Performance analysis of matrix-free conjugate gradient kernels using SYCL

Igor A. Baratta
Chris N. Richardson
Garth N. Wells
ia397@cam.ac.uk
cnr12@cam.ac.uk
gnw20@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Abstract

We examine the performance of matrix-free SYCL implementations of the conjugate gradient method for solving sparse linear systems of equations. Performance is tested on an NVIDIA A100-80GB device and a dual socket Intel Ice Lake CPU node using different SYCL implementations, and compared to CUDA BLAS (cuBLAS) implementations on the A100 GPU and MKL implementations on the CPU node. All considered kernels in the matrix-free implementation are memory bandwidth limited, and a simple performance model is applied to estimate the asymptotic memory bandwidth and the latency. Our experiments show that in most cases the considered SYCL implementations match the asymptotic performance of the reference implementations. However, for smaller but practically relevant problem sizes latency is observed to have a significant impact on performance. For some cases the SYCL latency is reasonably close to the reference (cuBLAS/MKL) implementation latency, but in other cases it is more than one order of magnitude greater. In particular, SYCL built-in reductions on the GPU and all operations for one of the SYCL implementations on the CPU exhibit high latency, and this latency limits performance at problem sizes that can in cases be representative of full application simulations, and can degrade strong scaling performance.

Keywords: Conjugate gradient method, matrix-free, performance modelling, memory bandwidth, latency, SYCL.

ACM Reference Format:
Igor A. Baratta, Chris N. Richardson, and Garth N. Wells. 2022. Performance analysis of matrix-free conjugate gradient kernels using SYCL. In *International Workshop on OpenCL (IWOCL'22), May 10–12, 2022, Bristol, United Kingdom, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3529538.3529993

IWOCL'22, May 10–12, 2022, Bristol, United Kingdom, United Kingdom

ACM ISBN 978-1-4503-9658-5/22/05.
https://doi.org/10.1145/3529538.3529993

1 Introduction

The solution of sparse linear systems underpins many numerical methods in science and engineering. Finite element and finite difference methods generate sparse linear systems, with sparse matrices usually formed and stored, and then solved using a sparse solver. So-called 'direct' sparse solvers, e.g. LU factorisation, can be used, but typically have $O(n^2)$ cost complexity for three-dimensional problems, where n is the number of unknowns. Moreover, the fundamental structure of direct solver algorithms makes efficient parallelisation difficult. Alternatively, iterative solvers can yield performance better than $O(n^2)$, and in cases can solve problems with $O(n)$ cost. Well known iterative methods, such as the conjugate gradient method, lend themselves naturally to parallel implementations and core operations can be represented straightforwardly in terms of distinct parallel computational kernels.

Many well established iterative methods for linear systems from the Krylov family of methods, which includes the conjugate gradient method, can be implemented 'matrix-free', avoiding the need to form and store a sparse matrix. Rather than forming sparse matrices, the solvers work with the *action* of the matrix operator on a vector. The action can often be evaluated by simulation codes directly, bypassing the creation of a sparse matrix. This can reduce the computational cost complexity, provide substantial memory savings, and avoids the performance bottleneck of forming a sparse matrix, which can involve irregular memory access, look-ups and substantial memory traffic. It also avoids the need for a sparse matrix vector multiplication kernel. For these reasons, matrix-free methods are attracting considerable attention for CPU and GPU architectures, and especially as suitable algorithms for future exascale systems.

In this work we assess the performance of two different SYCL implementations for the core operations (kernels) for a matrix-free conjugate gradient solver. The performance of the considered operations is limited by memory bandwidth.

We measure the effective memory bandwidth for a range of operations on different hardware, and use a simple performance model to estimate the latency. This analysis can inform code design and guides target problem sizes in order to hide latency effects.

A motivation for this study is the construction of full matrix-free finite element solvers using SYCL that can be executed on CPU and GPU architectures. A complete solver involves a range of operations. Understanding the overall performance characteristics, and using this to guide design and optimisation, requires analysis of each stage of the solution process. In this work we focus on the matrix-free linear solver stage, treating the computation of the discrete operator action on a vector as a 'black box'. We do not consider the performance of the action evaluation, which is application dependent.

2 Related work

The presented work is motivated by the Exascale Computing Project *Center for Efficient Exascale Discretizations* 'bake-off' problems for high-order finite element methods [8]. The bake-off problems involve the evaluation of a finite element operator for a range of simple equation types, and application of a Jacobi preconditioned or unpreconditioned conjugate-gradient solver. CPU and CUDA implementations of the bake-off problems are discussed in [8], and overall performance is compared. We wish to develop and analyse SYCL implementations of the bake-off problems, but to understand the factors that affect the headline performance we need to carefully examine the performance of each stage in the finite element solver. Here we start with the kernels for a matrix-free conjugate gradient solver.

The kernels examined in this work effectively measure memory bandwidth performance, kernel latency and the efficiency of reductions. Earlier work has examined memory bandwidth performance, including for SYCL. BabelStream [6] was used to study performance for a range of programming models across a range of architectures. BabelStream works with problem sizes that assess the asymptotic memory bandwidth. A range of simple test problems that measure memory bandwidth performance is presented in [9], but without performance data. Our analysis and results are related to the performance analysis in [4], where conjugate gradient operations were implemented with the OCCA library, which was used to create code for CUDA and HIP. Distinguishing features of our work are (i) the consideration of a range problem sizes that includes cases where latency can have a major effect on the observed performance, and (ii) assessment of SYCL 2020 implementations for conjugate gradient operations on CPUs and GPUs. With our focus on both asymptotic bandwidth and latency, we can identify important cases where latency can be the limiting performance factor for practical applications.

3 Matrix-free conjugate gradient solver

The conjugate gradient algorithm for solving $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ real, symmetric positive-definite matrix, is summarised in Algorithm 1. Per iteration, the conjugate gradient method involves one operator action evaluation ($A\mathbf{p}_i$ in Algorithm 1), three vector updates ('axpy' operations), one inner product and two norm computations [3].

Algorithm 1 Conjugate Gradient method for solving $A\mathbf{x} = \mathbf{b}$.

1: $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
2: $\mathbf{p}_0 = \mathbf{r}_0$
3: $i = 0$
4: **while** $i < i_{\max}$ and $\mathbf{r}_i \cdot \mathbf{r}_i > \epsilon$ **do**
5: $\quad \mathbf{y}_i = A\mathbf{p}_i$
6: $\quad \alpha_i = \dfrac{\mathbf{r}_i \cdot \mathbf{r}_i}{\mathbf{p}_i \cdot \mathbf{y}_i}$
7: $\quad \mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{p}_i$
8: $\quad \mathbf{r}_{i+1} = \mathbf{r}_i - \alpha \mathbf{y}_i$
9: $\quad \beta_i = \dfrac{\mathbf{r}_{i+1} \cdot \mathbf{r}_{i+1}}{\mathbf{r}_i \cdot \mathbf{r}_i}$
10: $\quad \mathbf{p}_{i+1} = \beta_i \mathbf{p}_i + \mathbf{r}_i$
11: $\quad i = i + 1$
12: **end while**

For a 'standard' conjugate gradient solver, the application code creates a sparse matrix A which is passed to the conjugate gradient solver, and at every iteration the matrix–vector product $A\mathbf{p}_i$ is computed. Sparse matrix–vector products have low arithmetic intensity, and in memory constrained environments the storage required for the sparse matrix can be a considerable fraction of the total memory footprint, especially for high-order methods. High-order methods are increasingly appealing as the arithmetic intensity of operations for building the sparse matrix or evaluating its action can be increased relative to low-order methods.

In a matrix-free computation, the application solver is called at each iteration to compute the vector $A\mathbf{p}_i$ (the 'action' of A on $\mathbf{p}_i$). A key to overall efficiency is the fast evaluation of the action of A by the application library. Fast evaluation of the action is highly dependent on specifics of the application, and is outside of the scope of this investigation. Discussion on fast evaluation of the action for high-order finite element methods can be found in [8] and [12].

We focus on the performance of SYCL implementations for the (i) dot/inner product computations (lines 6 and 9 in Algorithm 1), (ii) vector updates (lines 7, 8 and 10 in Algorithm 1), and (iii) a fused operation (lines 7, 8 and 9 in Algorithm 1 executed in a single kernel).

4 Performance model

To evaluate the performance of kernels we apply, where suitable, the simple performance model

$$T = T_0 + \frac{N_B}{W_a}, \tag{1}$$

where T is the total execution time for a kernel, T_0 is the latency, N_B is the number of bytes accessed (loads and stores) and W_a is the asymptotic memory bandwidth [10]. For a kernel we evaluate N_B by hand and measure T experimentally. The values T_0 and W_a are then estimated by applying linear regression to the measured data. The performance model assumes that the cost of floating point operations is negligible relative to the cost of memory movements, which is reasonable for the problems we will consider. The performance model is not appropriate for devices with multi-level memory caches, since a model with a single value for the memory bandwidth, W_a, is not a reasonable approximation. In the experimental section, we highlight cases where the performance model is not appropriate and where it is not used.

5 Kernel implementations

We present SYCL implementations for the three kernel operations:

1. Vector update (`axpby`),
2. Inner product (`dot`),
3. Fused update (`fused`),

and discuss the implementations in the remainder of this section.

5.1 Vector update

The vector update operation has the form

$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}, \tag{2}$$

where $\alpha, \beta \in \mathbb{R}$ and $\mathbf{y}, \mathbf{x} \in \mathbb{R}^n$. It requires $2n$ reads and n writes. The number of bytes streamed (N_B) is $12n$ for single precision and $24n$ for double precision, and the number of floating point operations is $3n$ ($2n$ multiplications and n additions). The SYCL implementation of the vector update is presented in Listing 1. One work-item computes one component of the array $\mathbf{y}$.

The vector update is a very lightweight kernel, and it is shown in [6] that most parallel programming models are able to achieve a large fraction of peak memory bandwidth for a similar kernel (`triad`). However, this characterisation of the performance is for the limit of large arrays. For practical applications, problem sizes for operations will not necessarily reach the asymptotic problem size, and observed performance can be affected by the latency of different programming models and implementations. In [6] and [11], the array lengths were chosen to be larger than last level cache to eliminate effects for multi-level caches. Large arrays can hide latency that might impact significantly on performance

```
1  template <typename T>
2  sycl::event axpby(sycl::queue& queue,
3                    std::size_t n, T alpha, T beta,
4                    const T* x, T* y, std::size_t wgs,
5                    const std::vector<sycl::event>& events
                         = {})
6  {
7    auto event = queue.submit([&](sycl::handler& h) {
8      h.depends_on(events);
9      h.parallel_for(sycl::range<1>{n, wgs}, [=](sycl::
           nd_item<1> it) {
10       std::size_t i = it.get_global_id(0);
11       if (i < n)
12         y[i] = alpha * x[i] + beta * y[i];
13     });
14   });
15   return event;
16 }
```

Listing 1. Vector update kernel in SYCL.

for the array sizes used in production applications, hence it can be helpful to understand performance characteristics across a wide range of problem sizes.

5.2 Inner product and norm computation

A SYCL implementation for computing

$$\alpha = \mathbf{x} \cdot \mathbf{y} \tag{3}$$

where $\alpha \in \mathbb{R}$ and $\mathbf{y}, \mathbf{x} \in \mathbb{R}^n$, is presented in Listing 2. It exploits SYCL 2020 built-in reductions. We note that the study in [5] did not consider reductions in hipSYCL as SYCL 2020 reductions were not yet available in the hipSYCL [1] implementation. The time complexity of reduction is linear with the number of elements, and in general, it requires $2n$ reads and no global memory writes. The number of bytes streamed (N_B) is $8n$ for single precision and $16n$ for double precision and the number of floating-point operations is $2n$ (n multiplications and n additions).

There are two approaches to reductions in SYCL 2020. The 'basic' approach uses a simple *parallel for* with the SYCL implementation/runtime left to make scheduling decisions, e.g., workgroup size, and reduction implementation decisions, e.g., how much group-local memory to use. We focus on the 'ND-range' approach since in many algorithms the reduction can be fused with other kernels.

For the kernel in Listing 2, there are two parameters that affect the execution of the reduction: the workgroup size and the batch size. The batch size is the number of elements that each work-item in a workgroup will be individually responsible for reducing. Each work-item then cooperates with other work-items in the same group to perform the local reduction. The array length must be a multiple of the product of the batch size and the workgroup size. In practice this can be satisfied using padding.

```
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];
}
```

Listing 2. Inner product computation using SYCL 2020 reductions.

```
template <typename T>
T cg_update(sycl::queue& queue, std::size_t n,
            T alpha, const T* p, const T* y,
            const T* x, T* r,
            std::size_t wg_size, 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) {
        r[i] = -alpha * y[i] + r[i];
        x[i] = alpha * p[i] + x[i];
        sum += r[i] * r[i];
      }
    });
  });

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

Listing 3. Fused update kernel in SYCL.

5.3 Fused update

The final kernel we consider fuses the operations on lines 7, 8 and 9 of Algorithm 1:

$$\begin{aligned} \mathbf{x} &= \alpha\mathbf{p} + \mathbf{x}, \\ \mathbf{r} &= -\alpha\mathbf{y} + \mathbf{r}, \\ \gamma &= \mathbf{r} \cdot \mathbf{r}. \end{aligned} \tag{4}$$

The above three operations can be combined (fused) into a single kernel, which can reduce the number of memory movements, and by increasing the amount of work in a kernel the impact of latency on performance can possibly be reduced. The fused kernel requires $4n$ reads and $2n$ writes, whereas executing three unfused kernels in sequence would require $5n$ reads and $2n$ writes. Fused kernels have been previously investigated for conjugate gradient solvers in [7].

Listing 3 presents the SYCL implementation for the fused update. The implementation is self-explanatory and simply adds vector updates to the inner product kernel (Listing 2).

6 Performance experiments

We measure the performance of the three kernels in the preceding section for a range of problem sizes, and where appropriate perform linear regression to determine the parameters in the performance model. The presented measured bandwidth is the 'effective bandwidth' – the number of bytes loaded and stored divided by the measured time for the kernel. All tests use double precision.

The timing data is collected by running the kernels 300 times for each combination of array length, workgroup size and batch size. The achieved memory bandwidth is then computed by dividing the total amount of data moved (reads and writes) by the total time to run the kernel invocations. Array lengths n are varied from 1024 to 1024^3, increasing by a factor of two for each test. For all cases in which the performance model is fitted to the experimental data, the coefficient of determination (R^2) is not less that 0.99. Data is presented using a log–linear scale, which can give a different visual pereception of the fitting error.

Experiments are performed on an NVIDIA A100-80GB device and on a dual socket Intel 8368Q (Ice Lake) CPU node using hipSYCL [1] and Intel LLVM/SYCL [2] implementations. For experiments on the A100 GPU we use the CUDA backend for both SYCL implementations. We note that hipSYCL uses an OpenMP backend on CPUs, and Intel LLVM/SYCL uses OpenCL. For experiments on the A100 device we use cuBLAS implementations as a performance reference, and on the CPU we use Intel MKL implementations as a performance reference. The performance tests use CUDA v11.4.100, hipSYCL v0.9.2, Intel LLVM/SYCL tag `sycl-nightly/20220222` and LLVM 13.0.1. We use AOT (ahead-of-time) compilation to avoid the cost of JIT while performing the benchmarks. For Intel CPUs we use `-fsycl-targets=spir64_x86_64`. For testing on a CPU using Intel LLVM/SYCL we set the environment variables `DPCPP_CPU_PLACES=numa_domains` and `DPCPP_CPU_CU_AFFINITY=close`. Further details on software

versions used and instructions for reproducing the experiments are provided in Appendix A.

For reference, STREAM Triad on the dual socket Ice Lake node measures a bandwidth of 271 GB s^{-1}. The peak memory bandwidth for the A100-80GB, as reported by NVIDIA, is 1.94 TB s^{-1}.

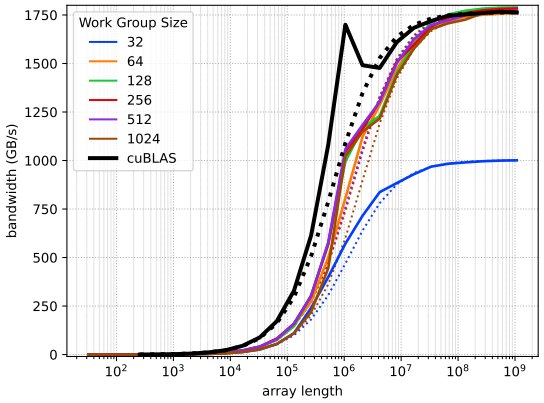
6.1 Vector update

Figure 1 shows the measured memory bandwidth for the axpby kernel (Section 5.1) in the A100 GPU for workgroup sizes from 32 to 1024. For large enough workgroup sizes (> 32) the measured asymptotic memory bandwidth is very close for the two SYCL implementations and cuBLAS. The hipSYCL bandwidth is generally lower than cuBLAS for smaller array sizes, whereas Intel LLVM/SYCL and cuBLAS are closely matched for all array sizes when the number of SYCL work-items per workgroup is greater than 32. Also shown in Fig. 1 is the fitted performance model from Eq. (1) for each case.

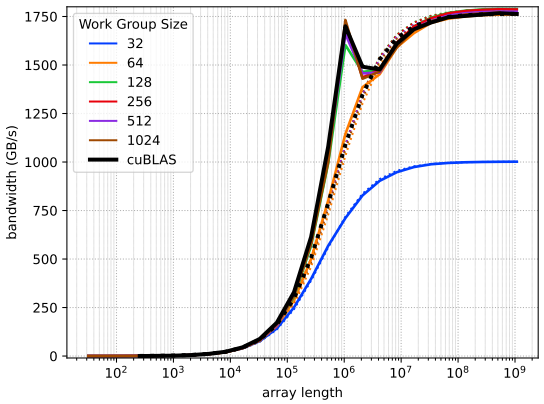
For the A100 GPU, the maximum number of thread blocks per streaming multiprocessor (SM) is 32, and the maximum number of concurrent warps per SM is 64. Therefore, work-groups of size 32 (1 warp per work-group) result in at most 32 concurrent warps per SM or 50% of theoretical occupancy, which explains why the asymptotic performance for the work-group size equal to 32 is significantly lower than the other work-group sizes. Increasing the work-group size to 64 makes it possible to achieve full theoretical occupancy.

Table 1 presents the fitted performance model parameters. As expected from inspection of Fig. 1, the estimated asymptotic memory bandwidths for the SYCL tests are very close to the measured cuBLAS reference (W_a = 1711 GB s^{-1} for cuBLAS). The latency (T_0) for Intel LLVM/SYCL and cuBLAS is similar (around 9 µs), with the latency for hipSYCL two to three times higher. Even though the model is not effective in explaining the spikes for the Intel LLVM/SYCL and cuBLAS cases, as it ignores the effect of multilevel memory caches, the R-squared for the linear fit is greater than 0.99, ensuring that the model is generally valid, except in the small region where cache effects are dominant. An equivalent model can be obtained by applying linear regressions to arrays larger than the last-level cache.

Figure 2 shows the measured bandwidth on a dual socket Ice Lake node. We see that the hipSYCL implementation, which uses OpenMP as the backend, can exploit the cache memory bandwidth when the array fits in the cache. We do not attempt to fit the performance model in this case as a single parameter for the bandwidth is clearly not appropriate. The asymptotic performance of the hipSYCL and Intel LLVM/SYCL implementations is very close, and consistent with the MKL performance and measured STREAM bandwidth of 271 GB s^{-1}. It is noteworthy that the Intel LLVM/SYCL implementation with the OpenCL backend is unable to exploit the cache bandwidth. As would be expected,



(a) hipSYCL



(b) Intel LLVM

Figure 1. Measured performance of the vector update kernel (axpby) on the A100 GPU. Solid colored lines show the measured memory bandwidth for different workgroup sizes. Dotted lines are for the fitted performance model. Heavy black lines are for cuBLAS.

	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

Table 1. Performance model parameters for the vector update kernel (axpby) on the A100 GPU for different workgroup (WG) sizes. For cuBLAS T_0 = 8 µs and W_a = 1711 GB s^{-1}.

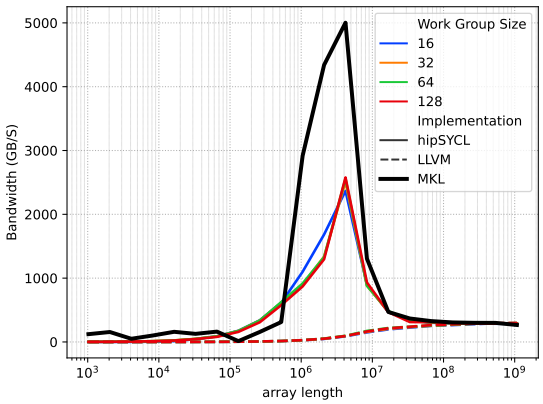


Figure 2. Measured performance of the vector update (axpby) kernel on a dual socket Ice Lake CPU node for hipSYCL (solid lines) Intel LLVM/SYCL (dashed lines) for different workgroup sizes. The heavy black line is the MKL performance.

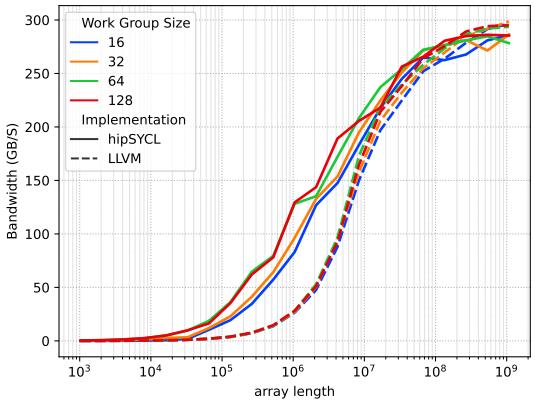


Figure 3. Measured performance of the vector update (axpby) kernel on an Ice Lake node with cache effects removed.

the workgroup size has no significant effect on performance on the CPU.

To eliminate the effect of the cache, we can 'invalidate' the cache by loading an auxiliary array that is larger than the L3 cache (58 368 kB) before executing the kernel. The performance for this case is shown in Fig. 3 and the performance model parameters are presented in Table 2. Clearly the latency is considerably higher for the Intel implementation compared to hipSYCL.

6.2 Inner product

We now consider the inner product kernel from Section 5.2. In Fig. 4 we show the measured memory bandwidth on the

	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

Table 2. Performance model parameters for the vector update (axpby) kernel on the Ice Lake node with cache effects removed. For STREAM Triad W_a = 271 GB s^{-1}.

	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

Table 3. Fitted performance model parameters for the inner product (dot) kernel on the A100 device with a batch size of 16. For cuBLAS T_0 = 23 µs and W_a = 1739 GB s^{-1}.

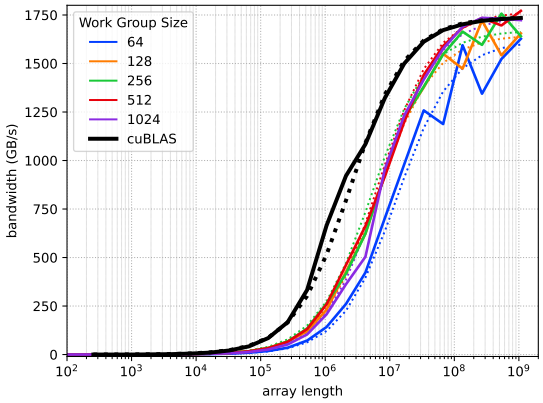
	Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})
64	27	197
128	41	405
256	62	983
512	108	1746
1024	105	1746

Table 4. Fitted performance model parameters for the inner product kernel with a batch size of 256 on the A100 device. cuBLAS values are T_0 = 23 µs and W_a = 1739 GB s^{-1}.

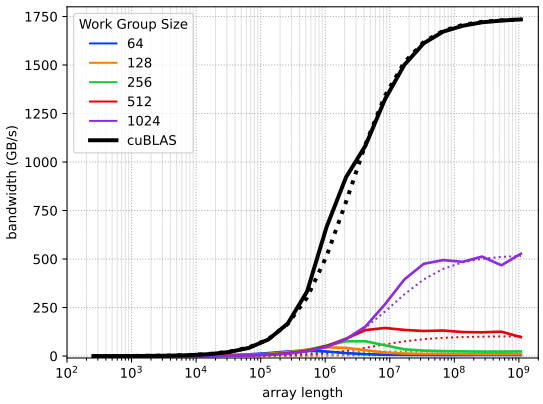
A100 GPU for a batch size (bs) of 16 and for different workgroup sizes. The performance of the built-in reduction is reasonable with hipSYCL compared to cuBLAS, but poor for Intel LLVM/SYCL. At best, the Intel implementation achieves less than 30% of the cuBLAS asymptotic bandwidth.

Examining the performance model parameters in Table 3, we see a high latency for Intel LLVM/SYCL compared to hipSYCL and cuBLAS. The hipSYCL latency is closer to cuBLAS for this test case, with latency approximately three times greater than for cuBLAS when the workgroup size is sufficiently large, whereas for Intel LLVM/SYCL, the latency is more than 10 times greater than for cuBLAS.

Increasing the batch size, giving more work to each work item, improves the Intel LLVM/SYCL performance, but requires a large array length before matching the cuBLAS performance (Fig. 5). Examining the fitted performance model parameters for this case (Table 4), the estimated latency is still large compared to the cuBLAS reference latency.



(a) hipSYCL.



(b) Intel LLVM/SYCL.

Figure 4. Measured performance for the inner product kernel (`dot`) on the A100 device using a batch size of 16. Solid colored lines show the measured memory bandwidth for each workgroup size and dotted lines show the fitted performance model. Heavy black lines are for cuBLAS.

The performance of the kernel on the CPU node is presented in Fig. 6. As for the vector update kernel, the Intel LLVM/SYCL implementation is considerably slower for the problem sizes where hipSYCL and MKL can exploit the CPU cache. The asymptotic performance of the two SYCL implementations and MKL is very close for sufficiently large workgroup sizes. Table 5 shows performance model parameters for the Intel LLVM/SYCL implementation. The performance model is not fitted for hipSYCL as it is not appropriate when the CPU cache is exploited. Again we see that the asymptotic bandwidth is good and effectively equal to the STREAM bandwidth, but the latency is very high.

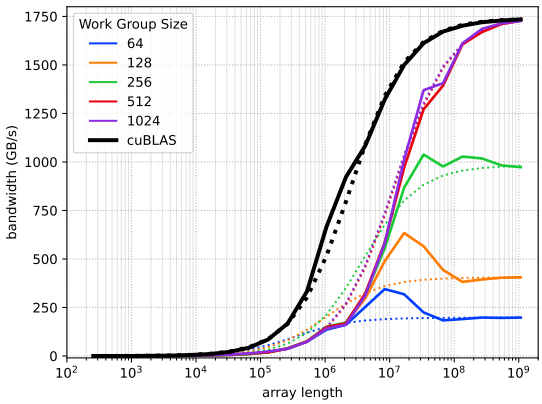


Figure 5. Measured performance for the inner product kernel with Intel LLVM/SYCL with a batch size of 256. Solid colored lines show the measured memory bandwidth for each workgroup size. Dotted lines show the fitted performance model. Black lines are for cuBLAS.

	Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})
16	1884	255
32	1790	268
64	1587	279
128	1564	279

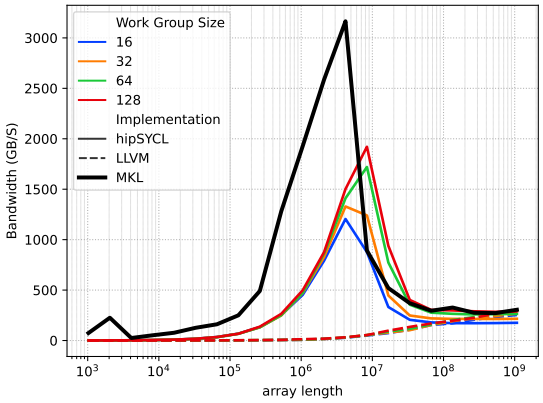
Table 5. Performance model parameters for the inner product kernel (`dot`) using a batch size of 1 for different workgroup (WG) on a dual socket Ice Lake node.

6.3 Fused update

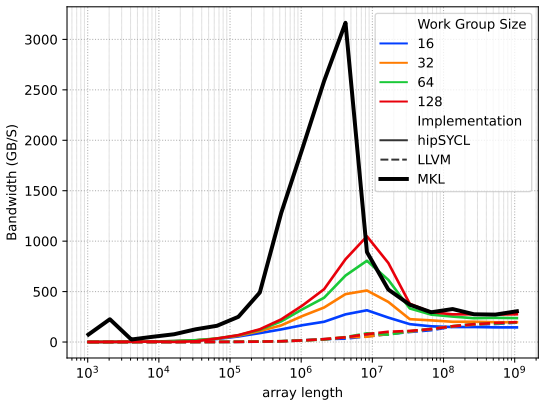
We now consider the fused kernel from Section 5.3, which has the potential to better hide kernel latency. Informed by the results for the reduction kernel, we use a batch size for GPU experiments of 16 for hipSYCL and 256 for Intel LLVM/SYCL, and a batch size of 1 for CPU experiments. The cUBLAS and MKL reference implementations use standard API calls and are therefore not fused.

Figure 7 presents the measured bandwidth for hipSYCL and Intel LLVM/SCYL on the A100 GPU. With hipSYCL, performance is insensitive to the workgroup size for sizes greater than 32, whereas for Intel LLVM/SYCL the workgroup size needs to be greater than 128 before the asymptotic bandwidth is achieved. The performance model parameters are shown in Table 6. The fused kernel hides the latency observed in the previous section for the reduction.

To assess relative performance, Fig. 8 presents the time to execute the (i) fused and unfused kernels with the Intel LLVM/SYCL implementation, (ii) fused and unfused kernels with the hipSYCL implementation normalised by the time to execute the three operations using cuBLAS. For both SYCL

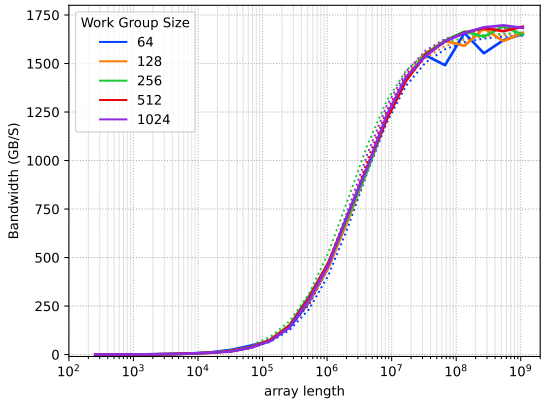


(a) batch size 1.

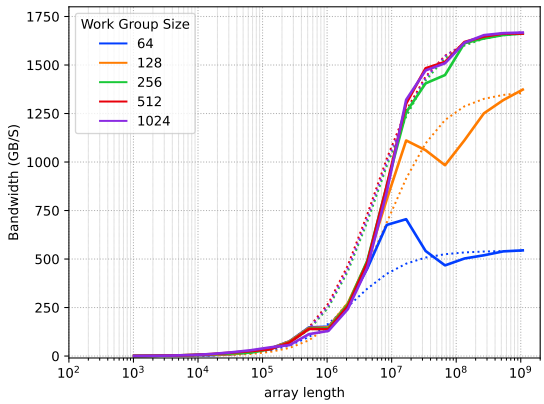


(b) batch size 16.

Figure 6. Measured performance for the inner product kernel (`dot`) on a dual socket Ice Lake node. Colored lines show the measured memory bandwidth for each workgroup size. Dotted lines show the fitted performance. The heavy black line is the MKL performance.



(a) hipSYCL



(b) Intel LLVM/SYCL

Figure 7. Measured performance for the fused update kernel on the A100 device. Solid colored lines show the measured memory bandwidth for different workgroup sizes. The dotted lines show the fitted performance model.

	hipSYCL		Intel LLVM/SYCL	
WG Size	T_0 (µs)	W_a (GB s^{-1})	T_0 (µs)	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

Table 6. Fitted performance model parameters for the Fused Update (`update`) kernel on the A100 device with a batch size of 16 for hipSYCL and 256 for LLVM/SYCL.

implementations we use a workgroup size of 512. It is clear that the fused SYCL kernels are nearly always faster than the unfused SYCL kernels, and the fused hipSYCL implementation is generally faster than the fused Intel LLVM/SYCL implementation. For small array lengths, the cuBLAS implementation is considerably faster than the SYCL implementations, which can be attributed to lower latency.

Figure 9 presents the measured performance of the fused kernel on the CPU node. For both SYCL implementations we use a workgroup size of 128. As seen for other kernels, hipSYCL outperforms Intel LLVM/SYCL for most array sizes, and both implementations converge to the same asymptotic limit for large arrays. hipSYCL is run only for a workgroup

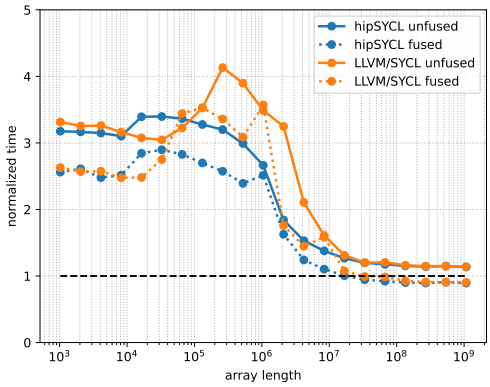


Figure 8. Normalised speed the fused update kernel on the A100 GPU. A normalised time of one corresponds to three operations (unfused) using cuBLAS.

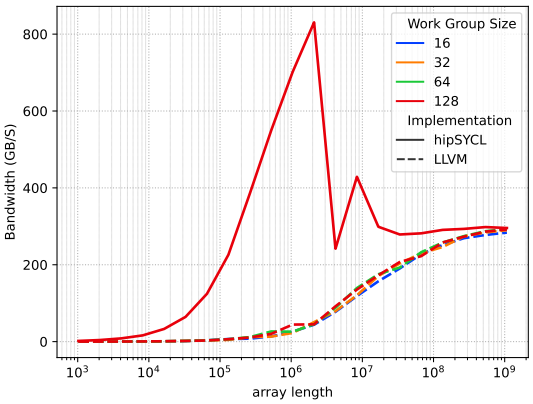


Figure 9. Measured performance for the fused update on a dual socket Ice Lake CPU node.

size of 128 since the reduction support was still in development and was not fully compliant with SYCL 2020 by the time we collected the results.

Table 7 shows the fitted performance model parameters for Intel LLVM/SYCL on the CPU node. As for other kernels, we do not attempt to fit the performance model to the hipSYCL CPU results. Consistent with other cases, the asymptotic bandwidth is very close to the STREAM bandwidth, but the latency is high.

Figure 10 presents the time to execute (i) fused and unfused kernels with Intel LLVM/SYCL, (ii) fused and unfused kernels with hipSYCL,normalised by the measured time using MKL. Note, unlike in preceding figures, that we use a log scale for the normalised time in Fig. 10 since the measured data spans a wide range of values. We observe that MKL is almost always faster for smaller array sizes. With hipSYCL,

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

Table 7. Fitted performance model parameters for the fused update kernel (`dot`) using and a batch size of 1 on a dual socket Ice Lake node.

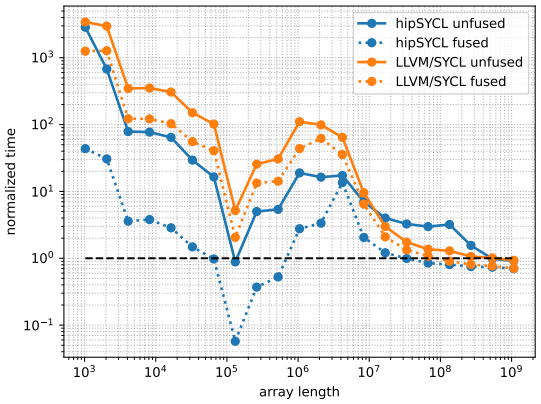


Figure 10. Normalised fused and unfused update kernel timings. A normalised time of one is for the three operations (unfused) using MKL.

the fused kernel is considerably faster than the unfused kernels for nearly all array lengths. The performance with Intel LLVM/SYCL is poor for smaller arrays. For large array sizes, the performance of the fused kernels converges, as does the performance of the unfused implementations.

7 Conclusions

The performance of simple matrix-free conjugate gradient SYCL kernels with two SYCL implementations on GPU and CPU hardware has been examined, and compared to vendor BLAS implementations. The kernels are all memory bandwidth limited. Results indicate that for large problem sizes the performance of the SYCL kernels and the vendor libraries is very close. However, for smaller array sizes different implementations have considerably different performance characteristics. The simple performance model, applied to the cases that do not demonstrate the effects of multi-level memory caches with different bandwidth, 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. In particular, the latency for reductions using SYCL is high

compared to vendor implementations. Of particular note is the poor performance of the Intel LLVM/SYCL implementation on a CPU (which uses an OpenCL backend), which we attribute to very high latency. The hipSYCL implementation, (which uses an OpenMP backend for CPUs) performs considerably better.

For the tests presented, hiding latency requires sufficiently large arrays (including for cuBLAS); for the considered cases, with the SYCL implementations the double precision array length needs to reach approximately 10^8 before the asymptotic memory bandwidth was observed. The matrix-free kernels considered require only four vectors to be stored, which for a length of 10^8 requires 3.2 GB of storage for double precision arrays of length 10^8. In a typical application, particularly for unstructured grids, storage of other data structures in the solver will consume considerably more memory than the conjugate gradient vectors and may dictate the maximum array size. Our observations on latency lend further support to the use of matrix-free solvers. Storage of a sparse matrix requires considerably more memory than the four conjugate gradient vectors, and could limit problem sizes and make it difficult to achieve the the asymptotic 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. All three topics are important for overall application performance, with a careful examination of each topic supporting a deeper understanding of whole application performance.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through the Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV) project (EP/S005072/1). Resources were provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service, provided by Dell EMC and Intel using Tier-2 funding from EPSRC (EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council.

A Reproducibility

The source code used for the experiments and instructions for reproducing the experiments are provided at https://github.com/Wells-Group/sycl-cg-matrix-free.

References

[1] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 1 pages. https://doi.org/10.1145/3388333.3388658

[2] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. 2020. Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 2 pages. https://doi.org/10.1145/3388333.3388653

[3] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM.

[4] Noel Chalmers and Tim Warburton. 2020. Portable high-order finite element kernels I: Streaming Operations. (2020). https://arxiv.org/abs/2009.10917

[5] Tom Deakin, Simon McIntosh-Smith, S. John Pennycook, and Jason Sewall. 2021. Analyzing Reduction Abstraction Capabilities. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, St. Louis, MO, USA, 33–44. https://doi.org/10.1109/P3HPC54578.2021.00007

[6] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.

[7] Maryam Mehri Dehnavi, David M Fernández, and Dennis Giannacopoulos. 2011. Enhancing the performance of conjugate gradient solvers on graphic processing units. *IEEE Transactions on Magnetics* 47, 5 (2011), 1162–1165.

[8] Paul Fischer, Misun Min, Thilina Rathnayake, Som Dutta, Tzanio Kolev, Veselin Dobrev, Jean-Sylvain Camier, Martin Kronbichler, Tim Warburton, Kasia Świrydowicz, and Jed Brown. 2020. Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 562–586. https://doi.org/10.1177/1094342020915762

[9] Jeff R. Hammond and Timothy G. Mattson. 2019. Evaluating Data Parallelism in C++ Using the Parallel Research Kernels. In *Proceedings of the International Workshop on OpenCL* (Boston, MA, USA) *(IWOCL'19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 6 pages. https://doi.org/10.1145/3318170.3318192

[10] Roger W. Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (1994), 389–398. https://doi.org/10.1016/S0167-8191(06)80021-9

[11] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2021. On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements. In *International Workshop on OpenCL* (Munich, Germany) *(IWOCL'21)*. Article 8, 13 pages. https://doi.org/10.1145/3456669.3456701

[12] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Tim Warburton. 2019. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications* 33, 4 (2019), 735–757.