MANCHESTER
1824

The University of Manchester

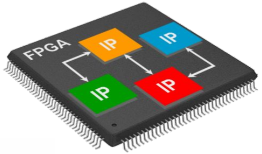
# Running Parallel Bytecode Interpreters on Heterogeneous Hardware

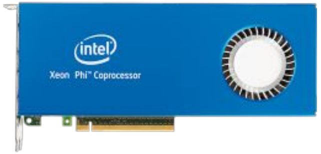
Juan Fumero, Athanasios Stratikopoulos, Christos Kotselidis
MoreVMs 2020
<juanfumero@acm.org>
University of Manchester

# Outline

- Motivation
- Parallel Bytecode Interpreter
    - Initial implementation
    - Parallel Implementation
    - Multiple-heap configuration
- Initial Results
- Takeaways

# Heterogeneous Hardware is Everywhere





# Motivation

**RQ: Heterogeneous systems are everywhere.**

**a) Can we run an Interpreter on multiple Heterogeneous Devices?**
**b) Can we increase performance?**

- Accelerate components of the actual VMs
    - Garbage Collection (GC)

[US Pattent 2010 0082930 A1] GPU Assisted Garbage Collection (AMD)
[ISMM'12] Offloading Garbage Collection on GPUs
[CASES'16] Generational GCs on Integrated GPUs (FastCollect)

# Motivation

**RQ: Heterogeneous systems are everywhere.**

**a) Can we run an Interpreter on multiple Heterogeneous Devices?**
**b) Can we increase performance?**

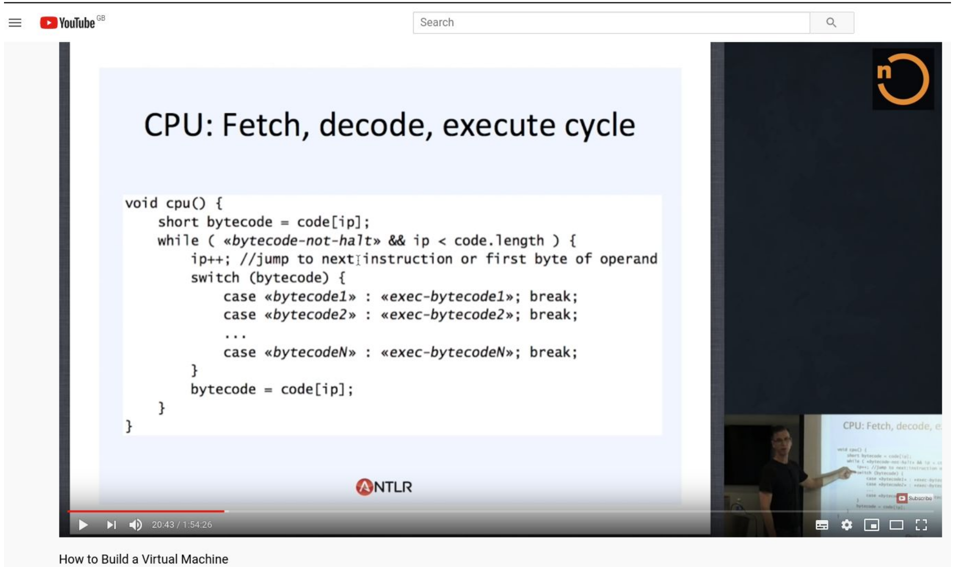
- Accelerate components of the actual VMs
  - GC?
  - **Bytecode interpreters**

In this work, we focus on the bytecode interpreter

[OOPSLA’19] CUDA Single Thread Interpreters

# Bytecode Interpreter

Subset of Java bytecodes → Toy example, but simple and powerful enough to start computing some workloads



CPU: Fetch, decode, execute cycle

```
void cpu() {
    short bytecode = code[ip];
    while ( «bytecode-not-halt» && ip < code.length ) {
        ip++; //jump to next instruction or first byte of operand
        switch (bytecode) {
            case «bytecode1» : «exec-bytecode1»; break;
            case «bytecode2» : «exec-bytecode2»; break;
            ...
            case «bytecodeN» : «exec-bytecodeN»; break;
        }
        bytecode = code[ip];
    }
}
```

How to Build a Virtual Machine

Example BC Interpreter by Professor Terence Parr

Simple stack-based machine

We extend it with more bytecodes and port to OpenCL

https://www.youtube.com/watch?v=OjaAToVkoTw

# Bytecode Interpreter

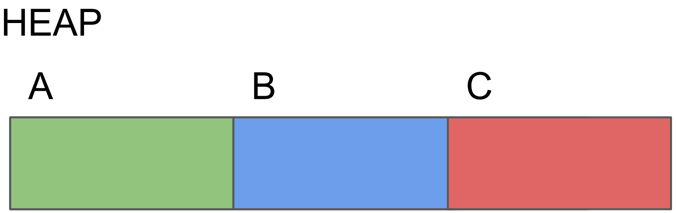
- Arithmetic Operations: IDIV, IADD, IMUL, ISUB
- Bitwise: RSHIFT, LSHIFT
- Comparisons: ILT, IEQ
- Memory: STORE, LOAD, GSTORE_INDEXED, GLOAD_INDEXED,
- Control Flow: BR, BRT, BRF, HALT, RET, CALL
- Interpreter Control: POP, DUP, ICONST1, ICONST <n>
- Auxiliar: PRINT

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



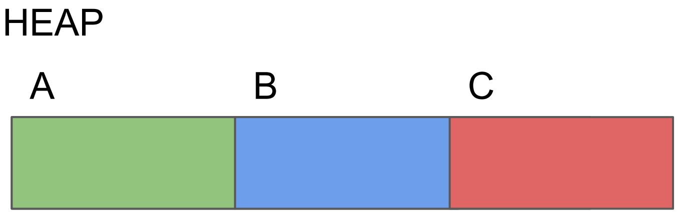
HEAP

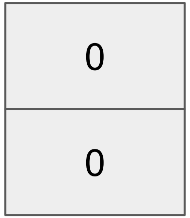
A B C

0

# Bytecode Interpreter

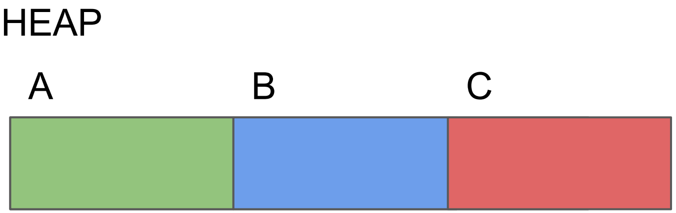
```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



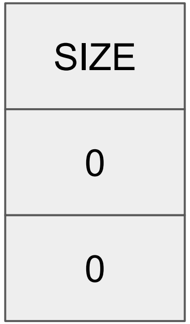


# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```


HEAP

A B C

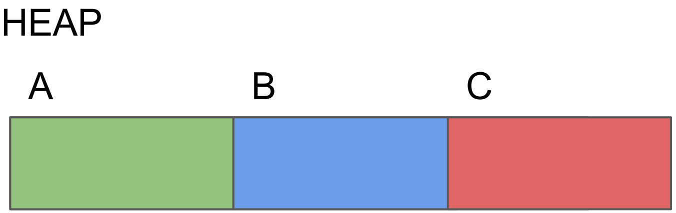

SIZE

0

0

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

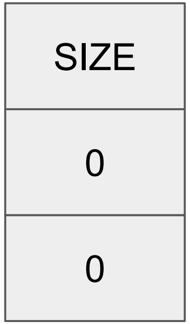


HEAP

A

B

C



SIZE

0

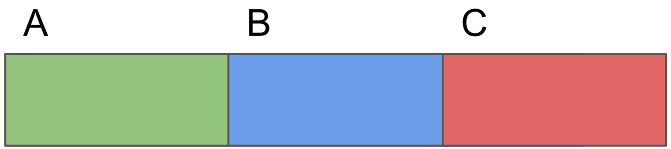
0

SIZE == 0? GOTO 23: next

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

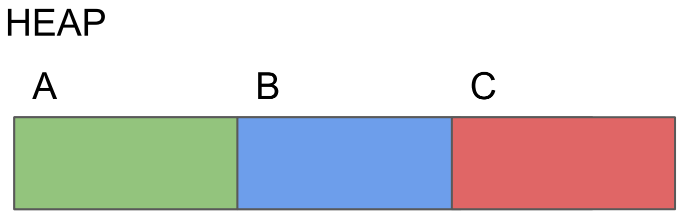
HEAP



0

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

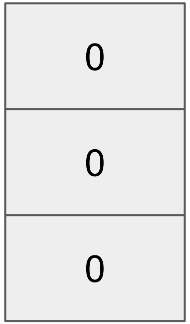


HEAP

A

B

C



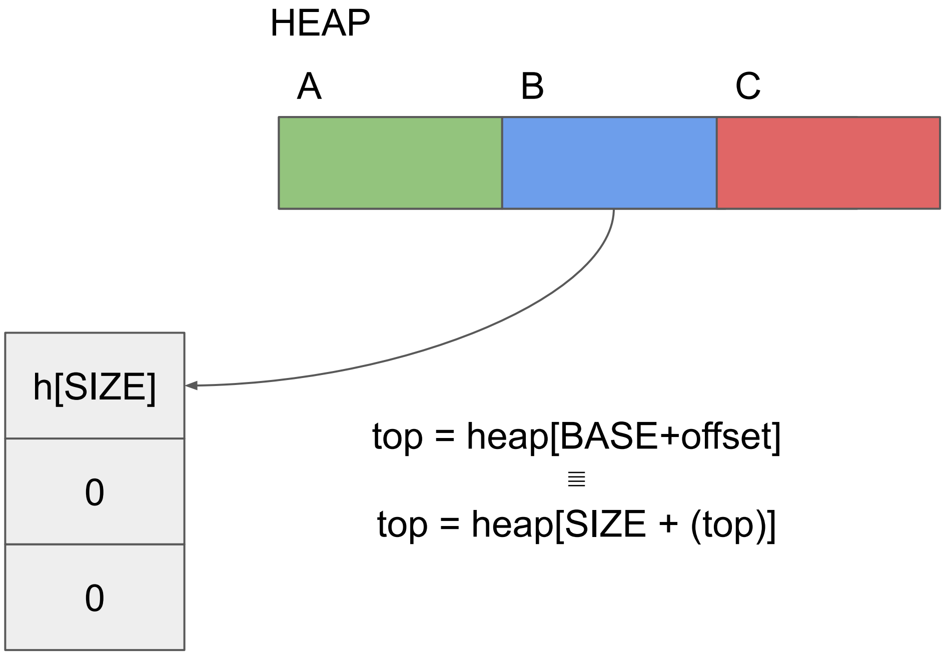
0

0

0

# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



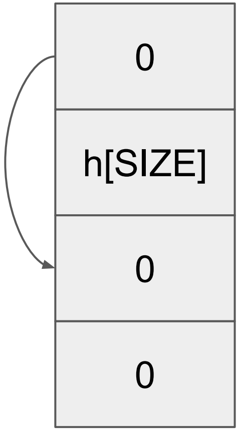
top = heap[BASE+offset]

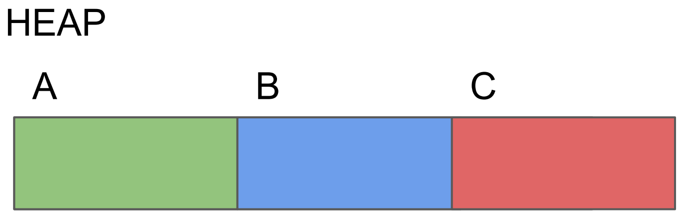
≡

top = heap[SIZE + (top)]

# Bytecode Interpreter

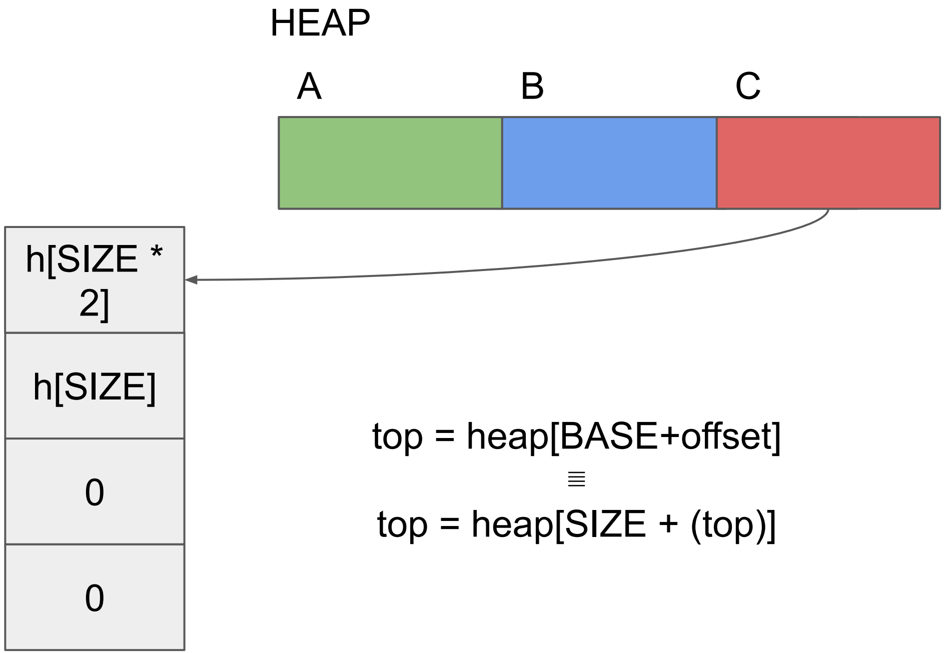
```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```





# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



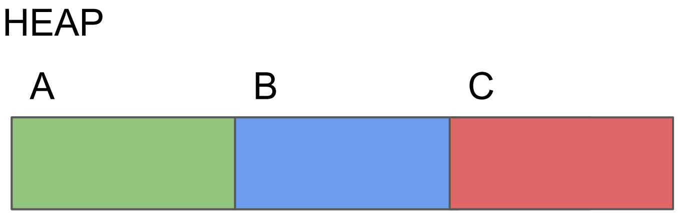
top = heap[BASE+offset]

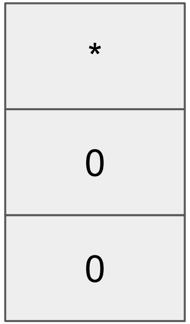
≡

top = heap[SIZE + (top)]

# Bytecode Interpreter

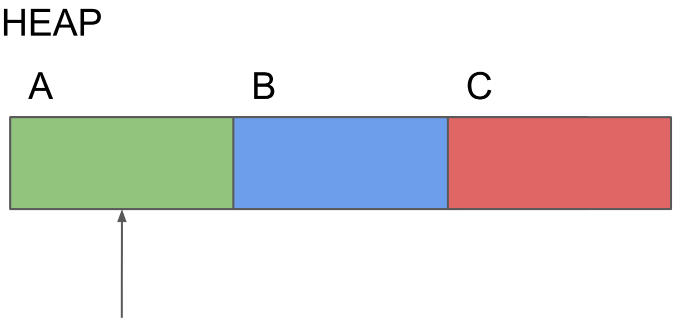
```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```





# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



heap[BASE+offset] = top

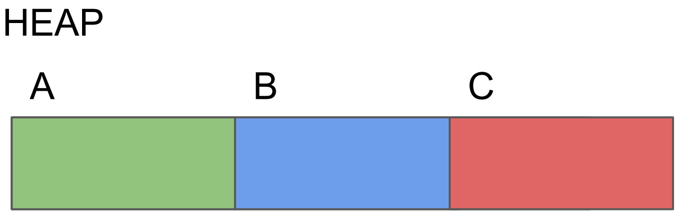
=

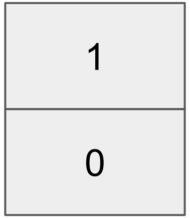
heap[BASE + (top-1)] = top

0

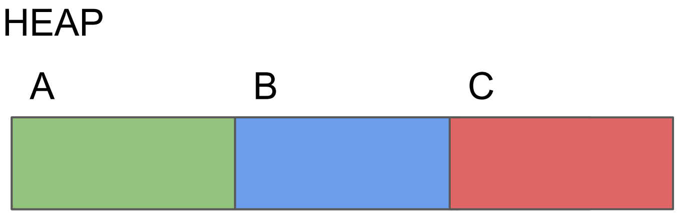
# Bytecode Interpreter

```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

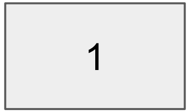




# Bytecode Interpreter

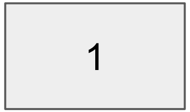


```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```

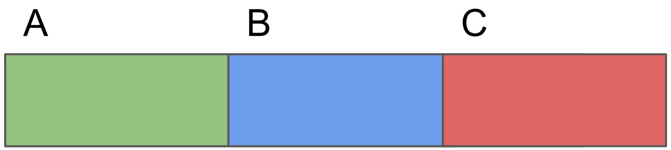


# Bytecode Interpreter

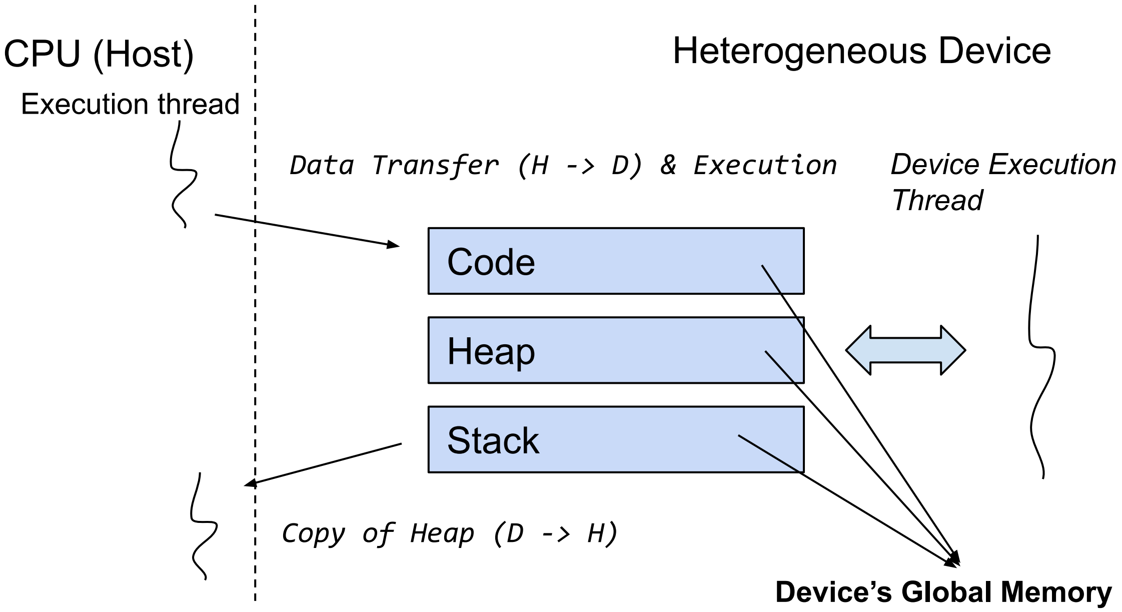
```
// Expressing vector multiplication
ICONST, 0,
DUP,
ICONST, SIZE,
IEQ,
BRT, 23,
DUP,
DUP,
GLOAD_INDEXED, SIZE,
LOAD, 1,
GLOAD_INDEXED, SIZE * 2,
IMUL,
GSTORE_INDEXED, BASE,
ICONST1,
IADD,
BR, 2,
POP,
HALT
```



HEAP



# OpenCL Bytecode Interpreter



CPU (Host)

Heterogeneous Device

Execution thread

*Data Transfer (H -> D) & Execution*

*Device Execution Thread*

Code

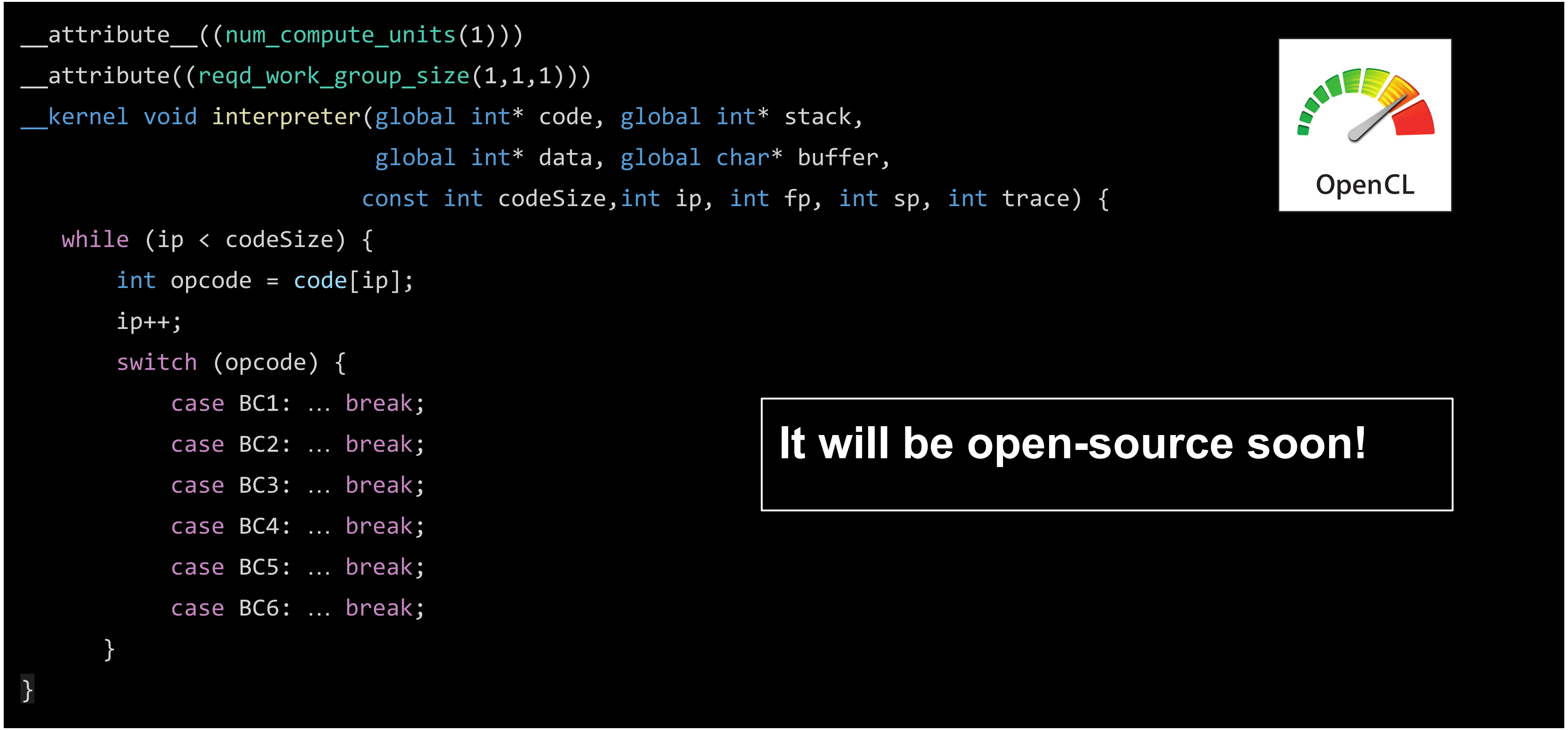
Heap

Stack

*Copy of Heap (D -> H)*

**Device's Global Memory**

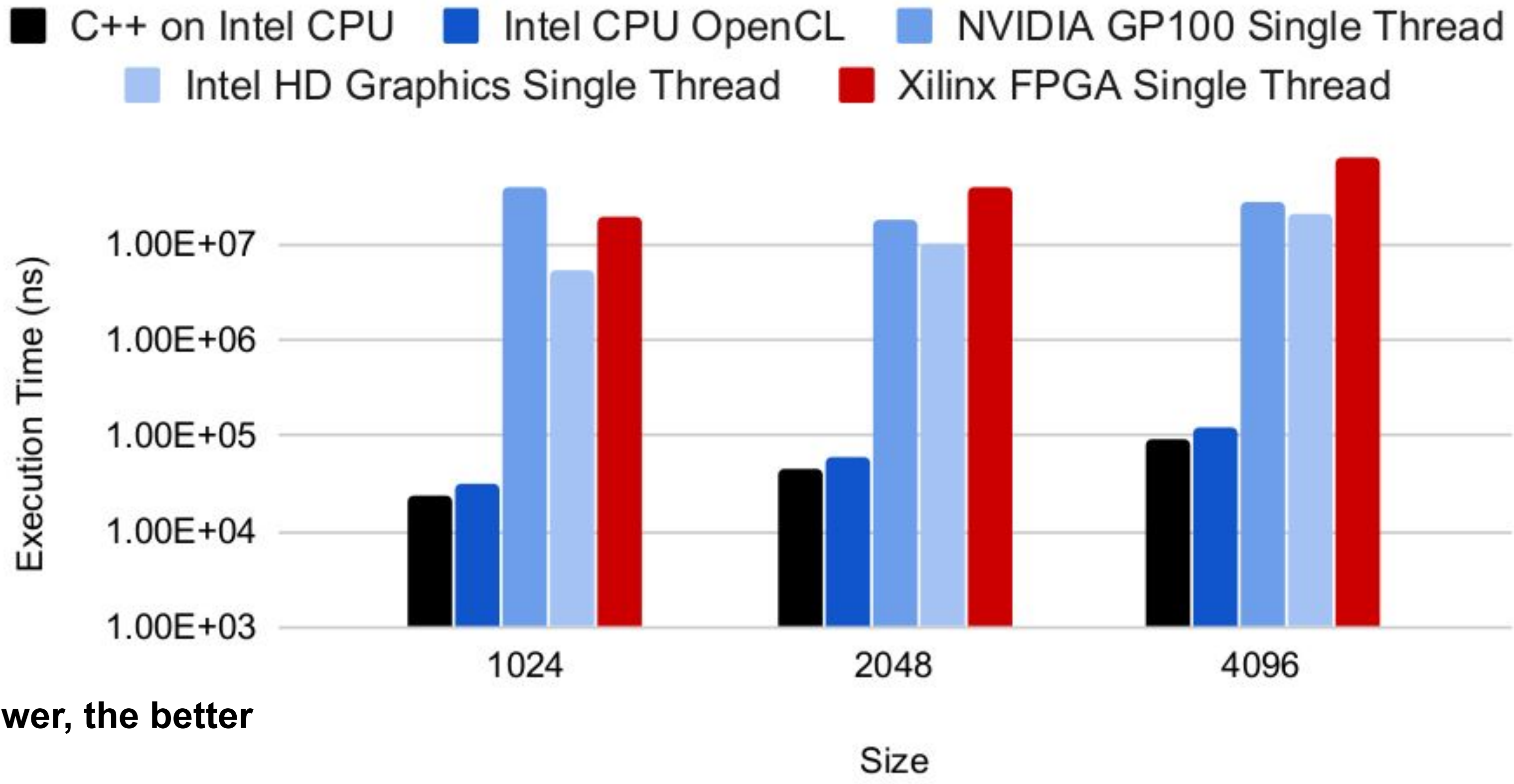
# OpenCL Bytecode Interpreter



```
__attribute__((num_compute_units(1)))
__attribute((reqd_work_group_size(1,1,1)))
__kernel void interpreter(global int* code, global int* stack,
                          global int* data, global char* buffer,
                         const int codeSize,int ip, int fp, int sp, int trace) {
   while (ip < codeSize) {
       int opcode = code[ip];
       ip++;
       switch (opcode) {
           case BC1: … break;
           case BC2: … break;
           case BC3: … break;
           case BC4: … break;
           case BC5: … break;
           case BC6: … break;
      }
}
```

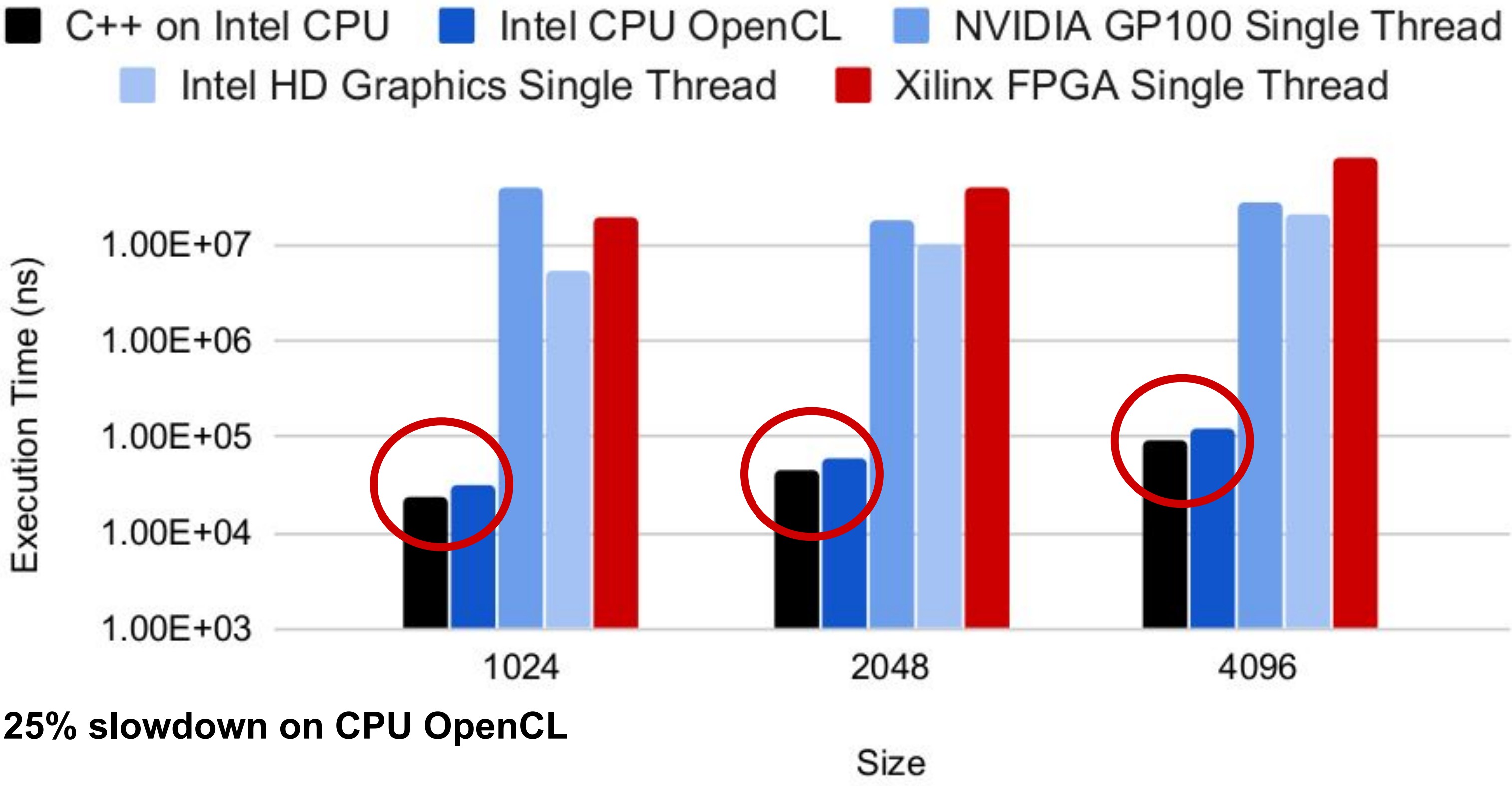
**It will be open-source soon!**

# Initial Results



**The lower, the better**

# Initial Results



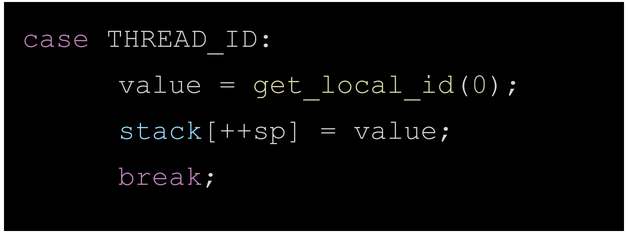
**Only 25% slowdown on CPU OpenCL**

# Can we do better?

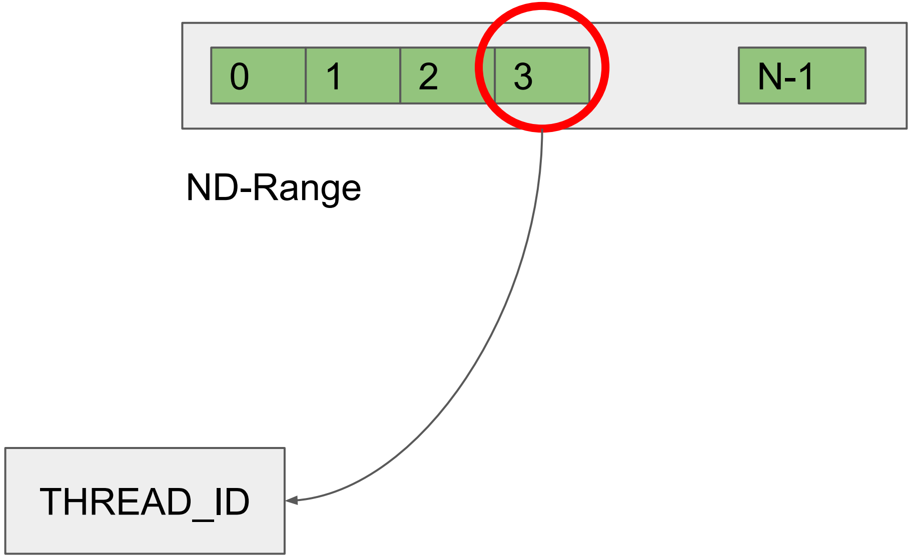
# Optimising for Heterogeneous Hardware

A) Parallel Interpreter through the introduction of Thread-Identifier
B) Memory regions (tier memory)

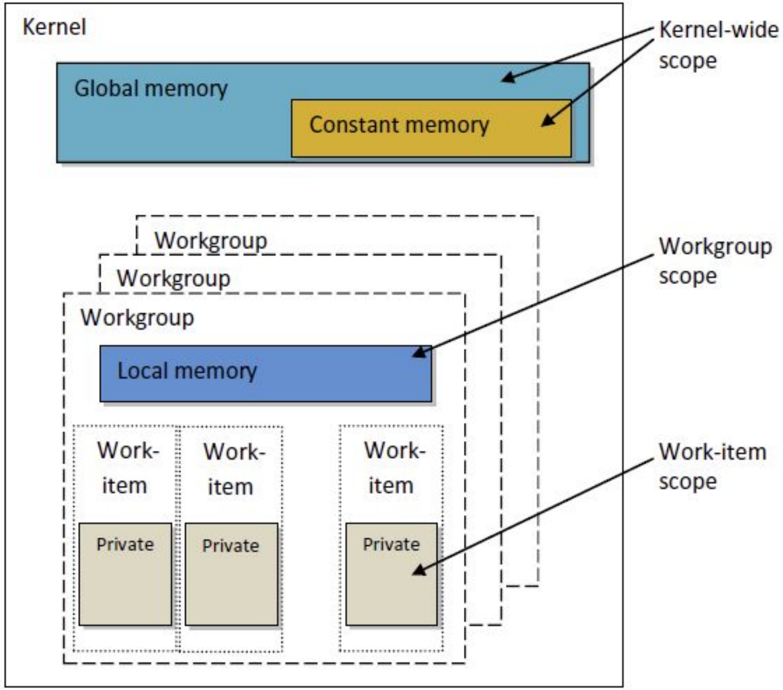
# A) Thread-ID in the Interpreter



```
case THREAD_ID:
      value = get_local_id(0);
      stack[++sp] = value;
      break;
```



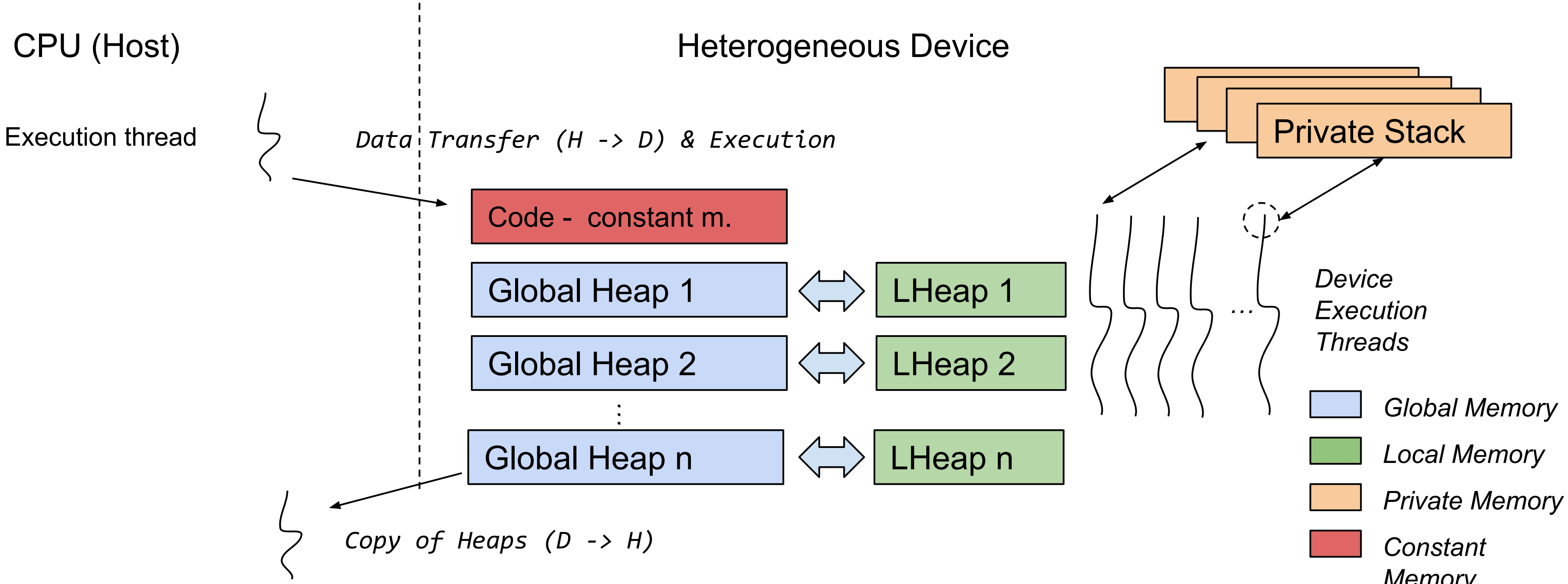
# Understanding OpenCL MM



Source: https://www.mql5.com/en/articles/407

- Exploit Memory Regions

# B) Memory Regions for out BC-Interpreter



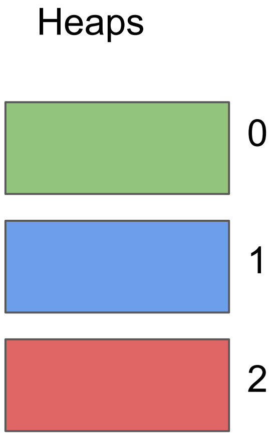
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector multiplication
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```

- One stack per thread
- Code in constant memory
- Multiple Global heaps
- Stack in private memory

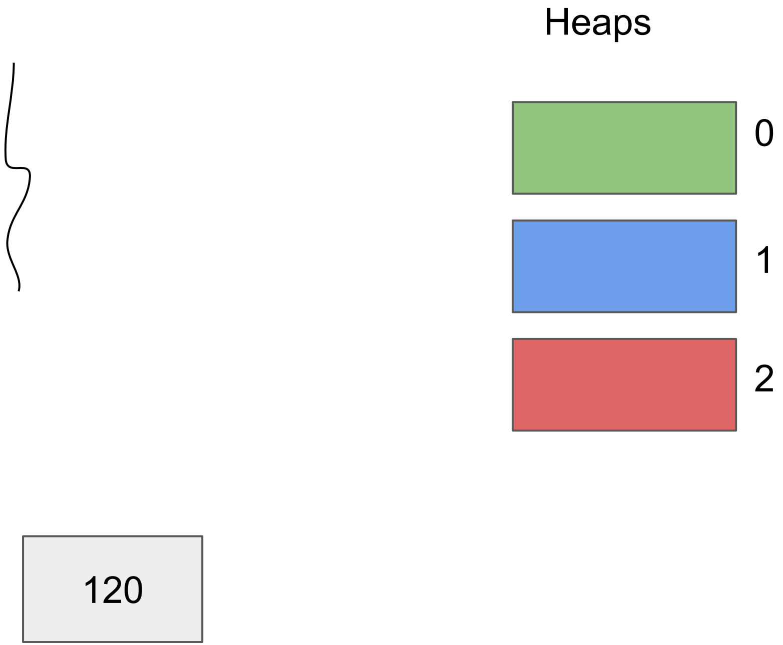
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



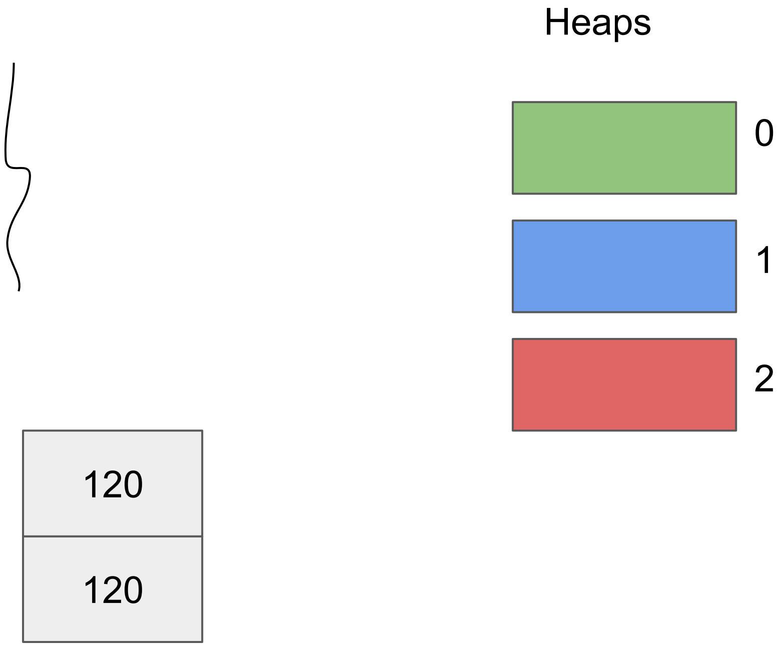
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



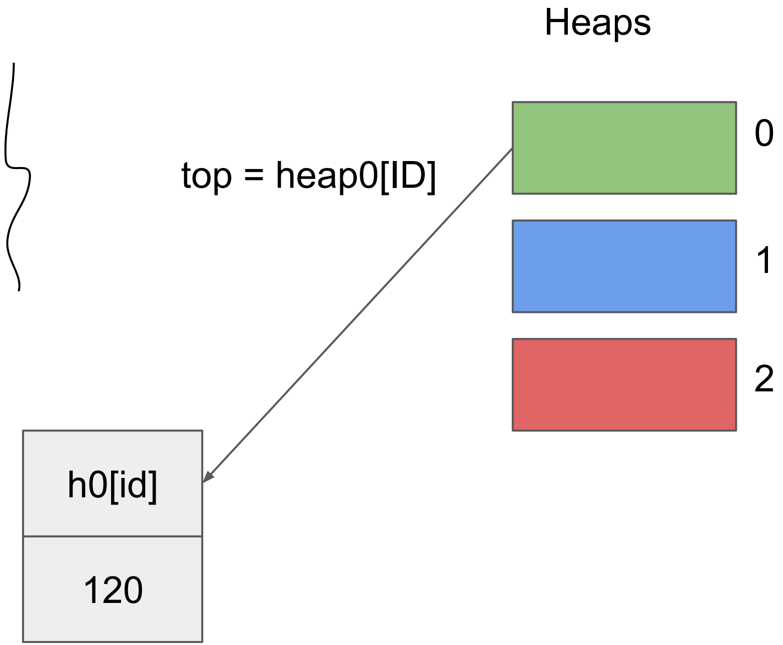
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



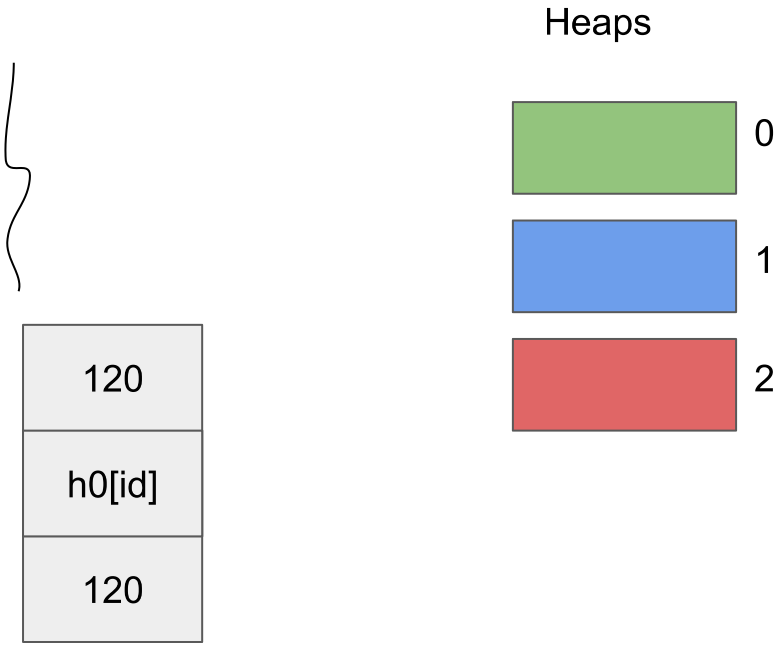
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



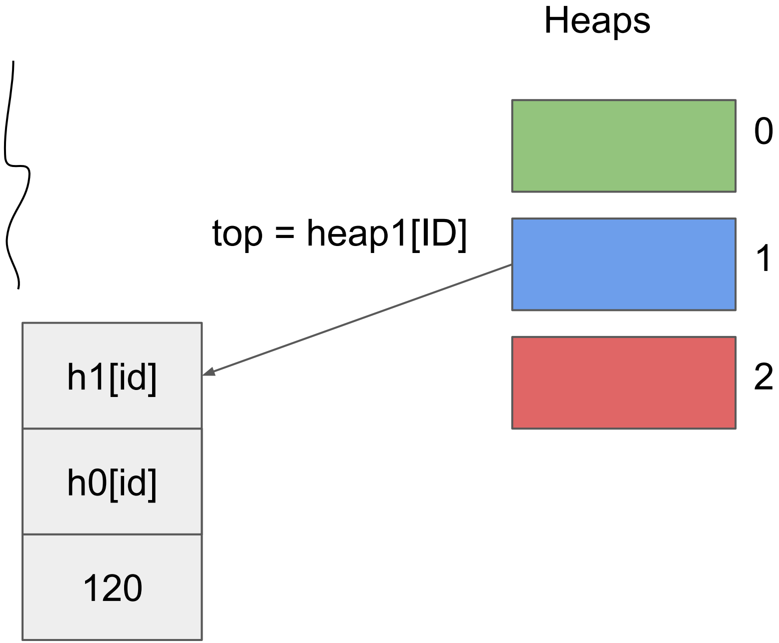
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



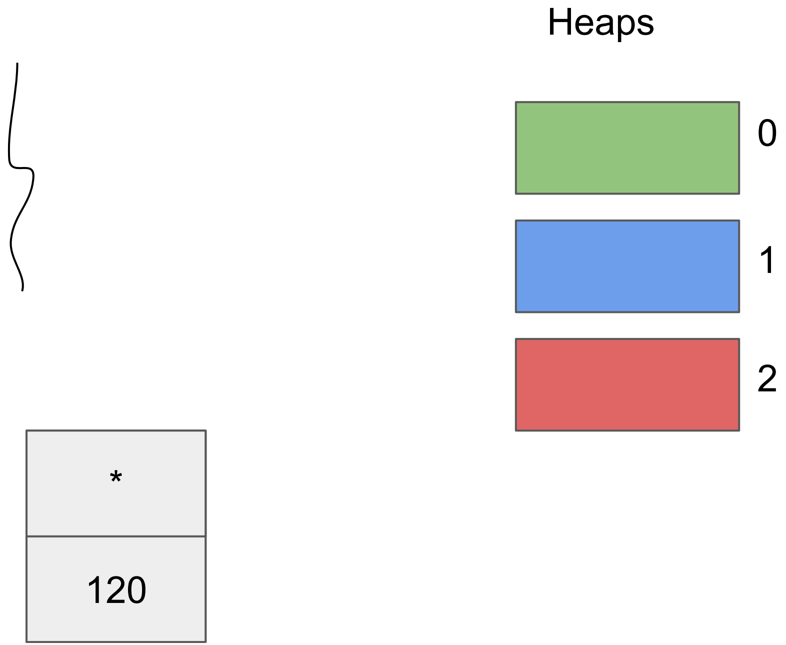
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



Heaps

0

1

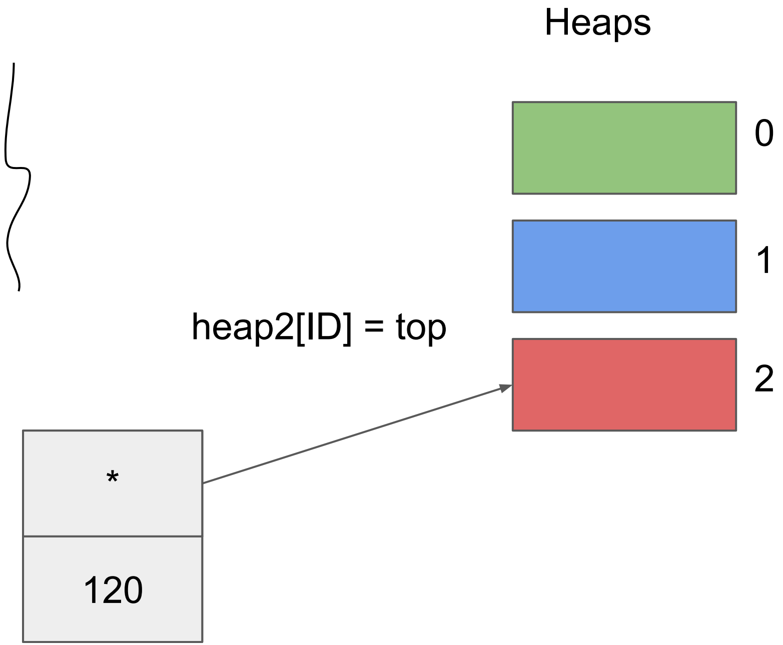
2

*

120

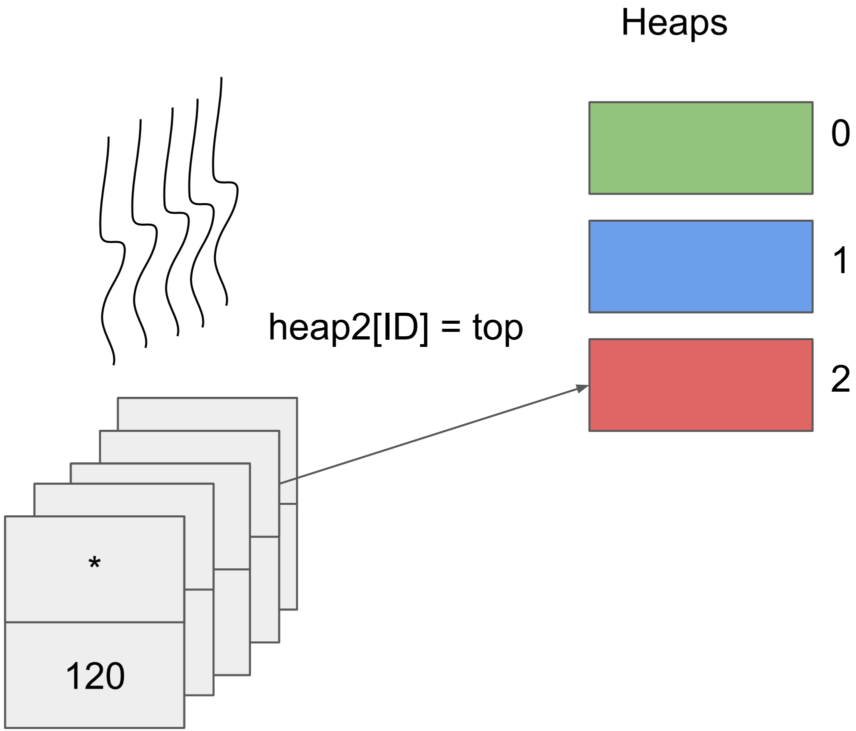
# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```

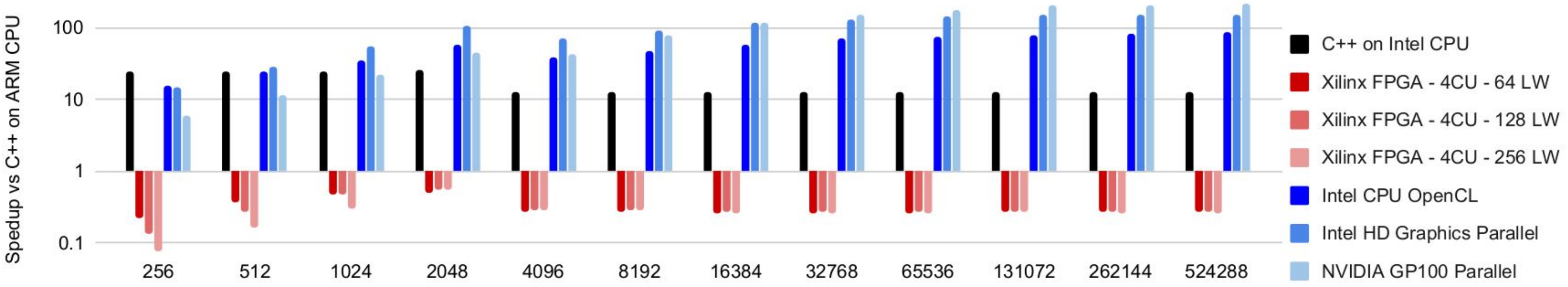


# OpenCL Bytecode Interpreter (II)

```
// Expressing vector addition
THREAD_ID,
DUP,
PARALLEL_GLOAD_INDEXED, 0,
THREAD_ID,
PARALLEL_GLOAD_INDEXED, 1,
IMUL,
PARALLEL_GSTORE_INDEXED, 2,
HALT
```



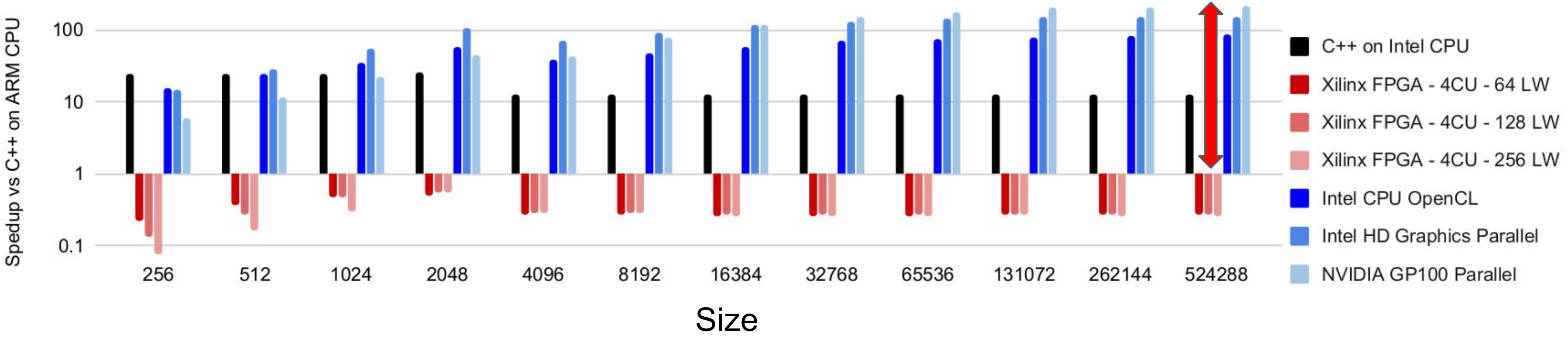
# Initial Results



**The higher, the better**

# Initial Results

Compared to ARM 1GHz: ~151x on Intel HD Graphics ~214x on NVIDIA



GPUs:

Compared to Intel i7:

- 11x on Intel HD Graphics
- 17x on NVIDIA GP100

FPGAs:

Slowdown on FPGAs:

- Not enough space
- Hard to tune

# How this can be useful?

1) Main CPUs with “space” for hardware specialization
   a) FPGAs inside the CPU
   b) Space for custom instructions (e.g., ARM Custom Instructions - ARM Cortex M33 - )

2) If workloads follow SIMT patterns and/or pipeline computation → Execution on parallel bytecode interpreters can be feasible

3) Existing VMs
   a) E.g., TornadoVM
      i) When the VM is compiling the code (JIT code to FPGA) → Use the parallel bytecode interpreter
      ii) Multiple heaps on heterogeneous hardware can be extremely useful (local, constant, etc).

# Takeaways

- Heterogeneous hardware is here to stay
- Managed runtime systems could potentially be accelerated using heterogeneous hardware
  - Garbage Collection [ISMM’12], [CASES’16]
  - Bytecode interpreters ← This work
  - Other components?
- Promising speedups even for simple examples
  - Parallel Bytecode interpreter
  - Multiple memory regions (full tier memory on the target device)

# Future Work

- Include more benchmarks
- Include more analysis for each memory region
- Comparisons against mainstream programming languages
- Use other standards such as SYCL C++ from Khronos Group
    - Intel DPC++
    - Codeplay ComputeCPP

# Thank you so much for your attention

This work is partially supported by the EU Horizon 2020 E2Data 780245.

Contact: Juan Fumero <juanfumero@acm.org>

Website: https://jjfumero.github.io