Performance Analysis of Matrix-free Conjugate Gradient Kernels Using SYCL

Igor Baratta, Chris Richardson and Garth N. Wells
Department of Engineering

UNIVERSITY OF CAMBRIDGE

Outline

Introduction

Conjugate Gradient Kernels

Performance Model

Experimental Results

Conclusion

Compute variety at scale

TOP 500
The List.

Supercomputer Fugaku

A64FX 48C 2.2GHz

Rmax (TFlop/s)
442,010.0

Summit

IBM POWER9 22C 3.07GHz
NVIDIA Volta GV100

Rmax (TFlop/s)
148,600.0

Sierra

IBM POWER9 22C 3.1GHz
NVIDIA Volta GV100

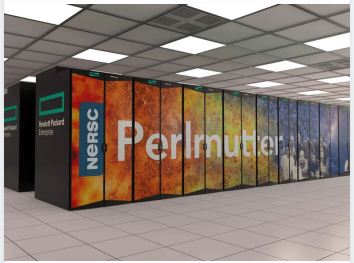
Rmax (TFlop/s)
94,640.0

Sunway TaihuLight

Sunway SW26010 260C
1.45GHz, Sunway

Rmax (TFlop/s)
93,014.6

Perlmutter



AMD EPYC 7763 64C
2.45GHz
NVIDIA A100 SXM4 40 GB

Rmax (TFlop/s)
70,870.0

Programming model

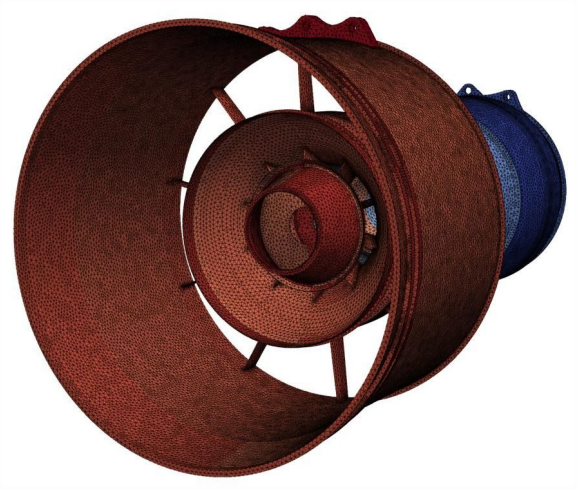
SYCL is a high-level single source parallel programming model, that can target a range of heterogeneous platforms:

- uses completely standard C++;
- both host CPU and device code can be written in the same C++ source file;
- open standard coordinated by the Khronos group.

SYCL implementations

- oneAPI Data Parallel C++ compiler - https://github.com/intel/llvm — tag - sycl-nightly/20220222
- hipSYCL - https://github.com/illuhad/hipSYCL — tag - v0.9.2

Finite element overview



$$A\mathbf{x} = \mathbf{b}$$

$$A_e = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$A = \begin{bmatrix} & \\ & \end{bmatrix}$$

FEniCS project

https://github.com/FEniCS/dolfinx https://github.com/FEniCS/basix

https://github.com/FEniCS/ufl https://github.com/FEniCS/ffcx

Conjugate Gradient to solve Ax=b

$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
$i = 0$
while $i < i_{\max}$ and $\mathbf{r}_i \cdot \mathbf{r}_i > \epsilon$ **do**
  $\mathbf{y}_i = A\mathbf{p}_i$
  $\alpha_i = \dfrac{\mathbf{r}_i \cdot \mathbf{r}_i}{\mathbf{p}_i \cdot \mathbf{y}_i}$
  $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha\mathbf{p}_i$
  $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha\mathbf{y}_i$
  $\beta_i = \dfrac{\mathbf{r}_{i+1} \cdot \mathbf{r}_{i+1}}{\mathbf{r}_i \cdot \mathbf{r}_i}$
  $\mathbf{p}_{i+1} = \beta_i\mathbf{p}_i + \mathbf{r}_i$
  $i = i + 1$
end while

$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}$$

$$\alpha = \mathbf{x} \cdot \mathbf{y}$$

$$\begin{cases} \mathbf{x} = \alpha\mathbf{p} + \mathbf{x} \\ \mathbf{r} = -\alpha\mathbf{y} + \mathbf{r} \\ \gamma = \mathbf{r} \cdot \mathbf{r} \end{cases}$$

Vector update using an ND-Range kernel

```
template <typename T>
sycl::event axpby(sycl::queue& queue,
                  std::size_t n, T alpha, T beta,
                  const T* x, T* y, std::size_t wgs,
                  const std::vector<sycl::event>& events = {})
{
 auto event = queue.submit([&](sycl::handler& h) {
   h.depends_on(events);
   h.parallel_for(sycl::range<1>{n, wgs}, [=](sycl::nd_item<1> it) {
     std::size_t i = it.get_global_id(0);
     if (i < n)
       y[i] = alpha * x[i] + beta * y[i];
   });
 });
 return event;
}
```

$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}$$

$2n$ reads + n writes
$2n$ multiplications + n additions

$$AI = \frac{3n}{24n} = \frac{1}{8}$$

Inner product using SYCL 2020 built-in reduction

```
template <typename T>
T dot(sycl::queue& queue,
      std::size_t n, const T* x, const T* y,
      std::size_t wgs, std::size_t bs,
      const std::vector<sycl::event>& events = {})
{
 // Get execution range
 sycl::nd_range<1> range{n / bs, wgs};

 sycl::buffer<T> sum{1};
 auto init = sycl::property::reduction::initialize_to_identity{};
 queue.submit([&](sycl::handler& h)
 {
   h.depends_on(events);
   auto reductor = sycl::reduction(sum, h, T{0.0}, std::plus<T>(),init);
   h.parallel_for(range, reductor, [=]( sycl::nd_item<1> it, auto& sum) {
     std::size_t idx = it.get_global_id(0);
     std::size_t size = it.get_global_range(0);
     for (std::size_t i = idx; i < n; i += size)
       sum += x[i] * y[i];
   });
 });

 sycl::host_accessor sum_host{sum};
 return sum_host[0];
}
```

$$\alpha = \mathbf{x} \cdot \mathbf{y} = \sum_{i}^{n} x_i y_i$$

$2n$ reads
n multiplications + n additions

$$AI = \frac{2n}{16n} = \frac{1}{8}$$

Inner product using group algorithms

```
queue.submit([&](sycl::handler &h) {
  h.depends_on(events);
  h.parallel_for(range, [=](sycl::nd_item<1> it) {
    std::size_t idx = it.get_global_id(0);
    std::size_t size = it.get_global_range(0);
    sycl::group group = it.get_group();

    // Compute local sum
    T t_sum = 0;
    for (std::size_t i = idx; i < n; i += size)
      t_sum += x[i] * y[i];

    // Compute group reduction
    T g_sum = sycl::reduce_over_group(group, t_sum, std::plus<T>());

    // Group Leader updates global
    if (group.leader()) {
      sycl::atomic_ref<T, sycl::memory_order::relaxed,
sycl::memory_scope::device,
                       sycl::access::address_space::global_space>
        atomic_sum(*d_sum);
      atomic_sum.fetch_add(g_sum);
    }

  }); // End of the kernel function
});   // End of command group
```

$$\alpha = \mathbf{x} \cdot \mathbf{y} = \sum_{i}^{n} x_i y_i$$

$2n$ reads
n multiplications + n additions

$$AI = \frac{2n}{16n} = \frac{1}{8}$$

Fused update using SYCL 2020 built-in reduction

```
sycl::nd_range<1> range{n / bs, wgs};
auto init = sycl::property::reduction::initialize_to_identity{};
queue.submit([&](sycl::handler& h) {
  h.depends_on(events);
  auto reductor = sycl::reduction(sum, h, T{0.0}, std::plus<T>(), init);
  h.parallel_for(range, reductor, [=]( sycl::nd_item<1> it, auto& sum) {
    std::size_t idx = it.get_global_id(0);
    std::size_t size = it.get_global_range(0);
    for (std::size_t i = idx; i < n; i += size) {
      r[i] = -alpha * y[i] + r[i];
      x[i] = alpha * p[i] + x[i];
      sum += r[i] * r[i];
    }
  });
});
```

$$\begin{cases} \mathbf{x} = \alpha\mathbf{p} + \mathbf{x} \\ \mathbf{r} = -\alpha\mathbf{y} + \mathbf{r} \\ \gamma = \mathbf{r} \cdot \mathbf{r} \end{cases}$$

$4n$ reads + $2n$ writes
$3n$ multiplications + $3n$ additions

$$AI = \frac{3n}{24n} = \frac{1}{8}$$

Without fusion $5n$ reads.

Performance Model

$$T = T_0 + \frac{N_B}{W_a}$$

T is the execution time

N_B is the number of bytes accessed (loads and stores)

T_0 is the latency

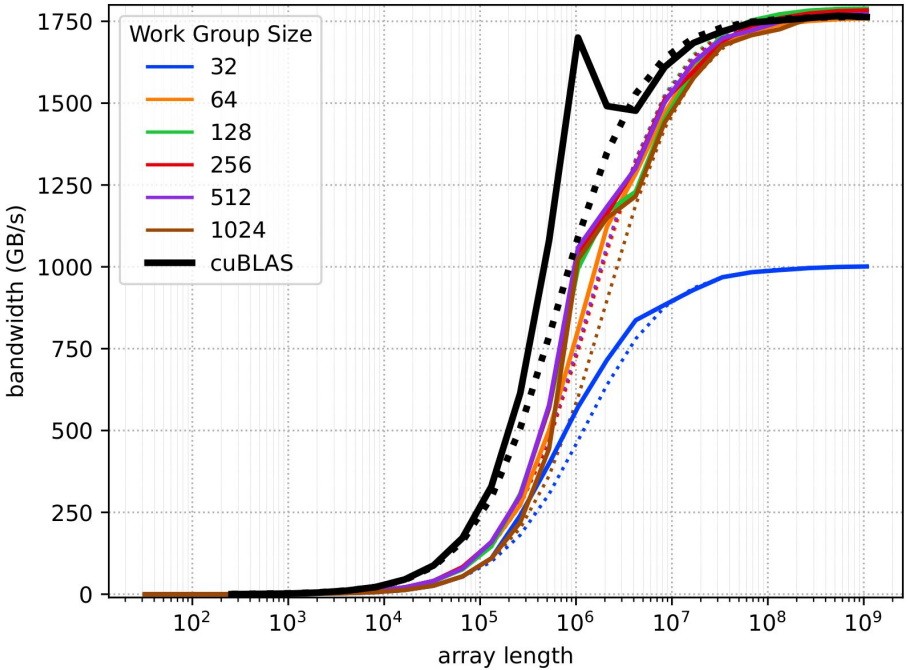
W_a is the asymptotic memory bandwidth

W is the observed memory bandwidth

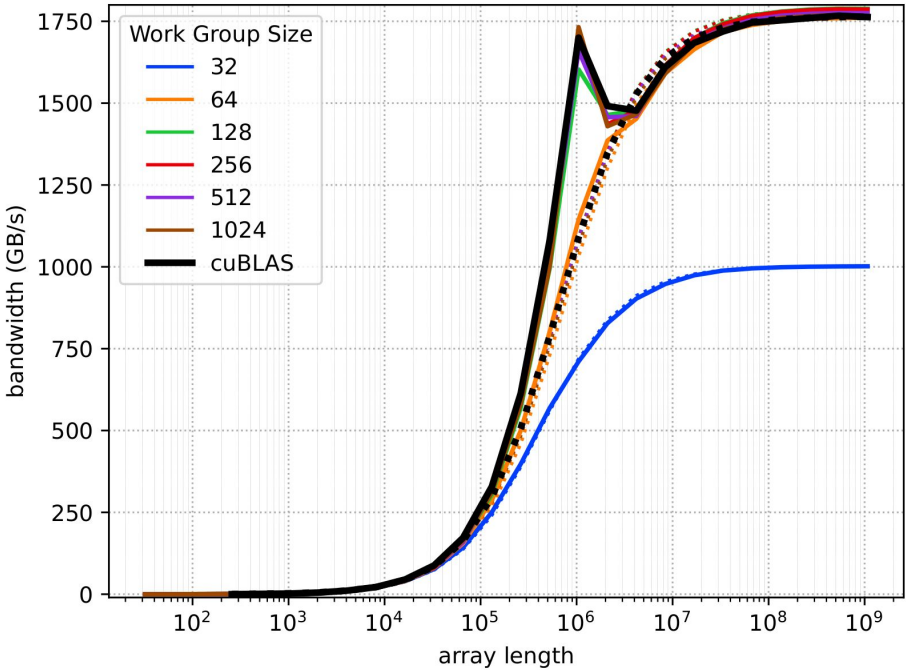
$$W = \frac{N_B}{T}$$

Vector update on the A100 GPU

hipSYCL



Intel SYCL



$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}$$

Fitted parameters for the vector update kernel (axpby) on the A100 GPU

	hipSYCL		Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})	T_0 (µs)	W_a (GB s^{-1})
32	28	1002	10	1002
64	20	1761	10	1762
128	20	1791	9	1791
256	20	1785	9	1788
512	19	1773	9	1776
1024	28	1768	9	1772

For cuBLAS T_0 = 8 µs and W_a =1711 GB s $^{-1}$

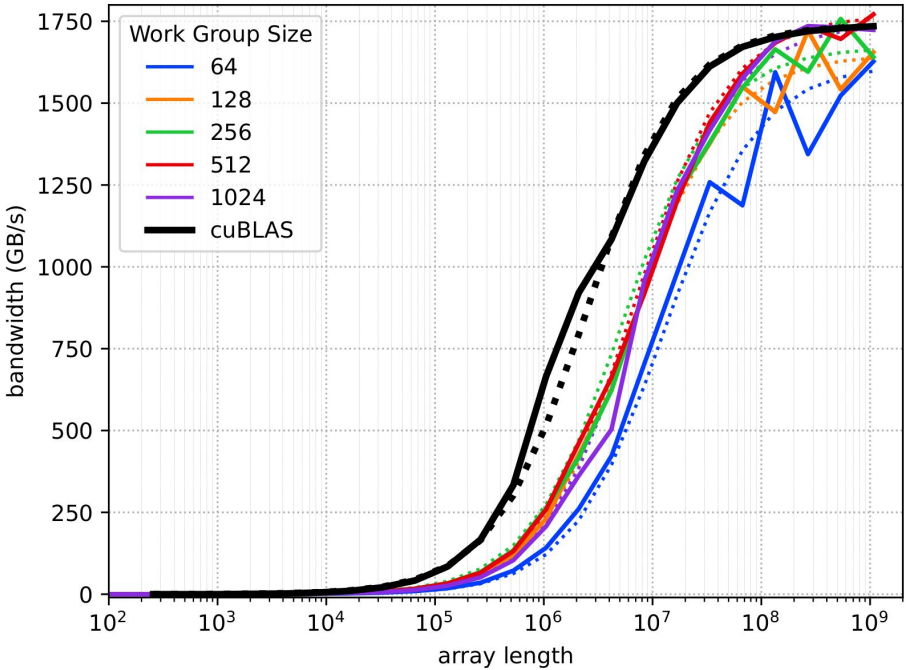
Fitted parameters for the vector update kernel (axpby) on the Ice Lake node

	hipSYCL		Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})	T_0 (µs)	W_a (GB s^{-1})
16	186	286	868	298
32	263	284	827	300
64	103	286	745	296
128	156	290	726	298

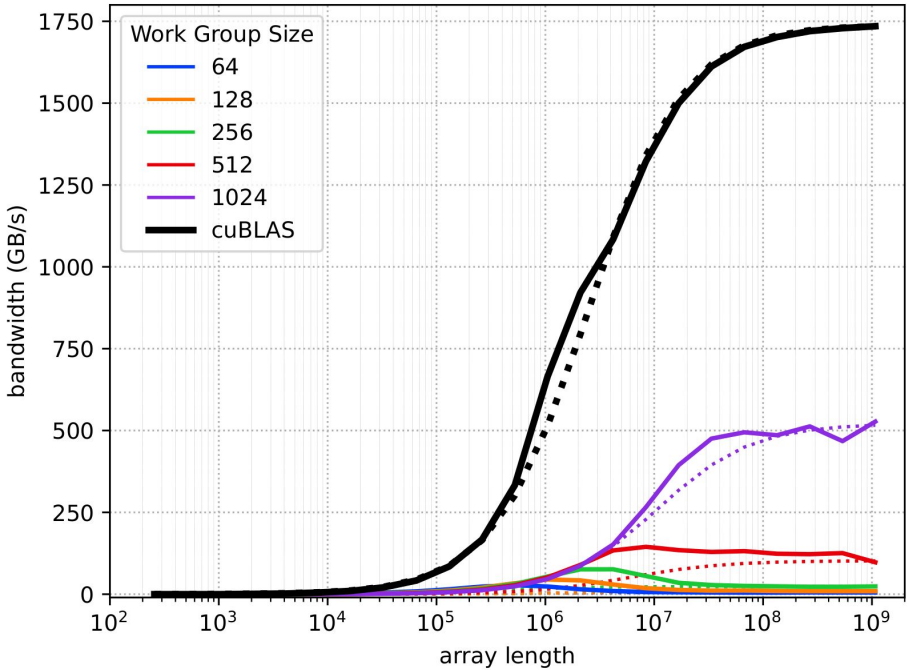
For STREAM Triad W_a = 271 GB s $^{-1}$

Inner product on the A100 GPU

hipSYCL



Intel SYCL



$$\alpha = \mathbf{x} \cdot \mathbf{y}$$

Fitted parameters for the inner product kernel (dot) on the A100

bs=16

	hipSYCL		Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})	T_0 (µs)	W_a (GB s^{-1})
64	128	1617	1582	5
128	62	1647	3563	9
256	51	1671	231	24
512	61	1769	924	102
1024	68	1742	330	502

For cuBLAS T_0 = 23 µs and W_a =1739 GB s^{-1}

Fitted parameters for the inner product kernel (dot) on the A100

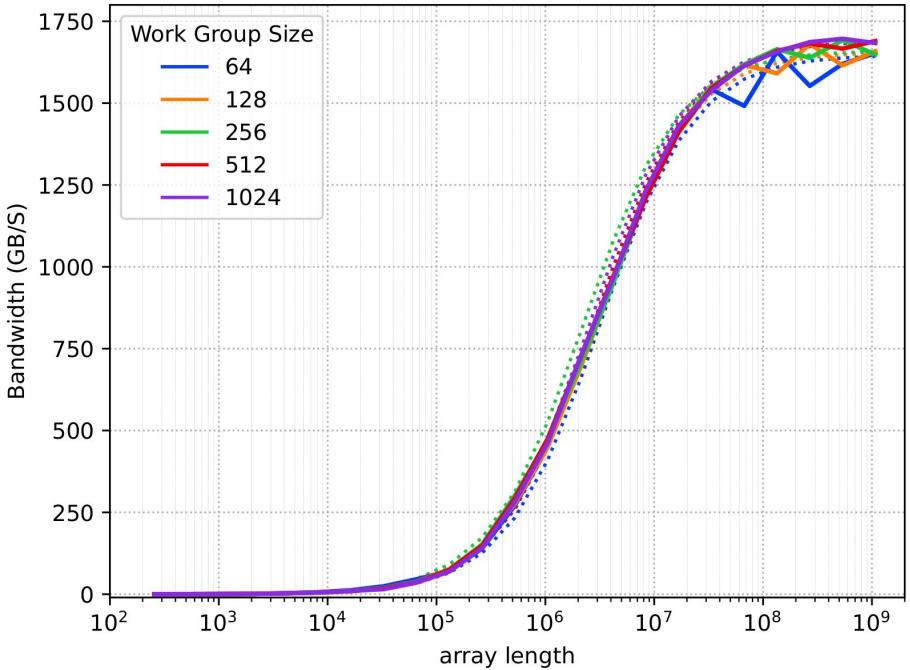
bs=256

	Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})
64	27	197
128	41	405
256	62	983
512	108	1746
1024	105	1746

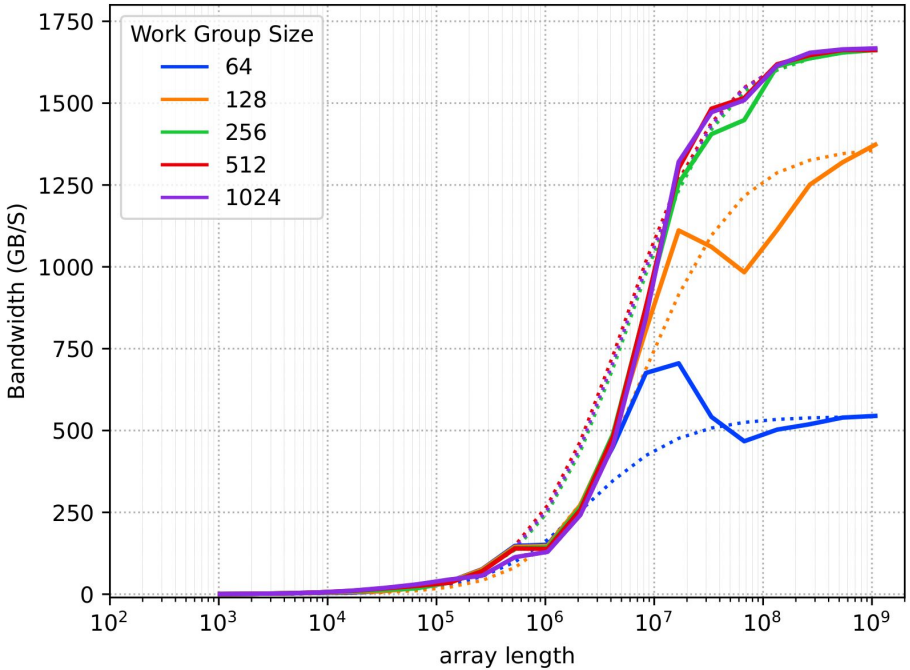
For cuBLAS T_0 = 23 µs and W_a =1739 GB s $^{-1}$

Fused update on the A100 GPU

hipSYCL



Intel SYCL

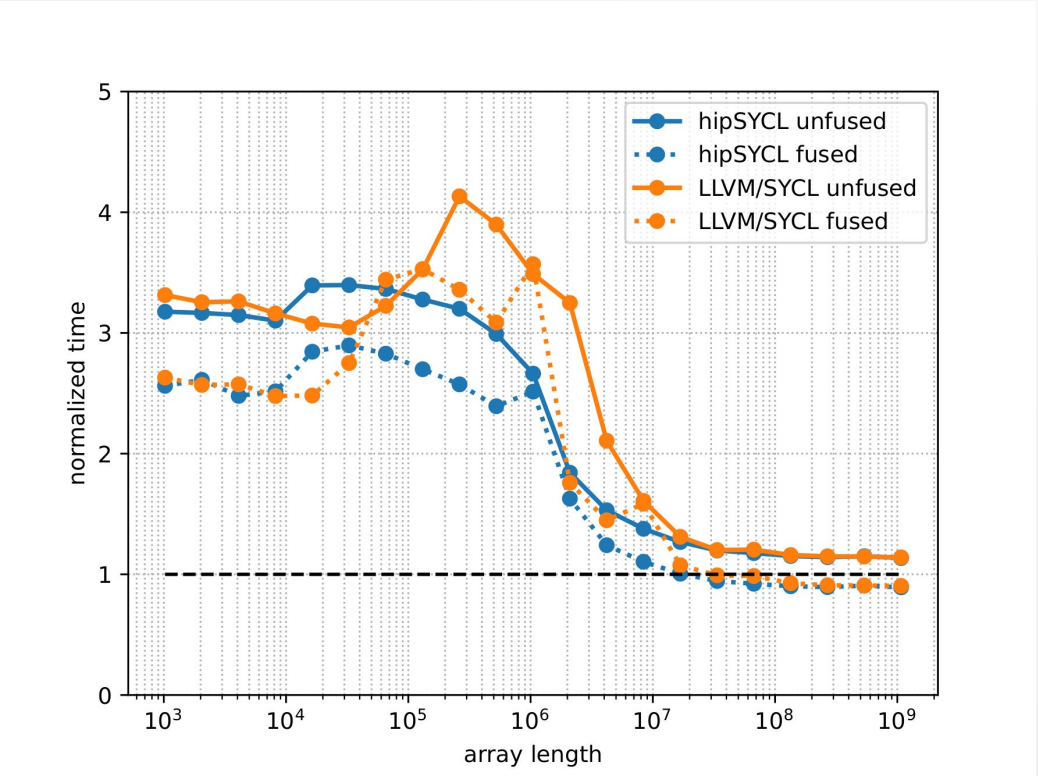


$$\begin{cases} \mathbf{x} = \alpha\mathbf{p} + \mathbf{x} \\ \mathbf{r} = -\alpha\mathbf{y} + \mathbf{r} \\ \gamma = \mathbf{r} \cdot \mathbf{r} \end{cases}$$

Fitted parameters for the fused update kernel on the A100

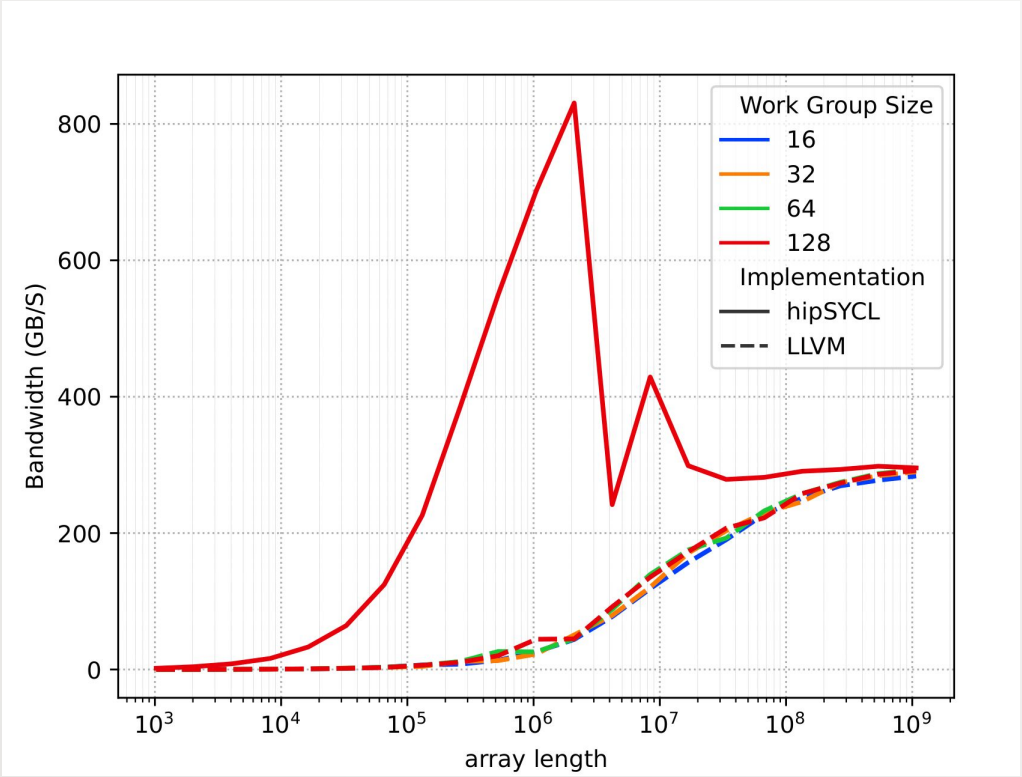
	hipSYCL		Intel LLVM/SYCL	
WG Size	T_0 (s)	W_a (GB s^{-1})	T_0 (s)	W_a (GB s^{-1})
64	93	1648	209	543
128	82	1657	291	1367
256	65	1661	171	1672
512	82	1691	155	1672
1024	76	1692	164	1678

Fused update on the A100 GPU



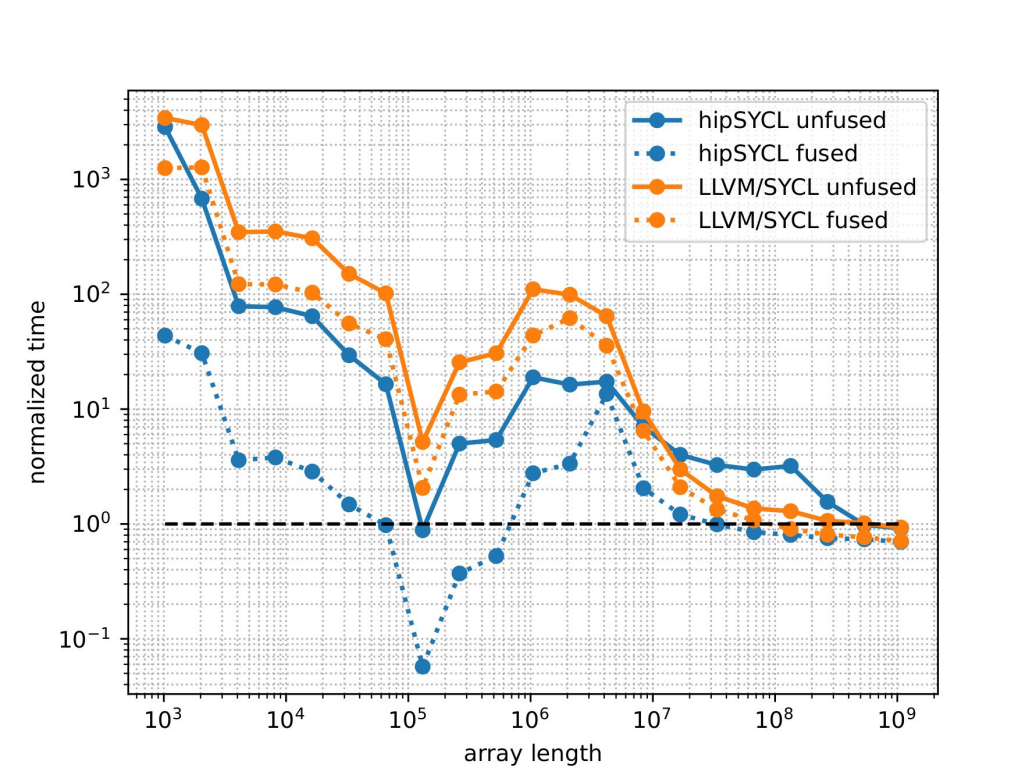
Normalised speed of the fused update kernel on the A100 GPU.
A normalised time of one corresponds to three operations (unfused) using cuBLAS.

Fused update on the Ice Lake CPU node



WG Size	Intel LLVM/SYCL	
	T_0 (µs)	W_a (GB s^{-1})
16	1675	285
32	1661	291
64	1554	294
128	1505	293

Fused update on the Ice Lake node



Normalised speed of the fused update kernel on the Ice Lake node.
A normalised time of one corresponds to three operations (unfused) using MKL.

Final observations

- Results indicate that for large problem sizes the performance of the SYCL kernels and the vendor libraries is very close.
- For smaller array sizes different SYCL implementations have considerably different performance characteristics.
- The performance model indicates that the two SYCL implementations do reach the asymptotic memory bandwidth of the vendor implementations, but that latency for the SYCL implementations is sometimes very high and that latency can dominate the measured performance.
- High latency can also affect strong scaling performance.
- Our examination focused only on matrix-free conjugate gradient kernels, and did not consider preconditioning or evaluation of the matrix action, which is also important for overall application performance.