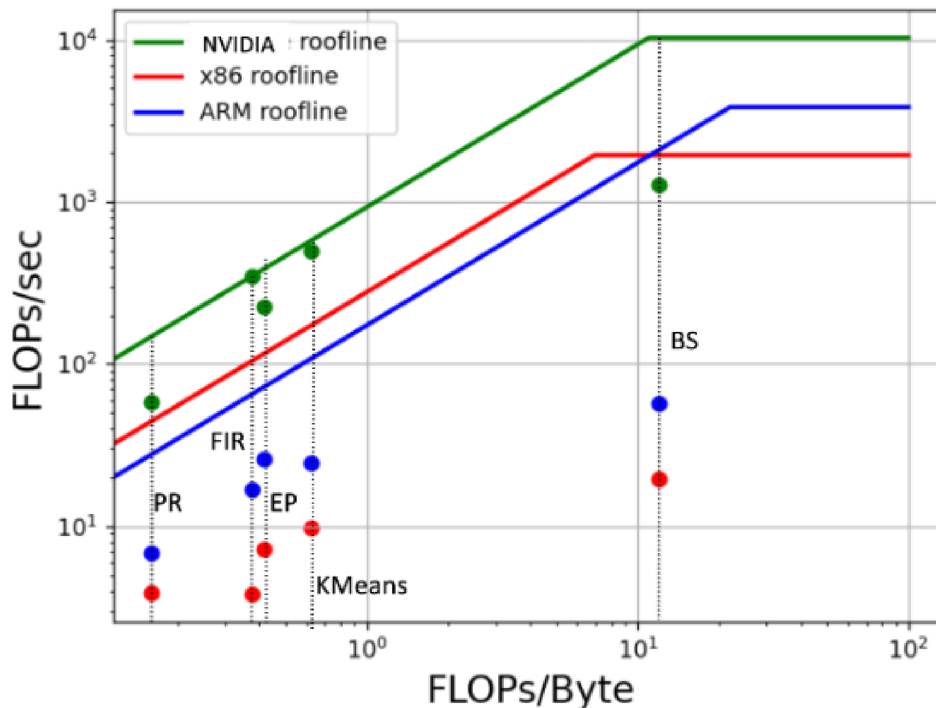


CuPBoP is 7.83x/4.25x times faster than HIP-CPU on AArch64/RISC-V.
CuPBoP is 2.28x faster than DPC++ and 3.36x faster than HIP-CPU on x86.

Optimizations:

- LLVM O3 optimization
- Lock free queue
- Lightweight kernel coalescing

Unsolved problems and future works

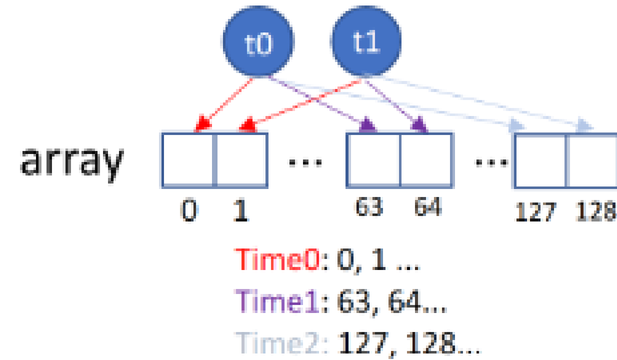


CUDA programs are close to the upper bound (green dots and curves)

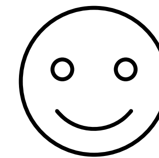
The translated CPU programs are much below the bound (red/blue dots and curves).

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;
}
...
```

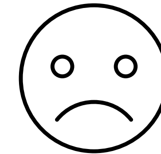


Good memory access pattern for NVIDIA GPUs!



Unsolved problems and future works

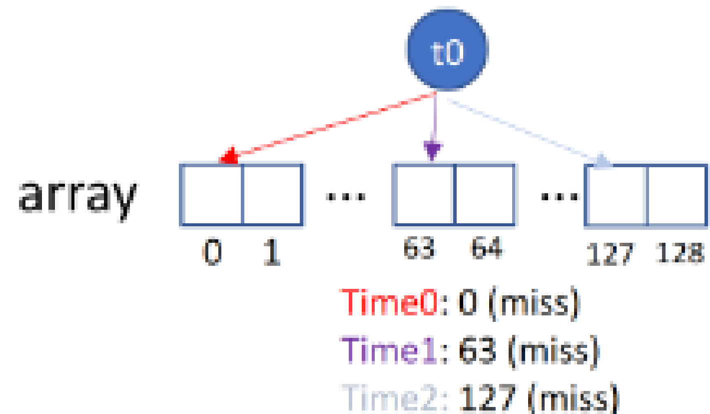

```
...
uint32_t priv_hist[256];
int index = threadIdx.x;
while (index < num_pixels) {
  priv_hist[pixels[index]]++;
  index += blockDim.x;
}
...
```



Poor memory access pattern for CPUs

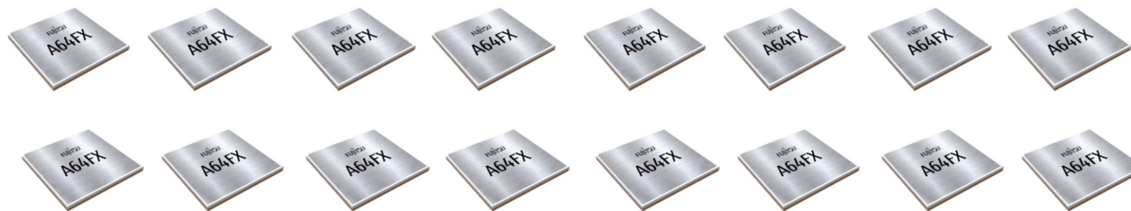
compilation transformation (to CPUs)

```
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



Q&A

Thanks for your time!

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

CuPBoP project

Paper for compilation transformations

Paper for CuPBoP