13th International Workshop on OpenCL and SYCL

IWOCL 2025

Comparative Analysis of Implementation Techniques for Sub-groups on CPUs

Moritz Heckmann, Saarland University

Joachim Meyer, Sebastian Hack (Saarland University)

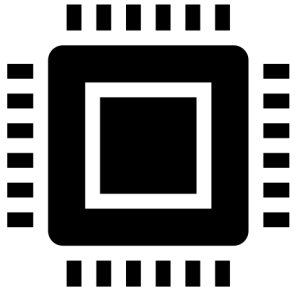
April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

Introduction

SYCL

- Khronos Industry Standard
- Enables heterogenous device programming using modern C++
- AdaptiveCpp open-source implementation



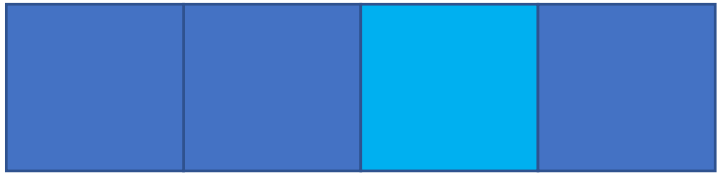
Parallel_for Nd_range

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```

Work-item
(Instantiation)

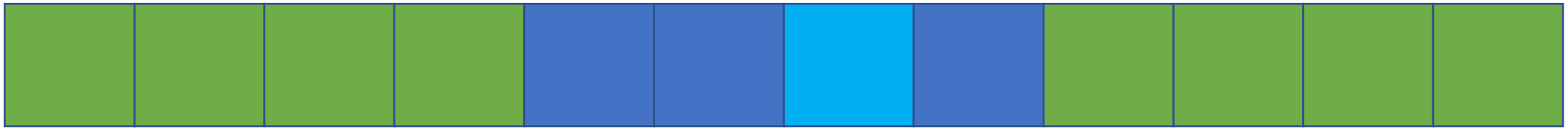
```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```



Sub-group

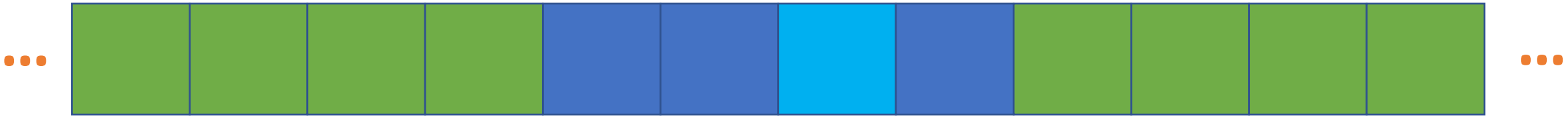
- The size of sub-groups is implementation defined

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```



Work-group

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```



Nd-range

Work/sub-group Barriers

- If a work-item reached a work/sub-group barrier, then it must wait until all other work-items, in its work/sub-group, have reached the barrier

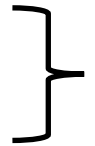
```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
      auto id = item.get_local_linear_id();
      auto size = item.get_local_range(0);
      scratch[id] = arr[id];
      work/sub_group_barrier();
      arr[id] = scratch[(id + 1) % size];
  }
);
```

How AdaptiveCpp Maps Kernels onto CPUs

- Uses continuation-based synchronization (CBS) to implicitly synchronize between work-items in a work-group
- Proposed by Karrenberg and Hack

Barrier-free Kernel

```
[=](sycl::nd_item<1> item) noexcept {
    auto id = item.get_local_linear_id();
    arr[id]++;
}
```



Barrier free region

Work-item loop


```
for (sycl::nd_item<1> item : work_group) {
  auto id = item.get_local_linear_id();
  arr[id]++;
}
```

Kernel With Barrier

```
[=](sycl::nd_item<1> item) noexcept {
    auto id = item.get_local_linear_id();
    auto size = item.get_local_range(0);
    scratch[id] = arr[id];
    work_group_barrier();
    arr[id] = scratch[(id + 1) % size];
}
```

Barrier free region

Barrier free region

```
for (sycl::nd_item<1> item : work_group) {
  auto id = item.get_local_linear_id();
  auto size = item.get_local_range(0);
  scratch[id] = arr[id];
}
// work_group_barrier();
for (sycl::nd_item<1> item : work_group) {
  arr[id] = scratch[(id + 1) % size];
}
```

Sub-group Implementation

- The size of sub-groups is implementation defined
- In AdaptiveCpp's CPU implementation sub-groups have a size of one

=> synchronization between work-items in a sub-group is a no-op

Problems With Sub-group Implementation

- Kernels written for different sub-group sizes are not portable
  - Nvidia: sub-groups of size 32
  - Amd: sub-groups of size 32/64
- Sub-groups of size one are useless
- It is difficult to write algorithms that support sub-groups of size one

Many Kernels Use Sub-groups

- GPUs have hardware-level support for sub-groups
- To achieve peak performance on GPUs we need to use sub-groups

How to Map Sub-groups onto CPUs

Hierarchical CBS

1. Work-group: CBS ignores sub-group barriers
2. Sub-group: CBS considers only sub-group barriers on regions from 1)

Split at Work-group Barriers

```
[=](sycl::nd_item<1> item) noexcept {
    // 1)
    sub_group_barrier();
    // 2)
    work_group_barrier();
    // 3)
}
```

Work-group barrier free region

Work-group barrier free region

Split at Sub-group Barriers

```
for (auto sub_group : work_group) {
  // 1)                          }  Barrier free region
  sub_group_barrier();
  // 2)                          }  Barrier free region
}
// work_group_barrier();
for (auto sub_group : work_group) {
  // 3)                          }  Barrier free region
}
```

```
for (auto sub_group : work_group) {
  for (auto work_item : sub_group) {
    // 1)
  }
  // sub_group_barrier();
  for (auto work_item : sub_group) {
    // 2)
  }
}
// work_group_barrier();
for (auto sub_group : work_group) {
  for (auto work_item : sub_group) {
    // 3)
  }
}
```

Whole-function Vectorization Approach

1. Remove sub-group barriers
2. Work-group: Apply CBS
3. Sub-group: Vectorize the work-item loops using whole-function (WFV) vectorization

Whole-function Vectorization

- Whole-function vectorization guarantees that every instruction in a function is executed for every vector lane in lockstep
- Whole-function vectorization is a generalization of outer-loop vectorization
- Proposed by Karrenberg and Hack
- Region Vectorizer (RV) as whole-function vectorizer

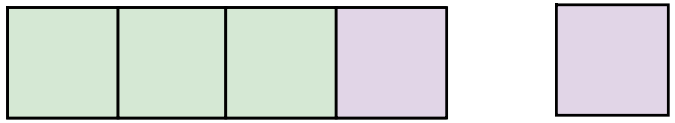
Mapping to Sub-groups

- Map sub-groups to vectors and work-items to vector lanes

$\Rightarrow$ Every instruction gets executed for every work-item in a sub-group in lockstep

$\Rightarrow$ Every work-item in a sub-group executes all instructions in front of a sub-group barrier before any work-item executes instructions after the sub-group barrier

vector **vector-lane**
1x4 **1x1**

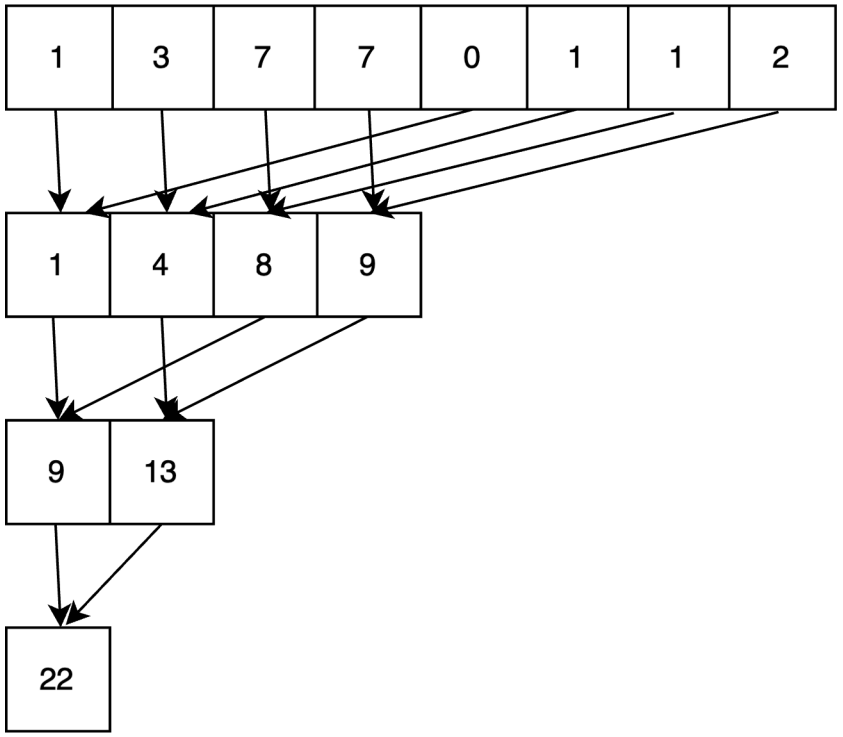


How to Implement SYCL Algorithms

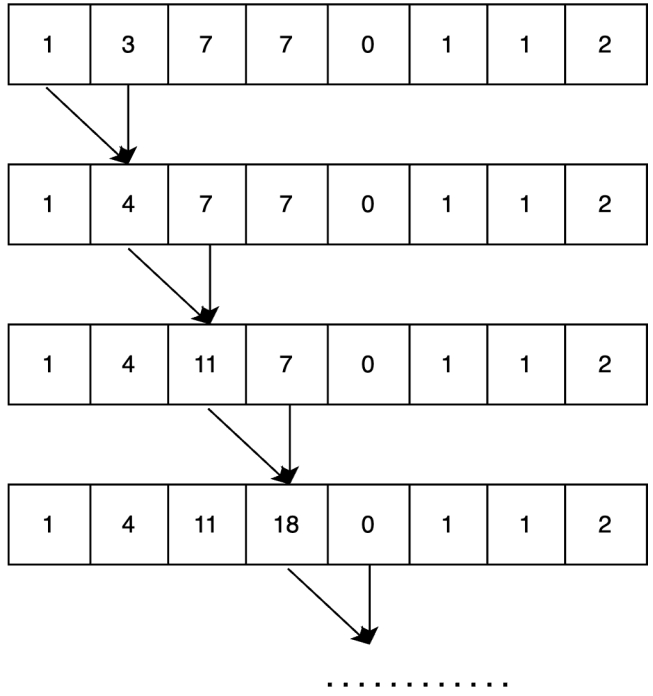
Reduction

- Combines inputs from work-items into one output using a binary operation

WFV



H-CBS



Optimizations

Variant / Optimization	H-CBS	WFV
Vector Reduce Intrinsic	1.75	~1

Geomean speedup over implementation without the optimization

Evaluation

Machines

- **AMD Epyc 7702** Zen2**,** AVX2
  - 2x64-core 2.00-3.35GHz
- **AMD Ryzen 9 7950X3D** Zen4**,** AVX-512
  - 16-core 4.2GHz
- **Intel Xeon E5-2698 v4** Broadwell*,* AVX2
  - 2x20-core 2.2-3.6GHz

Best Sub-group Size

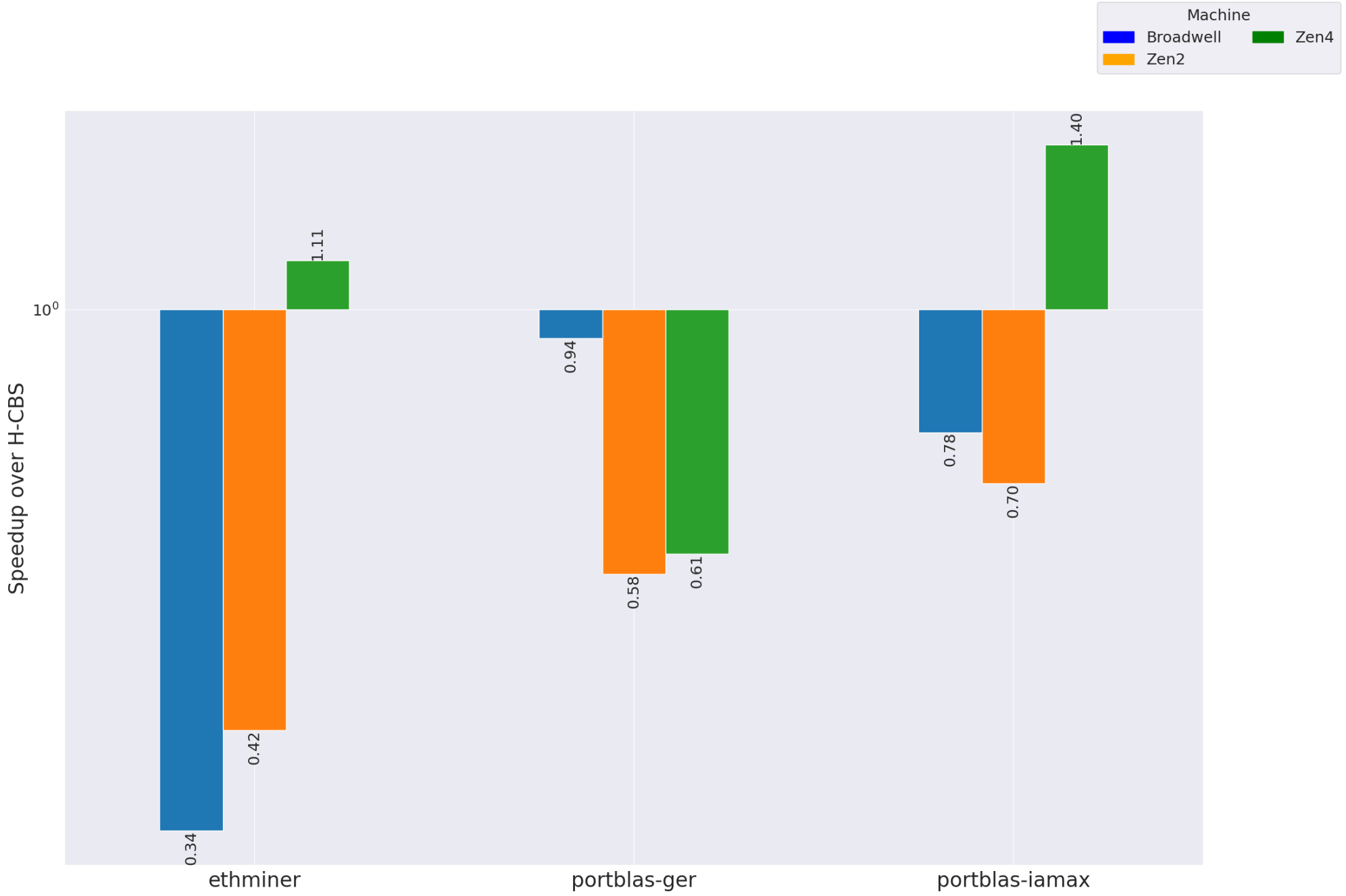
Sg-Size/ Variant	8	16	64
H-CBS	0.82	0.96	0.71
WFV	0.86	1.03	0.84

Geomean speedup over sub-group size 32
using SSCP

WFV Vs. H-CBS

Geomean Speedup
1.7

Geomean speedup of WFV over H-CBS
using SSCP and sub-groups of size 32

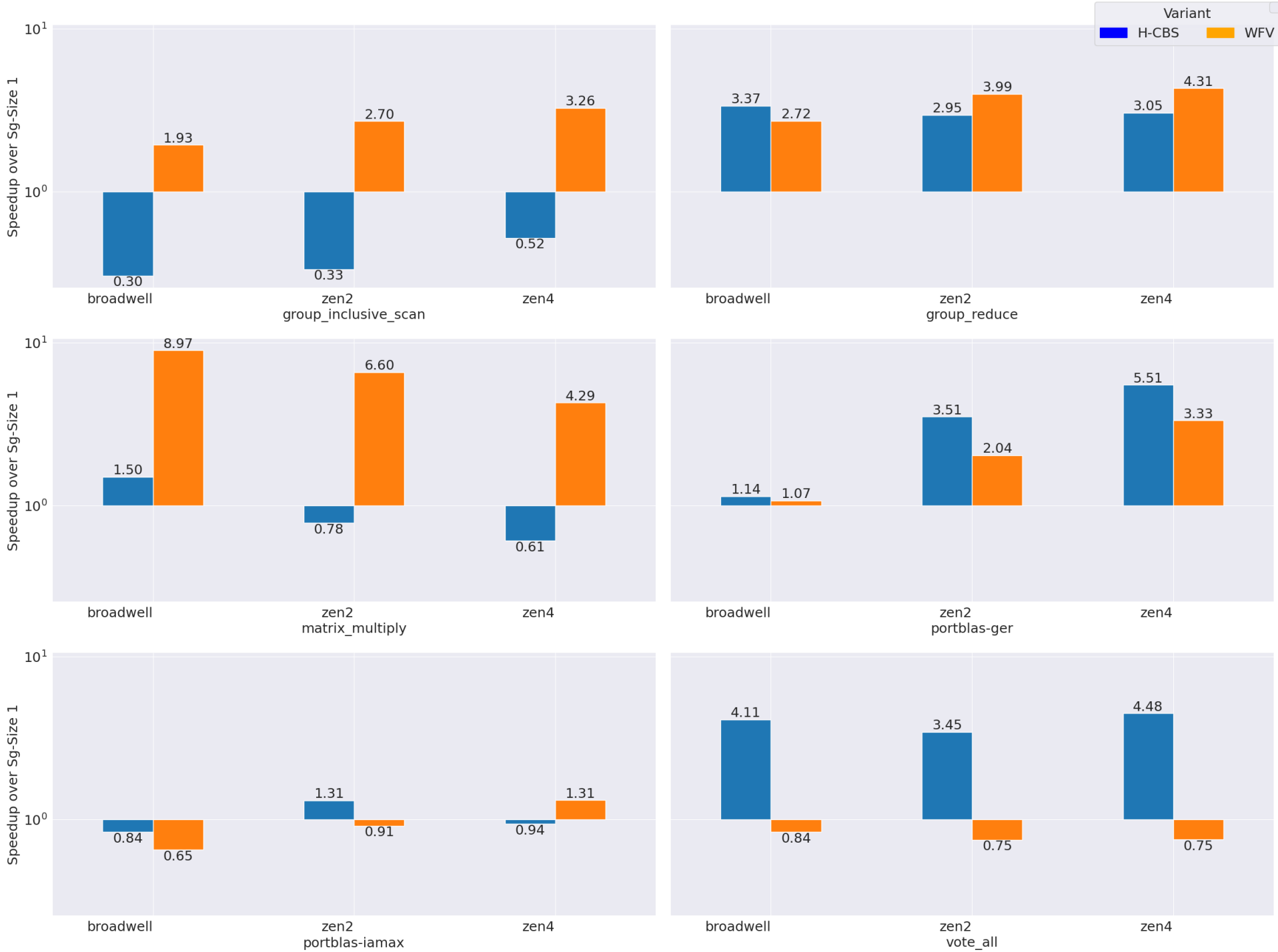

Machine
Broadwell
Zen4
Zen2
Speedup over H-CBS
10^0
0.34
0.42
1.11
0.94
0.58
0.61
0.78
0.70
1.40
ethminer
portblas-ger
portblas-iamax

```
for (i = 0; i < N; ++ i) {
  for (j = 0; j < N; ++j) {
      C[i+N*j] = 0
      for (k = 0; k < N; ++k) {
          C[i+N*j] += A[i+N*k] * B[k+N*j]
      }
  }
}
```

WFV and H-CBS Vs. Sg Size One

Variant	Geomean Speedup
WFV	2.08
H-CBS	1.51

Geomean speedup over sub-group size of
one implementation using SSCP


Variant
H-CBS
WFV
Speedup over Sg-Size 1
10^1
10^0
0.30
1.93
0.33
2.70
0.52
3.26
broadwell
zen2
zen4
group_inclusive_scan
3.37
2.72
2.95
3.99
3.05
4.31
group_reduce
1.50
8.97
0.78
6.60
0.61
4.29
matrix_multiply
1.14
1.07
3.51
2.04
5.51
3.33
portblas-ger
0.84
0.65
1.31
0.91
0.94
1.31
portblas-iamax
4.11
0.84
3.45
0.75
4.48
0.75
vote_all

Conclusion

- Both H-CBS and WFV outperform the sub-group size of one implementation
- WFV is faster than H-CBS