Benchmarking and Extending SYCL Hierarchical Parallelism

Tom Deakin and Simon McIntosh-Smith
Department of Computer Science
University of Bristol
Bristol, UK
Email: tom.deakin@bristol.ac.uk

Aksel Alpay and Vincent Heuveline
Engineering Mathematics and Computing Lab and
Interdisciplinary Center for Scientific Computing
Universität Heidelberg
Heidelberg, Germany
Email: aksel.alpay@uni-heidelberg.de

***Abstract*—SYCL is an open-standard, parallel programming model for programming heterogeneous devices from Khronos. It allows single-source programming of diverse attached devices in a cross-platform manner in modern C++. SYCL provides different layers of parallel abstractions, including Same Instruction Multiple Thread (SIMT) kernels, data-parallel loop concurrency and hierarchical parallelism. We discuss Scoped Parallelism as an extension to the existing Hierarchical Parallelism in SYCL, and highlight the advantages and disadvantages of these models from the perspective of the programmer and an implementer of SYCL. In this paper, we compare writing benchmark programs using SIMT kernel, hierarchical parallelism and scoped parallelism paradigms, and present results running on a high-performance CPU and GPU.**

I. Introduction

Heterogeneous parallel programming models are a vital component in providing a way for writing high performance applications in a way that is productive and performance portable across a wide variety of different devices. The extent to which different models provide this performance portability has been discussed elsewhere (e.g. [1]). The model must provide a way to express concurrency and parallelism in an abstract way so that implementations of that model have sufficient flexibility to map to the parallel features of the underlying target processor. This range of implementation options, both in how the user expresses parallelism and in how an implementation of any programming model maps to the underlying hardware, can effect the degree to which performance portability is realised in practice.

SYCL is one such parallel programming model [2]. Originally released in 2015, it provides a range of abstraction levels for writing parallel applications: data-parallel kernels, SIMT-style NDRange kernels (see Section II-A), hierarchical parallelism, and high-level task-based concurrency to allow for parallel scheduling of kernel tasks. There are multiple implementations of the SYCL specification; in this work, we will use the hipSYCL [3] and Intel oneAPI DPC++ implementations. In this study, we focus on expressing parallelism inside of a kernel, and in particular compare NDRange kernels with hierarchical parallelism and scoped parallelism.

In particular, we make the following contributions:

- We benchmark the performance of NDRange and Hierarchical Parallelism on CPUs and GPUs using the hipSYCL and DPC++ in-development implementations of SYCL.
- We implemented a new simple DGEMM benchmark application for testing parallel schemes in SYCL.
- Recent hipSYCL versions include support for a new model of parallelism as an extension to SYCL called scoped parallelism. We provide the first evaluation and discussion of this proposed model.

The remainder of the paper is structured as follows. Section II summarises the NDRange and Hierarchical parallelism features currently available in SYCL. We compare these features to those available in other programming models in Section III. We introduce the scoped parallelism model in Section IV. Section V describes the benchmark kernels we use, with their results presented in Section VI. We summarise our findings in Section VII.

II. Expressing Parallelism in SYCL

Data parallel heterogeneous SIMT programming will be familiar to those who know the heterogeneous programming models OpenCL, CUDA, or HIP. In this study, we focus on SYCL and will use that terminology throughout. In this style of parallel programming, a kernel function is written, and invoked once per work-item in the parallel iteration range with a call to `parallel_for` over a `range`. The kernel body can be thought of as the body of a loop, with the loop iterations being assigned to work-items which are allowed to execute in parallel. In SYCL, the `range` based kernel implies that all the work-items can be executed concurrently and there are no dependencies — in particular there is no way to synchronise between work-items. This gives the implementation freedom to map the work-items to threads, SIMD lanes, etc. as appropriate.

A. NDRange kernels

SYCL distinguishes between this basic `range` based kernel model and the `nd_range` model of OpenCL (etc), where additional semantics apply. In the `nd_range` model, the parallel iteration space of the kernel is given additional structure by grouping those work-items into work-groups. In SYCL we express this by giving the `parallel_for` an `nd_range`

instead of a `range`. This allows the programmer to specify the number of work-items in each work-group, and importantly this adds semantic constraints which allow synchronisation at explicit barriers between work-items in the work-group. This structure also adds constrains on the implementation, which must ensure that all work-items within a group make forward progress so that they may all reach these barriers. For example, if there were no explicit barriers, the SYCL implementation could implement work items as iterations of a sequential loop within a work group. The SYCL specification describes these restrictions in §3.8.3.4 of the specification [2]. Programmers need the ability to synchronise between work-items in the work-group to develop optimised applications; particularly in order to use local memory cooperatively within one group. We present some examples of such applications in Section V.

The SYCL specification explicitly allows library-only implementations of the standard. For those simple C++ library implementations that target CPUs without additional compiler support, these constraints add challenges to implement `nd_range` parallelism efficiently: the additional forward-progress guarantees require that each work-item is put into its own thread, fiber, coroutine or other mechanism to maintain independent stacks for each work-item such that execution of a work item can be preempted. Note that forward-progress guarantees in SYCL are limited to the requirement that all work-items reach group barriers before execution continues beyond the barrier[1]. This can add considerable overhead and limit vectorization across work-items in library-only SYCL implementations if explicit barriers or collective group algorithms are used (see e.g. [4] for details). However, mapping `nd_range` parallelism to e.g. OpenCL devices or CUDA GPUs is straight-forward due to the similarity of the models.

B. Hierarchical parallelism

Hierarchical parallelism has been available in SYCL since it was first released as version 1.2 in 2015. The motivation was to provide a *syntactic* alternative to programming in an NDRange style.

The hierarchical parallelism APIs gave a way to express the structure of `nd_range` kernels, and barriers within them, using nested levels of parallelism instead of a single-level SIMT programming paradigm. Instead of defining the structure in advance, an outer `parallel_for_work_group` was used to express what each work-group should do. Inside this, `parallel_for_work_item` could then be included to express what each work-item in the work-group should do. At the end of the `parallel_for_work_item` there would be an implicit barrier. This is the only synchronization mechanism in SYCL's hierarchical parellelism, since it does not allow the explicit barriers that can be employed by the programmer in `nd_range` parallelism. Note too that the implicit barriers are non-optional and there is no mechanism to state they are not required by a particular kernel.

[1]Note that this is not an issue on GPUs, where the execution model of the hardware already provides barrier semantics.

As such, this notation becomes quite convenient for expressing a two-level hierarchical structure of work-groups and work-items, while in principle retaining the ability to formulate all programs that are expressible in `nd_range` parallelism.

For hierarchical parallelism, SYCL allows the number of work-groups launched by a `parallel_for_work_group` call to specify the size of the work-groups or leave it up to the implementation at runtime. If the work-group size is unspecified, the programmer must specify a number of *logical* work-items in the nested `parallel_for_work_item` calls. These work-items are logical in the sense that they do not necessarily map one-to-one with the underlying physical work-items in the work-group, but state the amount of concurrent work within the work-group. A programmer can use these logical work-items for to grow/shrink the effective number of work-items in the work-group on demand even when work-group sizes were specified.

While the SYCL specification states regarding hierarchical parallelism that "this mode is purely a compiler feature and does not change the execution model of the kernel", it is important to realize that due to the lack of explicit barriers in the work-item loops, hierarchical parallelism does not force a SYCL implementation to provide the same forward progress guarantees for work-items as `nd_range` parallelism. This does in fact enable SYCL implementations to implement it using a different execution model internally: in particular on CPUs it is often very efficient to employ multi-threading across work-groups, and a (vectorized) loop across work-items within a work-group. The `parallel_for_work_group` and `parallel_for_work_item` APIs provide a convenient way for SYCL implementations to offer this alternative form of parallelism, and hipSYCL implements hierarchical parallelism on CPUs in this fashion. It is also noteworthy that this makes hierarchical parallelism efficiently implementable on CPUs as a C++ library, without any additional compiler logic.

On a data parallel accelerator such as a GPU, implementing hierarchical parallelism however requires non-trivial compiler support. Outside of `parallel_for_work_item`, it must be guaranteed that code is only executed once within a work group as if only a single work item were active. However, the kernel model on GPUs usually relies on the number of work items being static for the duration of the kernel (and known at the time of kernel submission). Therefore, additional operations usually have to be inserted by the compiler to prevent observable effects caused by the additional work items (e.g. by masking their memory accesses). Coupled with the requirement of the specification that variable declarations outside of `parallel_for_work_item` should be allocated in local memory, performance can be difficult to predict since the compiler might have to insert additional barriers to guard the local memory accesses, and per-work-group local memory is usually slower than per-work-item private memory.

Listing 1 shows a SYCL `nd_range` kernel. The same kernel is implemented using SYCL hierarchical parallelism and shown in Listing 2. They of course have much in

Listing 1. SYCL nd_range parallelism

```
Q.submit([&](handler& cgh) {
  // global memory, shared by all work-
     items
  accessor A {bufA, cgh};

  // local memory, shared by work-items
     in work-group
  local_accessor<int> locA {16, cgh};

  cgh.parallel_for(nd_range<1>{{1024},
     {16}}, [=](nd_item<1> it) {
    // private variable
    int i = it.get_global_id(0);

    locA[it.get_local_id(0)] = i;

    group_barrier(it.get_group());

    if (it.get_group().leader()) {
      int t = 0;
      for (int j = 0; j < 16; ++j)
        t += locA[j];
      A[it.get_group_id(0)] = t;
    }
  });
};
```

Listing 2. SYCL hierarchical parallelism

```
Q.submit([&](handler& cgh) {
  // global memory, shared by all work-
     items
  accessor A {bufA, cgh};

  cgh.parallel_for_work_group(range
     <1>{64},
    [=](group<1> g) {

    // local memory,
    // shared by work-items in work-group
    int locA[16];

    g.parallel_for_work_item(range
       <1>{16},
      [=](h_item<1> it) {
        // private variable
        int i = it.get_global_id(0);

        locA[it.get_local_id(0)] = i;
    }); // Implicit barrier

    if (g.leader()) {
      int t = 0;
      for (int j = 0; j < 16; ++j)
        t += locA[j];
      A[it.get_group_id(0)] = t;
    }
  });
};
```

common, but there a number of key differences to highlight. For `nd_range`, the size of the work-groups have been specified in the call to `parallel_for`, but with hierarchical parallelism we leave this up to the implementation, giving us later flexibility to launch a loop over logical work-items. Note, we could specify the size of the work-groups and so remove the need to specify a `range` in the nested `parallel_for_work_item` call.

An important implication is that of which memory space is used for variables. Variables defined inside `parallel_for` and `parallel_for_work_item` scope are placed in work-item private memory, and so each work-item has its own personal copy. The accessors defined inside command group scope typically refer to global memory, and is memory available for all work-items. For work-group local memory, there is some difference in the two approaches. For `nd_range` kernels, we must use a local accessor in command group scope to specify the amount of local memory available to each work-group. Note that this means the size of local memory requested is known only at runtime. In comparison, variable definitions in `parallel_for_work_group` are placed directly in *local* memory, shared between all work-items in the work-group. As such, the size of these variables is known at compile time, and array-types declared in this manner must be of a fixed size — if runtime-size arrays are required, local accessors can be used as well. As such, there are two ways to declare local memory arrays for hierarchical parallelism: within work-group scope if they are of a static size known at compile time, or via accessors for both compile time and runtime sizes just as for NDRange kernels.

Importantly, any scalar variables declared in work-group scope are *also* placed by default in local memory, with a copy shared between all work-items in the work-group. If private variables are required, they must be annotated as such with the `private_memory` class. If the compiler is able to do so, it might optimise away such constant local variables, but this is difficult to predict and might depend on the internal heuristics used by the compiler and the quality of the implementation.

These examples show that in principle, both the hierarchical and `nd_range` parallelism in SYCL can express the same concurrency and so is ideally merely a matter of programmer preference or productivity based on the algorithm they wish to implement.

SYCL 2020 was a significant update to the standard, but little was updated for hierarchical parallelism. A large number of new features, in particular sub-groups, expanded group algorithms and first-class reduction support, have not yet been fully integrated into hierarchical parallelism. Indeed,

the hierarchical parallelism does not extend to exposing sub-groups, and is limited to two levels: work-groups and work-items.

III. Related Work

Hierarchical parallelism is not unique to SYCL. Other parallel programming models include some notion of this.

Table I summarises the levels of parallelism available in SYCL's hierarchical parallelism and our scoped parallelism extension, and in OpenMP and Kokkos. We suggest a mapping of each level to the SYCL nomenclature based on our experiences using these models; note that there are other such mappings, in many cases the mapping in not uniform across all implementations, and an implementation may use a different mapping based on the specific use of the constructs.

A. OpenMP

Hierarchical parallelism in OpenMP can be expressed in a wide variety of ways [5]. Often, the parallelism that is realised is dependent on how the parallel threads are implicitly (or explicitly) bound to the parent OpenMP directive. Threads can fork other threads under the fork-join model, through the use of nested `parallel` regions. This allows, from the perspective of the application programmer, an unlimited number of parallel levels. In practice, the degree to which these "threads" will be truly parallel is limited by the implementation's `max-active-levels-var` Internal Control Variable (ICV). In addition, the `loop` construct, first introduced in OpenMP 5.0, can be used in a nested manner to define concurrent work. This construct uses the `bind()` clause to direct how the loop might be mapped onto the parent region: a teams or parallel region. Note that OpenMP also provides three levels of explicit parallelism: teams, threads and SIMD though the use of `teams`, `parallel` and `simd` constructs. In OpenMP it is possible to write multiple levels of parallelism using the `parallel` directive alone in contrast to the different APIs for different hierarchical levels used in Kokkos and SYCL. It is also possible to be explicit and map to the three levels that OpenMP exposes.

Table I shows a number of ways to map OpenMP parallelism. Teams are usually mapped to work groups, but implementations today map one of either threads or SIMD vectors to work-items. Using the combined construct `teams distribute parallel for simd` exposes a single level of parallelism where synchronization is disallowed, and so are usually mapped to work-items. Note that the `loop` construct may be nested and has explicit or implicit binding behaviour; therefore the mapping is highly situational.

B. Kokkos

Kokkos exposes three levels of parallelism: teams, threads, and vectors [6]. Various Policy types are used to distinguish the levels on the usual parallel dispatch API. Up to three levels of nested parallelism is allowed: each level corresponds to the policies. The notion of threads and teams aligns with SYCL's model of work-items and work-groups respectively. In NDRange kernels, SYCL exposes sub-groups to somewhat align with Kokkos's vector policy. In hierarchical parallelism however, they are not currently exposed to the programmer in the form of a parallel loop; vector types could be used instead. We refer reader interested in generalised performance comparisons between these programming models to previous work by us and others, such as [1], [7].

Table I shows how the three levels of parallelism in Kokkos map onto the SYCL execution model. It is a fairly straightforward mapping, however note that `ThreadVectorRange` is asking for vectorized operations, which in SYCL are usually exposed via subgroups.

IV. Scoped parallelism in hipSYCL

hipSYCL provides the scoped parallelism model as an extension, which targets similar use cases as hierarchical parallelism in SYCL 1.2.1. The first hipSYCL release to include support for the first version of scoped parallelism was hipSYCL 0.9.0 from 2020. In this work, we describe the latest iteration of scoped parallelism, which is currently being finalized and available as a prototype implementation in the hipSYCL repository[2].

Listing 3 provides an example code using scoped parallelism. We implement the same program as in Listings 1 and 2 so as to show the syntactic and semantic differences. We refer interested readers to the hipSYCL repository in footnote 2 for more detailed examples.

The fundamental difference between scoped parallelism and hierarchical parallelism is that scoped parallelism provides the SYCL implementation the freedom to choose the degree of parallelism that is exposed to the backend. For example, on GPU the SYCL implementation could decide to execute the kernel with all work-items running right from the start, while on CPU it could map kernel execution to multithreading across work-groups, and vectorized loops across work-items within a work-group. This additional flexibility for the SYCL implementation can introduce performance advantages. Additionally, it avoids both the implementation complexities of hierarchical parallelism on accelerators, and `nd_range` parallelism on CPUs for library-only implementations.

In scoped parallelism terminology, the actual degree of parallelism that is exposed to the backend is called the *physical parallelism*, while the user-requested work-group size is referred to as *logical parallelism*. The user can then distribute logical work-items across the physical parallel iteration space using the `distribute_items()` function. Additionally, `distribute_groups()` can be used to subdivide a group into smaller subunits and distribute those subunits across the physical iteration space. This allows the SYCL implementation to expose hardware subgroups, or even e.g. nested multi-dimensional tiling on CPUs. `distribute_groups()` calls can be nested arbitrarily deep.

Declared variables are always allocated in private memory: either in the private memory of the physical work-item

[2]https://github.com/illuhad/hipSYCL/pull/619

TABLE I
COMPARISON OF THE FIRST THREE NESTED PARALLELISM LEVELS FOR VARIOUS PROGRAMMING MODELS.

Model	Nesting level	Expressed by	Mapped to
SYCL (hierarchical)	1	`parallel_for_work_group()`	Work-groups
	2	`parallel_for_work_item()`	Work-items
	3	–	–
SYCL (scoped)	1	`parallel()`	Work-groups
	2	`distribute_groups()`	Subgroups or work-items
	3	`distribute_groups()`	(Sub-)Subgroups or work-items
	any	`distribute_items()`	Work-items
OpenMP	1	`teams`	Work-groups
	2	`threads`	Work-item or ignored
	3	`simd`	Ignored or work-item
	1	`teams distribute parallel for simd`	Work-item
	any	`loop bind()`	Dependent on nesting and binding
	any	`parallel`	Dependent on nesting and ICVs
Kokkos	1	`TeamPolicy`	Work-groups
	2	`TeamThreadRange`	Work-items
	3	`ThreadVectorRange`	Subgroup

if the declaration is outside of `distribute_items()`, or inside the private memory of the logical work-item if the declaration is inside of `distribute_items()`. This can avoid performance surprises in SYCL 1.2.1 hierarchical parallelism, where variable declarations outside of `parallel_for_work_item` are always allocated in local memory. Local memory allocation in scoped parallelism is possible if the user explicitly requests it using either local accessors or the `local_memory` wrapper type.

Synchronization within a group can be performed either by invoking `distribute_items_and_wait()` or `distribute_groups_and_wait()`, or by explicitly invoking a `group_barrier()` outside of `distribute_items()`.

The scoped parallelism model also supports SYCL 2020 reductions and some group algorithms.

Scoped parallelism can be efficiently implemented as a layer on top of both SYCL 1.2.1 hierarchical parallelism and `nd_range` parallel for models, allowing efficient library-only implementations of the model both for CPUs as well as on top of other data parallel heterogeneous models such as CUDA.

Because the number of levels of parallelism is arbitrary and multiple types of groups might be utilized simultaneously, the notion of a local ID as defined in SYCL 2020 is now ambiguous: the concept of a local ID is only meaningful if it is known from which work-group the local ID stems. To this end, the group object must be provided to `get_local_id()` calls. Alternatively, the `s_item::get_innermost_local_id()` method can be used to explicitly refer to the current level.

In addition, sub-groups in a scoped parallelism sense no longer necessarily map to SYCL's subgroup concepts. In SYCL, `nd_range` subgroups are only one-dimensional since they typically refer to SIMD units. However, in scoped parallelism multi-dimensional sub-groups might be required. While those might be implemented as true SYCL subgroups in some cases, this is not always possible, and the implementation might instead choose to map them to something else such as simple nested loops on CPUs, trivial scalar groups or abstrations such as CUDA Cooperative Groups on GPUs.

V. BENCHMARKS

In order to explore NDRange, hierarchical and scoped parallelism in SYCL, we introduce a number of simple benchmark codes.

A. SYCL-Bench

The SYCL Benchmark suite (SYCL-Bench) contains a collect of idiomatic kernels implemented in SYCL for microbenchmarking, testing SYCL runtime overheads and proxies of applications [4], [8]. We select three kernels for our study: nbody, reduction and segmented reduction. These benchmarks have been chosen because they already provide implementations in `nd_range` and `hierarchical` parallelism. We have additionally extended them with support for scoped parallelism. For all kernels, we use FP64.

The **nbody** kernel implements a typical $O(N^2)$ algorithm for simulating the movement of particles in a 3-dimensional domain under the influence of a $1/r^2$ force such as gravity. Each work-item updates the movement of a particle based on its distance to the other particles in the domain. The hierarchical and scoped parallelism versions of this benchmark use `private_memory` allocations of variables inside work-group scope. In addition, the scoped parallelism version allows for the removal of redundant work-group barriers, which exist in the hierarchical parallelism version due to the implicit barriers at the end of a `parallel_for_work_item`.

The **reduction** kernel implements the summation (accumulation) of a large array, using a tree-based parallel pattern. Local memory is used to store partial results, with the number of active work-items accumulating halving at each step. One value per work-group is written to global memory. The kernel is re-enqueued until the size of the input is small enough to

Listing 3. hipSYCL scoped parallelism

```
Q.submit([&](handler& cgh) {
  // global memory, shared by all work-
     items
  accessor A {bufA, cgh};

  cgh.parallel(range<1>{64}, range<1>{16}
    [=](auto grp) {

    // local memory,
    // shared by work-items in work-group
    local_memory<int[16], decltype(grp)>
        locA;

    // distribute_groups is optional,
    // but can expose subgroups of the
    // work group
    distribute_groups(
      grp, [&](auto subgroup){

      distribute_items(
        subgroup, [=](s_item<1> l_id) {

          // private variable
          int i = l_id.get_global_id(0);

          locA[l_id.get_local_id(grp)] =
              i;
        });
      });
    group_barrier(grp);


    single_item(grp, [&](){
      int t = 0;
      for (int j = 0; j < 16; ++j)
        t += locA[j];
      A[it.get_group_id(0)] = t;
    });
  });
};
```

be reduced by a single work-group, following a divide-and-conquer approach.

The **segmented reduction** kernel is similar, however only performs a single reduction step, returning one value per work-group. The final result is accumulated on the host in serial.

The version SYCL-Bench we used is available on GitHub at https://github.com/illuhad/sycl-bench/tree/scoped-parallelism.

B. DGEMM

In a real application, users should always defer to using a highly-optimised off-the-shelf implementation of a matrix multiplication, however the DGEMM kernel itself provides a useful parallel pattern for exploring types of hierarchical parallelism. In particular, it is well understood, a good test of compiler optimizations such as vectorization, and the expressibility of parallelism in the programming models.

Matrix multiplication of a $N \times P$-matrix A and a $P \times M$-matrix B to form the $N \times M$-matrix C can be expressed as the following:

$$C_{ij} = \sum_{k=0}^{P-1} a_{ik} \times b_{kj} \quad \forall i \in [0, N), j \in [0, M)$$

The computation of each element of C can be computed concurrently, and a naive parallel implementation follows this scheme. We call this our **Simple** kernel, and it is implemented using `range`-based `parallel_for` in SYCL.

A common optimisation is to tile the matrix into small blocks to significantly improve the data reuse of the A and B input matrices. The details of this optimisation can be found in many parallel programming tutorials. In our SYCL implementation, we compute 16×16 tiles of the C matrix, iterating over tiles of A and B. One work-group is launched per tile, and 256 (physical) work-items co-operate to compute the tiles. Tiles of A and B are first copied into *local memory*, which often provides performance benefits over global memory where the memory space has hardware support, such as on GPUs. We call this our **NDRange** kernel, as it is implemented using a `nd_range`-based `parallel_for`. It requires work-group barriers to synchronise work-items after copying tiles of A and B into local memory, and after updating the tile of C based on them.

This same tiled algorithm can be implemented using SYCL's hierarchical parallelism notation: `parallel_for_work_group` and `parallel_for_work_item`. We have made no other changes aside from writing the NDRange kernel in this form. We call this implementation our **Hierarchical** kernel.

Finally, we have used the Scoped Parallelism extension to SYCL described in Section IV to implement the tiled matrix multiplication. We call this our **Scoped** kernel. This benchmark exposes the same two levels of parallelism (tiles and work-items) as the preceding hierarchical version, so converting from hierarchical parallelism is mostly a matter of nomenclature. Note that barriers between parallel work-items are now explicitly used via the new `distribute_items_and_wait` API call, and the memory for tiles is explicitly marked as local.

This benchmark is implemented using double-precision floating point (FP64). An optimised matrix multiplication should be limited by the peak floating-point performance of the processor, however our focus in this study is on the relative performance of the mode of parallelism.

The benchmark is available on GitHub at https://github.com/UoB-HPC/sycl_dgemm.

VI. Results

The benchmarks introduced in the preceding section (Section V) are run on two processors: a dual-socket Cascade Lake

CPU, and a NVIDIA V100 GPU.

The Cascade Lake CPU is an Intel Xeon Gold 6230 CPU, running 2.10GHz. It has 20 cores, resulting a 40 core node in our dual-socket configuration. It has peak main memory bandwidth of 282 GB/s and peak FP64 floating point performance of 2.688 TFLOP/s.

The NVIDIA V100 PCIe GPU has 900 GB/s of main memory bandwidth and a peak FP64 floating point performance of 7 TFLOP/s.

For our choice of SYCL implementations, we used the Intel oneAPI version 2021.3.0 and a nightly build of the DPC++ compiler from August 19, 2021; and the hipSYCL implementation from the branch identified in footnote 2. For hipSYCL, we used the Clang/LLVM 11 and GCC 10.3 compilers for code generation.

For the hipSYCL measurements, the hipSYCL version is usually more relevant than the Clang/LLVM version for the performance of the specific benchmark applications that we investigate in this work. This is because the SYCL NDRange, hierarchical and scoped models are implemented inside hipSYCL, which controls how the C++ constructs are mapped to the hardware, while the compilers used by hipSYCL are mainly responsible for backend code generation. As such, the main factor for the relative performance between different models is usually the hipSYCL version. However, with hipSYCL on CPU, some differences may arise due to autovectorization behavior. We will point out where this play a role.

For both the DGEMM and SYCL-Bench benchmarks, we show results on the Cascade Lake CPU and V100 GPU using DPC++ and hipSYCL.

A. DGEMM

The results for the DGEMM benchmark are shown for the Cascade Lake CPU in Figure 2, and for the V100 GPU in Figure 1. The figures show the estimated obtained GFLOP/s. We use square matrices of the order indicated. The simple model of $2 \times N \times M \times P$ floating point operations are required to compute the matrix product is used to provide these performance metrics.

For the GPU in Figure 1, the tiling optimisation provides an important improvement, and that programmers see significant benefit from the complexities of managing work-groups for data locality. Note that expressing this kernel as hierarchical or scoped parallelism, rather than using NDRange kernels, provides further benefit when using hipSYCL (Figure 1a): performance increases by around 1.25x. Hierarchical parallelism is less performant when using the experimental DPCPP CUDA backend (Figure 1b). Profiling the hipSYCL build with *NVPROF* shows that the NDRange kernel required 48 registers, with the hierarchical and scoped parallelism kernels using only 32. All kernels used 256 threads per thread block. This is likely due to the hierarchical and scoped parallelism kernels utilizing local memory declared using sizes that are known at compile-time, while the `nd_range` kernel utilizes a local accessor. Since local accessors support sizes known only at runtime, it needs to additionally store the extents of the local memory regions. The CUDA Occupancy Calculator Spreadsheet[3] implies this reduction in register pressure increases occupancy from 63% to 100%, and allows for the kernel to reach the hardware limit of 64 warps per SM instead of only 40. For a compute bound code, it is important to ensure sufficient warps are available for scheduling in order to achieve good performance.

On CPU architectures, we find the challenges of NDRange parallelism in library-only SYCL implementations identified by Lal et al [4], as seen in Figures 2a and 2b. With compiler support, these problems are alleviated as shown in Figure 2c. Using hipSYCL's library-only CPU backend, we see high performance for hierarchical parallelism due its weaker requirements for work-item forward progress guarantees, resulting in efficient mapping of work-groups to CPU cores.

The Clang 11 compiler is used for our experiments in Figures 2a, which for the prototype hipSYCL implementation of scoped parallelism fails to perform effective vectorisation compared to hierarchical parallelism. Using hipSYCL with GCC 10 instead, shown in Figure 2b, both scoped and hierarchical parallelism vectorise well, with scoped parallelism outperforming hierarchical parallelism. We therefore conclude that the performance of scoped parallelism in this experiment is not an inherent limitation of the model, but a limitation in Clang 11 (we observed similar behaviour with Clang 12). Consequently, similar performance for hierarchical and scoped parallelism is possible if different compilers are used.

B. SYCL-Bench

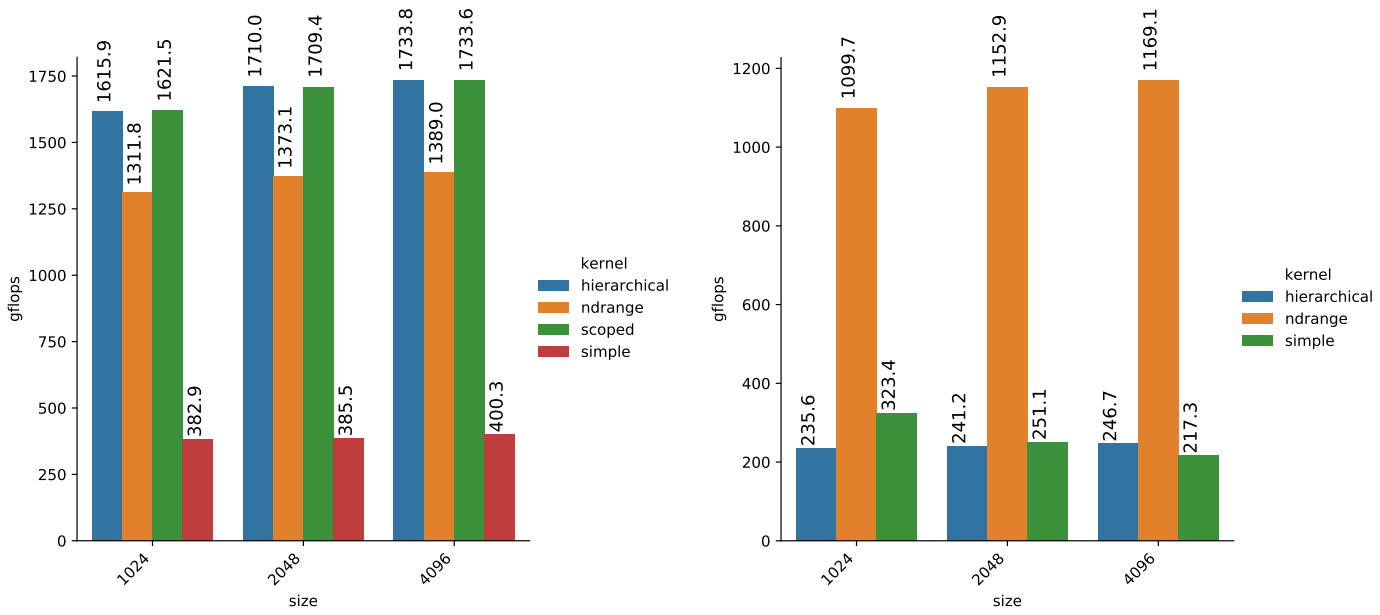
Tables II and III show the runtime results of the three SYCL-Bench benchmarks on the V100 GPU and Cascade Lake CPU respectively. Note that these figures report benchmark runtime, and so lower indicates faster runtime.

For all benchmarks, the V100 shows little difference in runtime for all the parallelism implementations with hipSYCL. This shows that the additional abstractions introduced by hierarchical and scoped parallelism don't necessarily introduce performance overhead compared to the `nd_range` model, which is the most straight-forward to map to GPUs.

As with the DGEMM benchmark, the hierarchical parallelism implementation in DPCPP is again slower on both our CPU and GPU systems than the NDRange implementation.

On the CPU however, there are significant differences for the reduction benchmarks as shown in Table III. Both hierarchical and scoped parallelism have similar runtimes and provide a significant improvement over the NDRange implementation. This is because for NDRange parallelism, hipSYCL's library-only CPU backend relies on multi-threading across work-groups, and, if barriers are present, executes each work-item in its own fiber. A fiber is a light-weight userland thread-like object that has its own stack, but is not subject to preemption. Instead, when a barrier is encountered, the active fiber yields control to the fiber scheduler used by hipSYCL,

[3]https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html



(a) HipSYCL

(b) DPCPP

Fig. 1. DGEMM benchmark results on a NVIDIA V100 GPU.

which then cycles through all fibers, switching the active stack each time, until all fibers have arrived at the barrier.

For a parallel reduction algorithm, only very few instructions (e.g. additions) are executed before a barrier must be invoked to make the changes visible to the other work-items. This means that the overhead of such a code for the context switches between the fibers is going to dominate the overall runtime, severely limiting performance. In order to run efficiently in such a scenario, the amount of work per work-item must be increased, such as by performing many additions of the reductions in the private memory of each work-item before exchanging information with other work-items — however, this is not how common `nd_range` reductions are written, since this contradicts the fine-grained parallelism model typically found on GPUs.

For the nbody benchmark on the CPU, the differences between the models are less significant in hipSYCL. This is because the nbody benchmark is highly compute-bound, with a high amount of work per work-item and few barriers. Performance is therefore mostly dependent on how well the compiler auto-vectorizes a particular formulation of the kernel. In addition, nbody is the only benchmark in this study that uses `private_memory` allocations in the hierarchical and scoped versions. At present, this is implemented with a dynamic memory allocation on the heap for every work group to allocate sufficient memory for each work item, which is avoidable in the DPCPP compiler-based implementation of SYCL.

VII. Conclusion

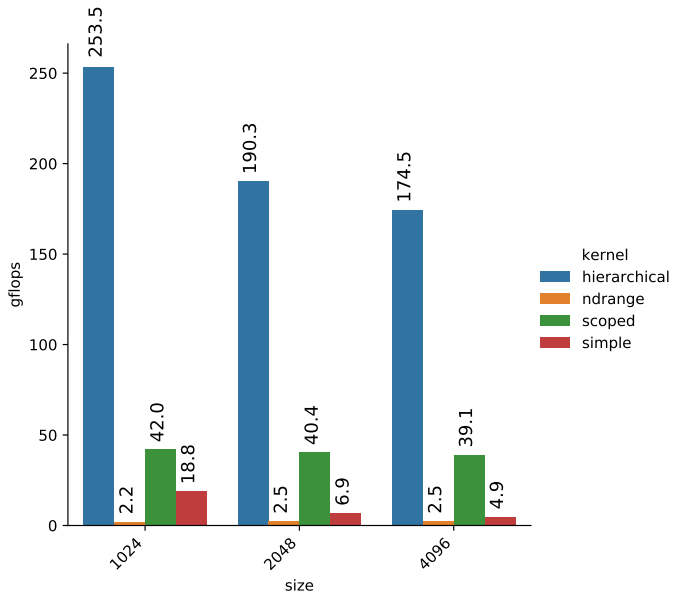
In this paper we have explored hierarchical parallelism in SYCL. Use of hierarchical parallelism or NDRange parallelism has implications for performance, even though they

TABLE II
SYCL-BENCH results on a NVIDIA V100 GPU.

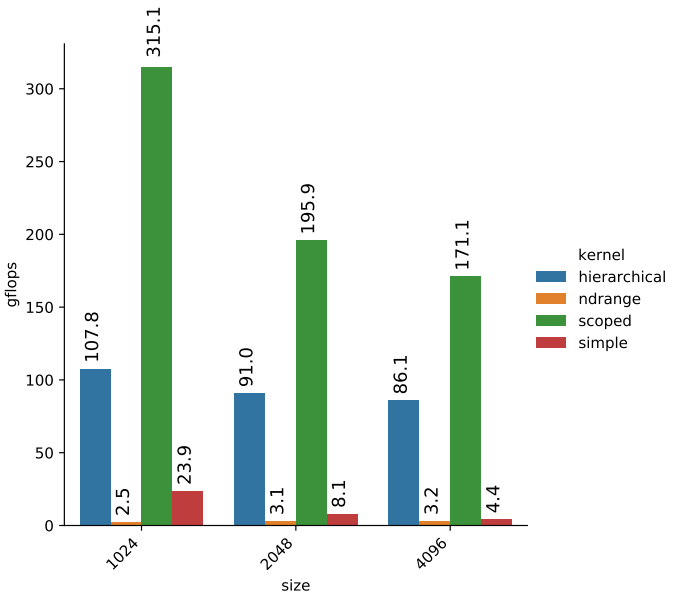
Benchmark	**Kernel**	**Runtime** (s) hipSYCL	DPCPP
nbody	ndrange	0.185	0.162
	hierarchical	0.187	0.311
	scoped	0.187	-
reduction	ndrange	0.002	Verification failed.
	hierarchical	0.002	Verification failed.
	scoped	0.002	-
segmented reduction	ndrange	0.002	0.002
	hierarchical	0.002	0.022
	scoped	0.002	-

TABLE III
SYCL-BENCH results on 2x20-core Intel Cascade Lake CPU.

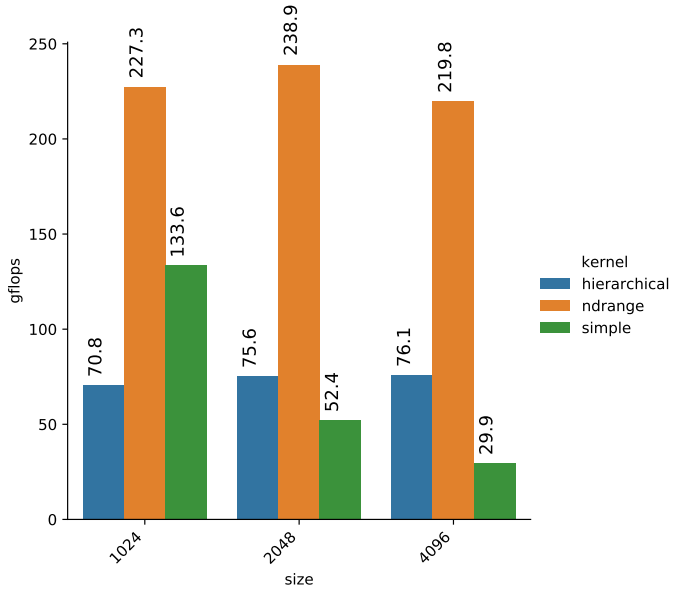
Benchmark	**Kernel**	**Runtime** (s) hipSYCL		DPCPP
		LLVM	GCC	
nbody	ndrange	4.053	3.478	0.700
	hierarchical	3.323	2.966	4.952
	scoped	5.665	4.250	-
reduction	ndrange	6.543	4.430	0.022
	hierarchical	0.033	0.044	0.457
	scoped	0.033	0.029	-
segmented reduction	ndrange	6.438	4.408	0.020
	hierarchical	0.037	0.036	0.207
	scoped	0.035	0.028	-



(a) HipSYCL (LLVM)



(b) HipSYCL (GCC)



(c) DPCPP

Fig. 2. DGEMM benchmark on 2x20-core Intel Cascade Lake CPU.

provide similar semantics in the SYCL standard. Implementing hierarchical parallelism alleviates some of the implementation challenges of implementing SYCL in a library-only implementation that's portable across CPUs and GPUs. We plan to extend our work to explore compiler-based GPU implementations of SYCL hierarchical parallelism. Our results show these hierarchical parallel approaches are not necessarily just syntactic equivalents to NDRange parallelism in SYCL.

For the DGEMM and Reduction kernels, using hierarchical parallelism in favour of NDRange parallelism provided significant performance benefits on both CPUs and GPUs when using hipSYCL. DPCPP performed well on CPUs with standard NDRange kernel, since as a compiler-based implementation, it can apply transformations to the code to map it well to the hardware. DPCPP's hierarchical parallelism implementation did not provide high performance in comparison.

We have also investigated the Scoped Parallelism extension to SYCL. For our benchmarks, these should result in similar mappings, and similar performance, to hierarchical parallelism, however vectorising compiler limitations yield some performance variations. Scoped parallelism provides additional flexibility over hierarchical parallelism for both

programmer and implementer. Programmers can use similar APIs for multiple levels, and SYCL implementers are free to map the parallel levels to hardware as appropriate.

Indeed, it would be possible to implement much of the scoped parallelism extension as a library on top of NDRange in general, and we hope to explore this in future work.

ACKNOWLEDGMENT

This work was in part funded by EPSRC through the Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV) project, grant number: EP/S005072/1.

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

REFERENCES

[1] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking Performance Portability on the Yellow Brick Road to Exascale," *Proceedings of P3HPC 2020: International Workshop on Performance, Portability, and Productivity in HPC, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2020.

[2] The Khronos SYCL Working Group, *SYCL 2020 Specification*, Mar. 2020.

[3] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388658

[4] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing," in *Euro-Par 2020: 26th International European Conference on Parallel and Distributed Computing*, ser. Euro-Par '20. Springer International Publishing, 2020.

[5] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The Ongoing Evolution of OpenMP," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8434208/

[6] C. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. A. Ibanez, N. Liber, J. Madsen, J. S. Miles, D. Z. Poliakoff, A. J. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9485033/

[7] J. R. Hammond, M. Kinsner, and J. Brodman, "A Comparative Analysis of Kokkos and SYCL as Heterogeneous, Parallel Programming Models for C++ Applications," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3318170.3318193

[8] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388669