Dynamic Memory Allocations in a Parallel Context

Aymeric Millan, Thomas Padioleau, Julien Bigot

Dynamic Memory Allocations in a Hierarchical Parallel Context

A study on performance, memory footprint, and productivity using SYCL

Aymeric MILLAN, Thomas PADIOLEAU and Julien BIGOT

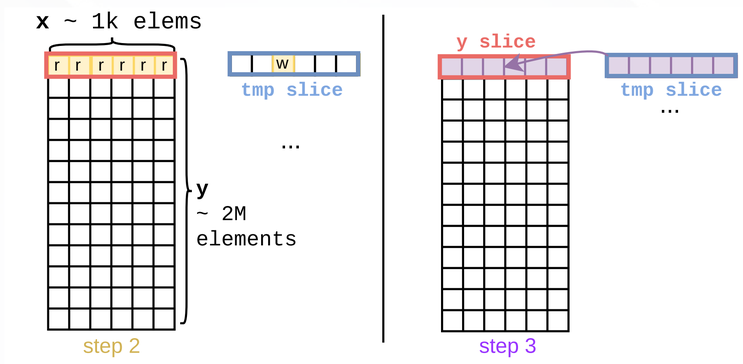
Université Paris-Saclay, UVSQ, CNRS, CEA, Maison de la Simulation, 91191, Gif-sur-Yvette, France

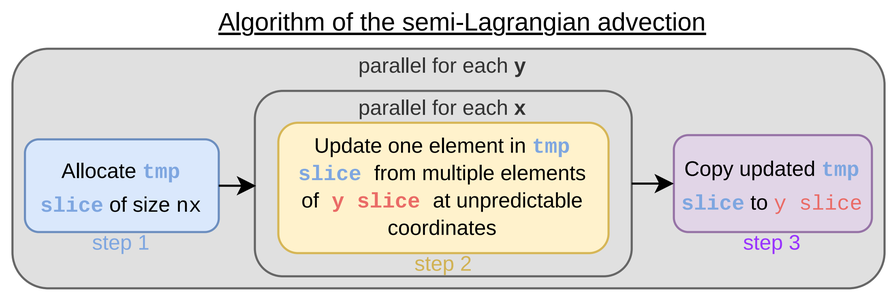
{aymeric.millan, thomas.padioleau, julien.bigot}@cea.fr

Introduction

This poster investigates the challenges of dynamic memory allocations in a hierarchical parallel context using the SYCL 2020[1] programming model and extensions. The study explores memory management using different parallelism paradigms. We study the **performance**, **productivity** and **memory footprint** of each implementation. Dynamic allocations in nested parallel *for* loops can be found in many situations, such as the semi-Lagragian advection scheme in the GYSELA[2] plasma turbulence simulation code.

- **Application:** 1D advection inside a multi-dimensionnal space:
  - x: advected dimension
  - y: independant advection problems
- **Sizes** representative of the GYSELA use-case
  - n_x: size of one advection problem (1.10^3)
  - n_y: number of advection problems in 1 node (2.10^6)





Minimal theoretical memory footprint

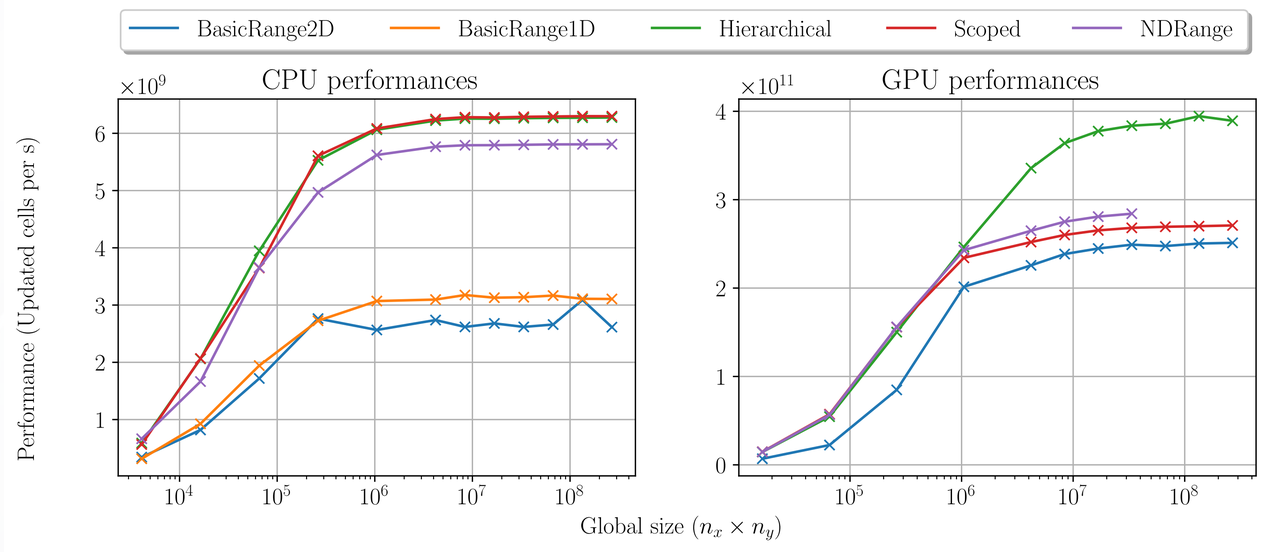
Sequential	Along x	Along x and y	Along y	Out-of-place
$f = 8\,p_y\,n_x$ and $p = p_y\,p_x$ (e.g., $p_{GPU} = 10^5$ and $p_{CPU} = 64$, $n_x = 10^3, n_y = 2.10^6$)				
$p = p_y = 1$ $f = 8\,n_x$ $\approx$ **8 kB**	$p = p_x, p_y = 1$ $f = 8\,n_x$ $\approx$ **8 kB**	$p_x = n_x$, $p = p_y\,n_x$ $f = 8p$ CPU $\approx$ **512 B** GPU $\approx$ **800 kB**	$p = p_y$ $f = 8\,p\,n_x$ CPU $\approx$ **512 kB** GPU $\approx$ **800 MB**	$f = 8\,n_x\,n_y$ $\approx$ **16 GB**
L1 •	L1	L2 ●●●	L3 CPU/RAM GPU	RAM ●●

The minimal theoretical memory footprint (f) depends on the parallelism strategy, and on p, the degree of parallelism, p_x and p_y the parallelism along x and y, respectively. One buffer of n_x `double` elements of 8 bytes is allocated for each of the p_y iterations over y executed concurrently. To get a sense of the orders of magnitude, we use $p_{CPU} = 64$ corresponding to the number of cores on a typical server CPU and $p_{GPU} = 10^5$, the number of active threads typically required to fill a GPU. We identify the level of cache where our GYSELA use-case would fit using each strategy on an NVIDIA A100 GPU (192 KB L1, 40 MB L2, and 40 GB RAM).

Evaluation

	Perf.	Mem. footprint	Productivity feeling
Seq. •	−−	++	Simple
BR2D •	−	−−	Simple
BR1D •	−	−−	Inner seq. loop in parallel_for
NDRa. •	+	+	Work groups + global size must be divisible by local size
Hier. •	++	+	Work groups + mapping logical-physical iteration spaces
Sco. •	+	+	Work groups + nested parallel loops

The **memory footprint** recalls the results of the block above. The **performance** is detailed in the plots below. The **productivity** is a personal feeling based on the code detailed in the block on the right. The experiments ran on two different hardware architectures: Intel Xeon Gold 6230 for CPU, and NVIDIA A100 40GB for GPU. We used hipSYCL 0.9.4 accelerated CPU and CUDA backends.



Implementations

Each implementation provides a different set of tools. More tools give more control over the optimizations but make the implementation harder, decreasing the **productivity** factor.

- Control over *work groups* (`wg`) **local** sizes (`lsize`)
- `local_accessor`, shared memory within the same work-group
- A mapping between a ***physical iteration*** space (`psize`) and a ***logical iteration space*** (`=lsize`)

• Sequential

Using C++ `for` loops. `tmp slice` is allocated only once and reused for each independant 1D advection problem in y

```
double* data = allocate(ny*nx);
double* tmp  = allocate(nx);
for (int i = 0; i < ny; i++)
    for (int j = 0; j < nx; j++)
        //tmp[j] = ...
    //data[i, :] = tmp
```

• NDRange

By specifying a *global* iteration space (the size of the global buffer) divided into *local* iteration spaces (i.e. the size of a work group). The number of work groups is deduced by the SYCL implementation.

```
range<2> gsize(ny, nx);
range<2> lsize(1, nx);
//nb wg deduced by SYCL impl ...
local_accessor tmp(range<1>(nx));
parallel_for(
    nd_range<2>(gsize, lsize),
    [=](sycl::nd_item<2> nd_itm){
/* ix=nd_itm.get_local_id(1);
    tmp[ix] = data...
    group_barrier(itm.group);
    data[iy,ix]=tmp[ix]; */});
```

• Hierarchical

By specifying the *number* of work groups and their *local sizes*. In addition, these kernels allow a finer optimization by providing a control on *physical* and *logical* iteration spaces sizes.

```
range<1> nb_wg(ny);
range<1> lsize(nx);
local_accessor tmp(lsize);
//hw related optimizations
range<1> psize(HW_SIZE);
parallel_for_work_group(
    nb_wg, psize,
    [=](group<1> g){
    g.parallel_for_work_item(
        lsize,
        [&](h_item<1> it) {
  /* parallel algo */});});
```

BasicRange kernels are implemented using a classical `parallel_for` mapped onto a `sycl::range`. These kernels are out-of-place and require a second `buffer` that is the same size of the global buffer.

• BasicRange2D

Using a range corresponding to the global buffer range `range<2>(ny, nx)`, thus accessing the values with a unique 2D `sycl::id`.

```
range<2> r2d(ny, nx);
buffer data(r2d);
buffer tmp(r2d);
q.parallel_for(r2d,
    [=](id<2> itm){
  /*tmp[itm] = ...*/});
//copy(tmp, data);
```

• BasicRange1D

Using a 1D range along the y dimension `range<1>(ny)` and sequentially iterating on x.

```
range<1> r1d(ny);
buffer data(r2d);
buffer tmp(r2d);
q.parallel_for(r1d,
    [=](id<1> i){
  for(int j=0; j<nx; j++)
    /*tmp[i][j]= ...*/});
//copy(tmp, data);
```

• Scoped

is an hipSYCL extension to SYCL's hierarchical parallelism[3]. This type of kernel is primarily oriented towards compile-time allocations rather than dynamic allocations.

```
range<1> nb_wg(ny);
range<1> lsize(nx);
local_accessor tmp(lsize);//...
parallel(
    nb_wg, lsize,
    [=](auto g) {
//distributed_groups(...
    distribute_items(g,
        [&](s_item<1> it){
  /* parallel algo */});});
```

Conclusion

It is not trivial to be able to finely control memory allocations in a hierarchical parallel context while guaranteeing code portability and performance. BasicRange kernels, the default and simplest way to write parallel loops in SYCL does not allow managing work-groups nor writing hierarchical nested for each loops, resulting in a poor performance as well as a large memory footprint in our experiments. This emphasizes the need of other ways to express parallelism, such as the Hierarchical or Scoped kernels, which seem to be the most suited for our needs although we loose the ease-of-use of BR kernels.

References

[1] "SYCL 2020 Specification (revision 6)". In: (2020).

[2] V. Grandgirard et al. "A 5D gyrokinetic full- f global semi-Lagrangian code for flux-driven ion turbulence simulations". In: (2016).

[3] Tom Deakin et al. "Benchmarking and Extending SYCL Hierarchical Parallelism". In: 2021.

[4] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: (2022).

[5] Aksel Alpay et al. "Exploring the possibility of a hipSYCL-based implementation of oneAPI". In: 2022.

[6] Philip Salzmann et al. "Celerity: How (Well) Does the SYCL API Translate to Distributed Clusters?" In: 2022.

[7] S.J. Pennycook et al. "Implications of a metric for performance portability". In: (2019).