Imperial College London

Department of AERONAUTICS

GPU Parallelisation of a 2D Navier-Stokes Solver

Nathanael Jenkins

CID 01843784

Abstract

While heterogeneous computing has the potential to dramatically improve the efficiency of some CFD solvers, it is difficult to successfully implement even in the most simple of cases. Competing hardware, frameworks, and compilers complicate this process, and many current solutions are still in their infancy. This report finds that the SYCL 2020 framework has a great deal of potential, but it is not yet superior to alternative means of parallelisation. Future revisions of SYCL are likely to resolve existing issues, and could make the framework (and heterogeneous computing) a much more useful tool.

August 13, 2021

Supervisor: Sylvain Laizet

Contents

List of Figures

1 Introduction

Heterogeneous computing is an alternative to traditional high-performance computing (HPC), making use of the specialised nature of GPUs and other hardware devices to compliment CPUs. GPUs are designed for calculations across large matrices, so they are theoretically ideal for fluid dynamics solvers. While traditional HPC primarily relies on CPU parallelism, there is growing interest in GPU computing to improve performance [1].

Frameworks such as OpenCL, CUDA and OpenACC support offloading to GPU devices, but these can be difficult to implement and portability can vary [2]. The SYCL framework claims to "deliver portable performance" [3] through "single-source heterogeneous programming for acceleration offload" [4]. It aims to allow developers to write code capable of running on the widest possible range of heterogeneous systems, whereas frameworks such as CUDA support only specific manufacturers or device types.

SYCL 2020 is a relatively young framework developed by Khronos Group [4], derived from a specification first introduced in 2014 [5]. Several compilers support SYCL for various hardware types, including popular open-source compilers hipSYCL [6] and triSYCL [7], or Intel's proprietary compiler DPC++, which is a part of their oneAPI toolkit [8].

In broad terms, the aim of this report is to investigate the performance and portability of GPU offloading using the SYCL framework. Both compute and memory performance will be explored, and portability will be tested across a range of hardware manufacturers, types, and operating systems. In particular, performance of a 2D finite difference Navier-Stokes solver will be tested with SYCL acceleration. The solver uses a second-order differencing method with the Adams-Bashforth temporal scheme and periodic boundary conditions to compute flow over a heat-exchanger.

2 Theory and Background

SYCL is a C++ framework for heterogeneous computing, with several current implementations to support the relatively wide range of hardware described in figure 1[4]. Most compilers including DPC++ and triSYCL generate GPU code using OpenCL, whereas the hipSYCL compiler uses manufacturer-specific approaches in CUDA, ROCm, and Level Zero [6]. hipSYCL and triSYCL interact with CPUs in parallel using OpenMP, whereas DPC++ uses 'host' [6]. SYCL is designed to support any hardware built on OpenCL 1.2 or greater as well as all CPUs [4].

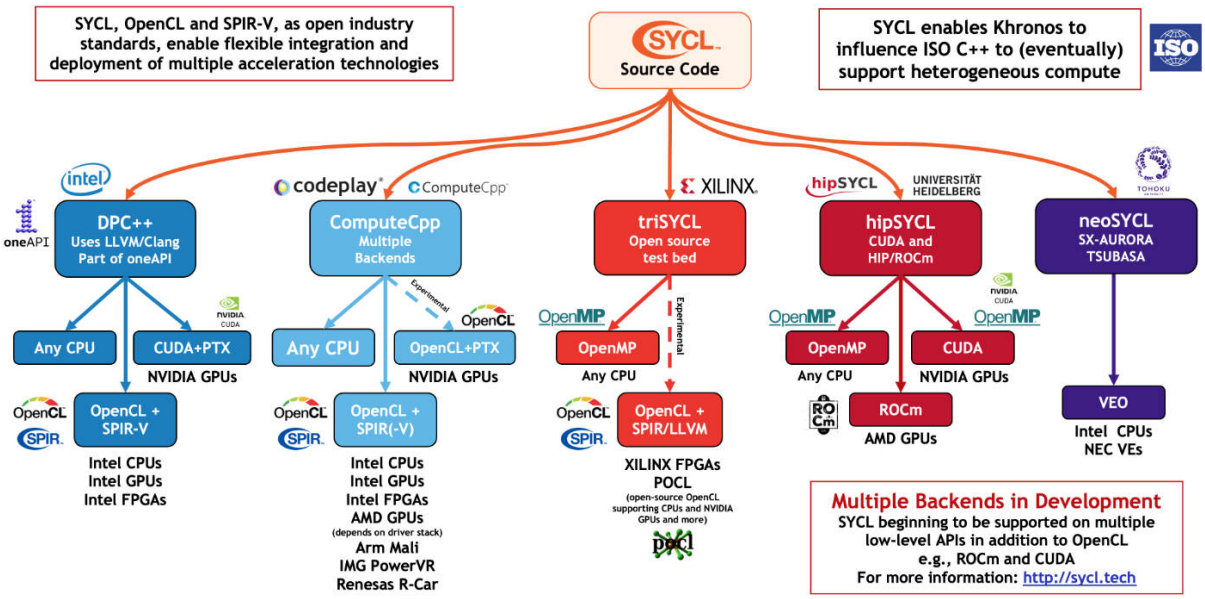


Figure 1: SYCL Implementations

Amdhal's Law describes the maximum increase in speed of a parallel application based on the percentage of the program which runs in parallel. Equation 1 [9] calculates the maximum speed-up of a parallel program (Δ), where p is the proportion of the program running in parallel. In the case of the

2D Navier-Stokes solver, its main time loop is entirely parallel, with the exception of printing field averages for each timestep, although the time taken to compute and display this is small. Therefore, the theoretical maximum speed-up for this program tends to more than 200 times, although in practise the number of parallel processors and memory bandwidth will limit this to a much lower value.

$$\Delta = \frac{1}{1-p} \tag{1}$$

3 CPU Parallelisation of a 2D Solver

The short finite-difference solver, written in Fortran 90, was initially only capable of serial CPU execution. The program was 'embarassingly parallel' [10] and was expected to run significantly faster when multiple threads were launched at once. A 'dependency tree' was produced, shown in Appendix A, which highlights which aspects of the code are parallelisable, and those which are dependent on one another. All operations inside of the time loop are performed on large arrays, so they are suitable for parallelisation.

3.1 Serial C++

To support SYCL, a C/C++ framework, the code was first translated from Fortran 90 into C++. This was relatively easy, although several important changes were necessary to achieve similar performance to the Fortran code. Figure 2 shows one function as represented in C++ and Fortran; testing found that the Fortran solver ran, on average, 2.8 times faster than this version of the C++ solver. This is partly because the C++ code initially used multidimensional arrays which are stored in stack memory. This caused several problems in testing, including segmentation faults for larger domain sizes unless stack limits were manually increased.

```
!First derivative in the x direction
!########################################
subroutine derix(phi,nx,ny,dfi,xlx)
  implicit none

  real(8),dimension(nx,ny) :: phi,dfi
  real(8) :: dlx,xlx,udx
  integer :: i,j,nx,ny

  dlx=xlx/nx
  udx=1./(dlx+dlx)
  do j=1,ny
     dfi(1,j)=udx*(phi(2,j)-phi(nx,j))
     do i=2,nx-1
        dfi(i,j)=udx*(phi(i+1,j)-phi(i-1,j))
     enddo
     dfi(nx,j)=udx*(phi(1,j)-phi(nx-1,j))
  enddo
  return
end subroutine derix
```

```
//=======================================================
//  First derivative in x-direction
void derix(double phi[nx][ny], double dfi[nx][ny], double &xlx){

    double udx=nx/(2*xlx);

    for(int j=0; j<ny; j++){
        dfi[0][j]=udx*(phi[1][j]-phi[nx-1][j]);
        for(int i=1; i<nx-1; i++){
            dfi[i][j]=udx*(phi[i+1][j]-phi[i-1][j]);
        }
        dfi[nx-1][j]=udx*(phi[0][j]-phi[nx-2][j]);
    }

    return;
}
```

Figure 2: Comparison of function 'derix' in Fortran90 and C++(17) respectively

3.1.1 Compiler choice

An investigation into compiler choice was first conducted on serial C++ and Fortran code. It found that the Intel ifort compiler produced the fastest programs based on Fortran code, which consistently outperformed any other Fortran or C++ tests. A summary of these results is shown in figure 3, which profiled performance for a domain of 248x248 over 10,000 timesteps. It is clear that the proprietary Intel fortran compiler (ifort) is superior to GNU fortran, and these results also demonstrate how the C++ code using stack memory performs slower than Fortran. Interestingly, the Intel C++ compiler performs no better than G++ with optimisation flags.

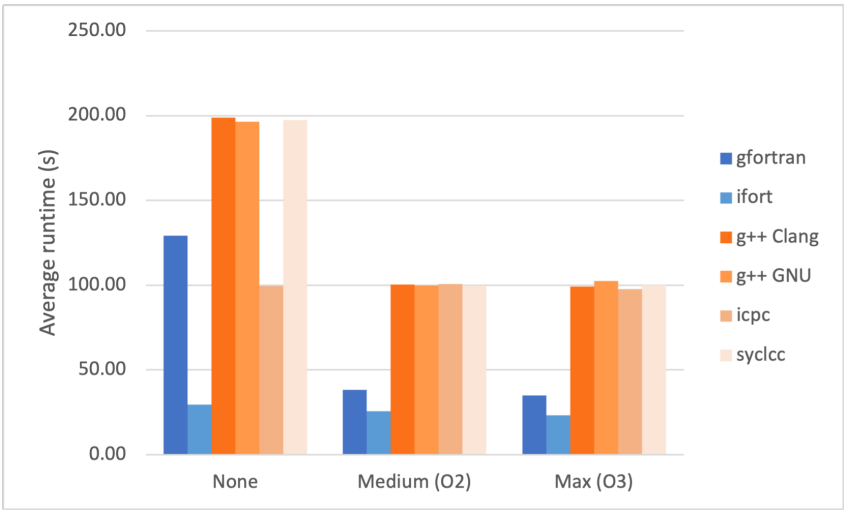


Figure 3: Effect of compiler and language on runtime

3.1.2 Memory allocation

Using std::malloc, arrays were replaced with vectors in heap memory. In serial, this code was still found to be 1.2 times slower than its equivalent Fortran solver when compiled using GNU G++ and Gfortran with -O3 optimisations, but at least equally as fast when compiled without optimisations. A summary of these results is shown in figure 1, which profiled performance for a domain of 200x200 over 10,000 timesteps.

Code	Compiler	Optimisations	Average runtime (s)
Fortran	gfortran	-	7.95
C++ (Stack)	g++	-	10.67
C++ (Heap)	g++	-	7.76
Fortran	gfortran	-O3	1.84
C++ (Stack)	g++	-O3	4.16
C++ (Heap)	g++	-O3	2.26

Table 1: Effect of optimisations and language on runtime for serial code

Different compiler optimisations affect this code in different ways, and it is clear in this case that the GNU fortran optimisations were more effective than equivalent optimisations in G++. While this report won't discuss the differences between serial code in any greater detail, it is important to note the effect of language, compiler, and optimisations on runtime.

3.2 Elementary Parallelisation

Parallelising the solver to run on multiple CPUs was easy using OpenMP in Fortran and relatively simple using SYCL in C++. The 'derix' function is shown again in figure 4 with parallelism. Note how the OpenMP directives can be ignored by the compiler if desired, whereas the SYCL code must use a SYCL compiler unless preprocessor directives are used to select between serial and parallel versions of the function.

```fortran
!=============================================
!  First derivative in the x direction
subroutine derix(phi,nx,ny,dfi,xlx)

  implicit none

  real(8),dimension(nx,ny) :: phi,dfi
  real(8) :: dlx,xlx,udx
  integer :: i,j,nx,ny

  dlx=xlx/nx
  udx=1./(dlx+dlx)
  !$OMP PARALLEL DO
  do j=1,ny
     dfi(1,j)=udx*(phi(2,j)-phi(nx,j))
     do i=2,nx-1
        dfi(i,j)=udx*(phi(i+1,j)-phi(i-1,j))
     enddo
     dfi(nx,j)=udx*(phi(1,j)-phi(nx-1,j))
  enddo
  !$OMP END PARALLEL DO
  return
end subroutine derix
```

```cpp
//=========================================================
//  First derivative in x-direction
void derix(double phi[nx][ny], double dfi[nx][ny], double &xlx){

    double udx=nx/(2*xlx);

    cl::sycl::queue q;
    q.parallel_for(cl::sycl::range{ ny, nx-2 }, [=](cl::sycl::id<2> idx){
        int i = idx[1]+1;
        int j = idx[0];
        dfi[i][j]=udx*(phi[i+1][j]-phi[i-1][j]);
    });
    q.parallel_for(cl::sycl::range{ ny }, [=](cl::sycl::id<1> idx){
        int j = idx[0];
        dfi[0][j]=udx*(phi[1][j]-phi[nx-1][j]);
        dfi[nx-1][j]=udx*(phi[0][j]-phi[nx-2][j]);
    });
    q.wait();

    return;
}
```

Figure 4: CPU Parallel code in Fortran 90 (OpenMP) and C++ (SYCL)

The implementation of SYCL in figure 4 is very crude, using a local sycl::queue construct with implicit data management. While this worked reasonably well for CPU parallelism, it is not ideal for GPU offloading as discussed in section 4.

3.2.1 Domain size

Firstly, testing was conducted to explore the effect of increasing the computational domain, since it is expected that parallelism performs best for large computational domains. Figure 5 illustrates the speed increase of parallel code when compared to serial code of the same domain size for domains between 50x50 and 2000x2000. These results suggest that, for these versions of the code, a domain between 500x500 and 750x750 is necessary to experience an increase in speed relative to serial code. For small domains, parallelisation inhibits performance due to increased overhead.

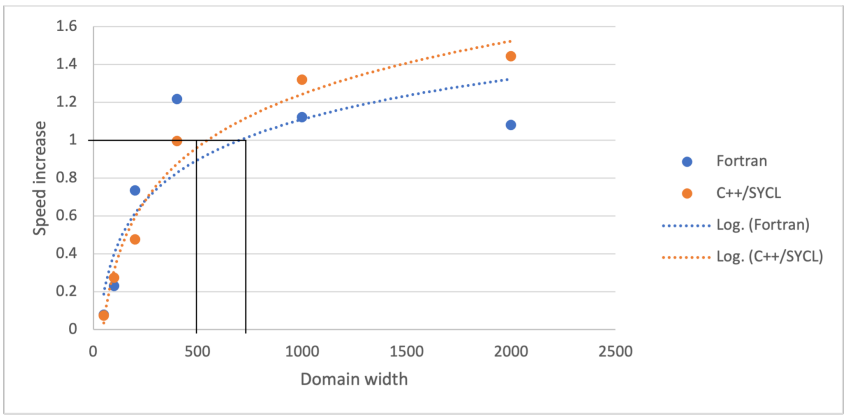


Figure 5: Effect of domain width on speed increase due to CPU parallelisation

3.2.2 Thread count

It was expected that speed would increase with the number of available computational processors, particularly for large domains. However, initial results peaked at 6 threads before decreasing beyond that. This could have been caused by optimisation flags becoming less effective with increased parallelism, since most optimisations aimed to improve the efficiency of loops which were instead run in parallel. Overhead increases with the number of threads so for the first test, using a 1000x1000 domain, this may have also influenced results. Additionally, this test took place on a local machine with 6 cores, but 12 processors using hyperthreading. The reduction in speed could be due to issues when hyperthreading.

The test was repeated for a 2000x2000 domain over 200 timesteps, this time generating reasonable results shown in figure 6. Using Amdhal's law, it was possible to prove using this data that the SYCL

implementation was running at least 33% of the program in parallel, and OpenMP Fortran was executing more than 64% in parallel. This crude implementation of SYCL is not particularly effective, particularly when compared to the OpenMP implementation.

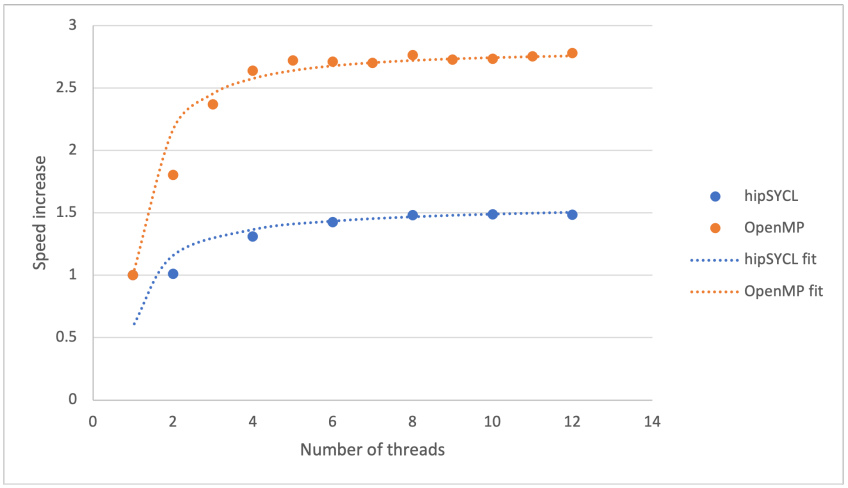


Figure 6: Speed increase due to CPU parallelisation with respect to the number of threads

It is important to note that, while OpenMP allows for an explicit number of threads to be launched, SYCL will simply use the maximum available number of processors, so limiting thread number was achieved by manually limiting the number of active processors on the machine. There is certainly more scope for investigation of CPU parallelisation using SYCL and OpenMP, but these findings fundamentally demonstrate that OpenMP is easier to implement and faster than a simple SYCL implementation of CPU parallelism.

4 GPU Parallelisation of a 2D Solver

4.1 Buffers

At first, an attempt to parallelise the program was made using the multidimensional array (stack-based) version of the C++ code. CPU parallel code launched separate queues inside each derivative function, and in the fluxx function, using lambdas to launch parallel_for kernels for each loop. This used implicit data management and q.wait() calls, giving the compiler freedom to manage offloading, as shown in figure 4.

To successfully offload to the GPU, an explicit data management scheme was implemented using buffers and accessors, as shown in figure 7. A single queue was created in the fluxx subroutine, which was shared between derivatives, with q.wait() calls moving out of the derivative functions and into fluxx, reducing the total number of calls. While the reduction in wait time may have had some positive impact, the creation of so many buffers and accessors was very slow, resulting in extremely long run times and a significant reduction in speed. In particular, the use of reinterpret_cast buffers likely resulted in excessive data movement.

```
//=========================================================
//  First derivative in x-direction
void derix(double phi[nx][ny], double dfi[nx][ny], double &xlx, cl::sycl::queue q){

    double udx=nx/(2*xlx);

    cl::sycl::buffer phi_buf(reinterpret_cast<double *>(phi), cl::sycl::range(nx, ny));
    cl::sycl::buffer dfi_buf(reinterpret_cast<double *>(dfi), cl::sycl::range(nx, ny));

    q.submit([&](auto &h1) {
        cl::sycl::accessor a_phi(phi_buf, h1, cl::sycl::read_only);
        cl::sycl::accessor a_dfi(dfi_buf, h1, cl::sycl::read_write);
        h1.parallel_for(cl::sycl::range(ny, nx-2), [=](auto idx) {
            int i = idx[1]+1;
            int j = idx[0];
            a_dfi[i][j]=a_phi[i+1][j]-a_phi[i-1][j];
            a_dfi[i][j]=a_dfi[i][j] * udx;
        });
    });
    q.submit([&](auto &h2) {
        cl::sycl::accessor a_phi(phi_buf, h2, cl::sycl::read_only);
        cl::sycl::accessor a_dfi(dfi_buf, h2, cl::sycl::read_write);
        h2.parallel_for(cl::sycl::range(ny, 2), [=](auto idx){
            int j = idx[0];
            a_dfi[0][j]=a_phi[1][j]-a_phi[nx-1][j];
            a_dfi[nx-1][j]=a_phi[0][j]-a_phi[nx-2][j];
            a_dfi[0][j]=a_dfi[0][j]*udx;
            a_dfi[nx-1][j]=a_dfi[nx-1][j]*udx;
        });
    });
    return;
}
```

Figure 7: GPU offloaded SYCL implementation using buffers and accessors

Figure 8 shows two parts of the Intel VTune time profile for the buffer-based solver over 10 timesteps for a 129x129 domain. The upper line illustrates thread activity, with each green ZeM block representing a oneAPI Level Zero module creation launched at sycl::queue initialisation. The large gaps between these show the slow data movement taking place between each iteration, which is illustrated in black. These data transfers are one of the reasons this version of the program is so slow.

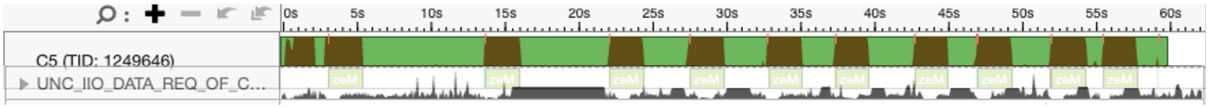


Figure 8: VTune profile for buffer-based GPU offloading over 10 timesteps

4.2 Implicit USM

To resolve the incredibly slow buffer-based code, an alternative memory handling approach was explored with the use of Unified Shared Memory (USM). Starting from the serial heap-based C++ solver, implicit USM was implemented, with each array initialised using sycl::malloc_shared. This allocates data to host or device memory, with freedom for movement between the two as needed [10]. A sample of this version of the program is shown in figure 9.

```
//============================================================
//  First derivative in x-direction
void derix(double *phi, double *dfi, double &xlx){

    double udx=nx/(2*xlx);

    q.submit([&](auto &h) {
        h.parallel_for(cl::sycl::range(ny, nx-2), [=](auto idx) {
            int i = idx[1]+1;
            int j = idx[0];
            dfi[i+nx*j]=phi[nx*j+i+1]-phi[nx*j+i-1];
            dfi[i+nx*j]=dfi[i+nx*j] * udx;
        });
    });
    q.submit([&](auto &g) {
        g.parallel_for(cl::sycl::range(ny), [=](auto idx) {
            int j = idx[0];
            dfi[nx*j]=udx*(phi[nx*j+1]-phi[nx*(j+1)-1]);
            dfi[nx*(j+1)-1]=udx*(phi[nx*j]-phi[nx*(j+1)-2]);
        });
    });

    return;
}
```

Figure 9: GPU offloaded SYCL implementation using USM

This was the first successful GPU offloading method, with initial testing demonstrating a speed increase of more than 10x for large domains compared to serial performance. Further testing using Intel devcloud validated this, as shown in figure 10. This test compared the SYCL-based solver with the original serial C++ implementation (heap-based) using the DPC++ compiler. Extrapolation suggests that a speed increase of more than 25x could be possible using this code. It is important to remember that this is a comparison with serial code compiled using the DPC++compiler without any optimisation flags. Use of a standard C++ compiler such as icpc with optimisations is likely to reduce the apparent speed increase to a value of 3-5 times.

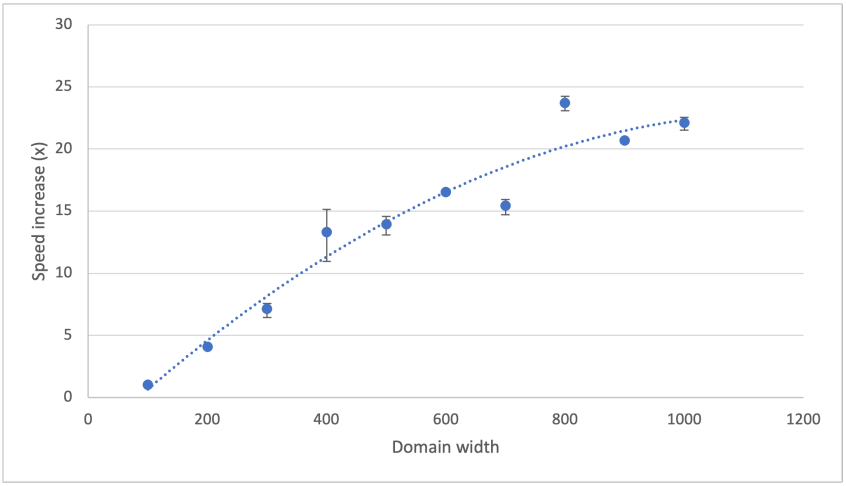


Figure 10: GPU offload speed increase using implicit USM

While GPU offloading is clearly effective in this case when compared to serial code, testing suggested that this SYCL implementation is no better than standard CPU parallelisation. Figure 11 illustrates how, at best, GPU offloading achieves almost identical performance to CPU parallelisation using the same SYCL code. Device choice was controlled using the sycl::device_selector class.

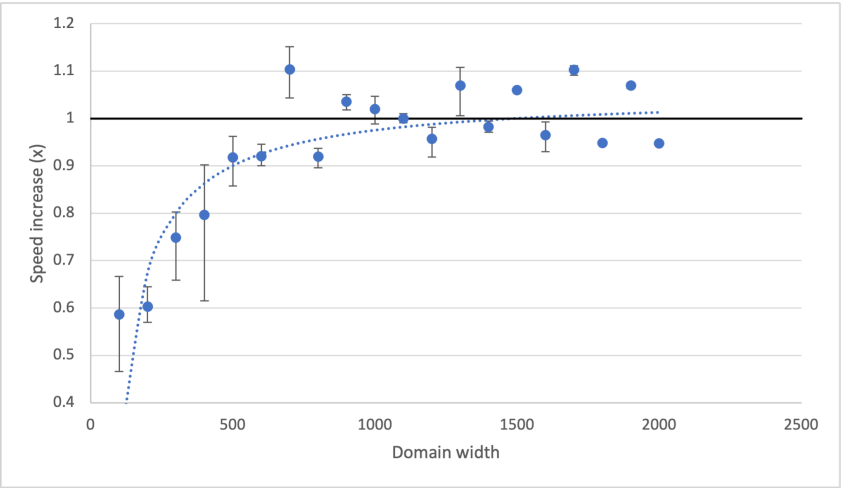


Figure 11: GPU parallel speed increase using implicit USM compared to CPU parallel

4.3 Explicit USM

While implicit USM is easy to implement, it does not give users control over data storage so it is likely to perform worse than an explicit approach. Using sycl::malloc_device and sycl::malloc_host allocations, it is possible to force the program to store certain data in preferred locations. Initially, a 'semi-explicit' approach was implemented where all arrays were stored on device memory, except for uuu, vvv, and scp which were needed for average calculations which used the host. These arrays, along with some constants, were allocated to shared memory. The functions did not need any changes for this implementation, so figure 9 ('derix') remained the same for the semi-explicit code.

Runtime was not significantly improved with this implementation, although marginally less data movement was measured between the GPU and the host. Figure 12 shows a Vtune profile of this program; note the relatively significant idle time at the start of the program spent initialising and offloading data to the GPU.

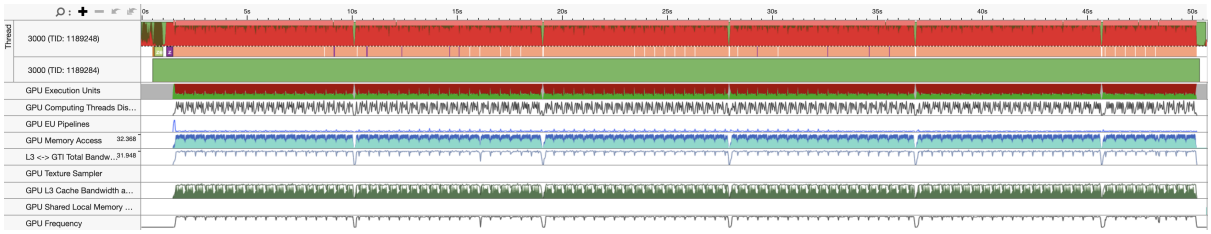


Figure 12: Vtune profile for 3000x3000 domain over 100 timesteps using semi-explicit USM

A fully explicit USM implementation was made difficult by the use of field averages for monitoring. The sum (reduction) of an array is not as easy to parallelise as conventional loops, although it is possible to achieve through the use of the sycl::reduction class. Explicit USM was found to perform marginally better than implicit USM, particularly for tests with more time steps. It is likely that the implicit approach offloads most of the data in the same way as the explicit code, but copies some data unnecessarily for each iteration.

Figure 13 illustrates the effectiveness of using explicit USM relative to the number of timesteps for various domain sizes. It is clear that explicit USM is slightly superior for any reasonable number of timesteps.

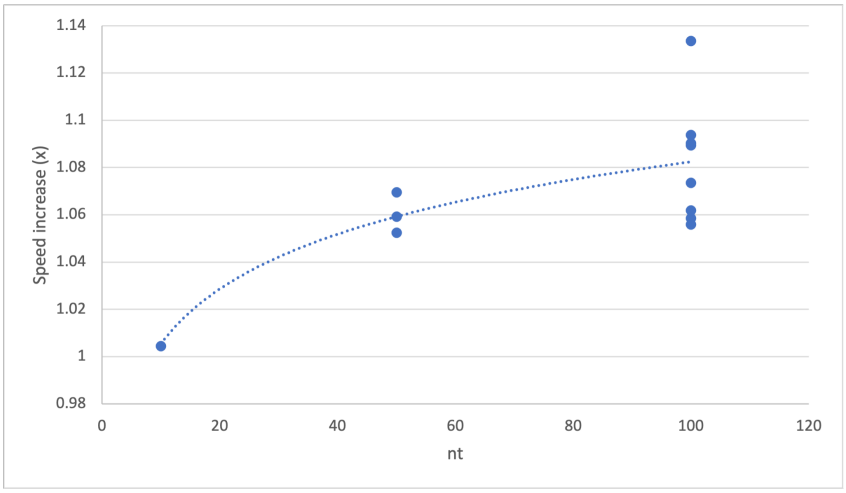


Figure 13: Speed increase achieved by explicit USM relative to implicit USM performance

4.4 Explicit Dependencies

Up to this point, race conditions were prevented with repeated use of q.wait() calls. However, these can be inefficient, as evidenced in one test where .wait() was inadvertently included after a loop inside deryy. This resulted in that function falling from 38% activity to only 2%, with idle time increasing from 2% to 80%.

Figure 14 shows the 'derix' function with the final SYCL implementation, using explicit USM and explicit dependencies. The events 'main' and 'sub' are passed as dependencies for other work groups in 'fluxx'.

```
//========================================================
//  First derivative in x-direction
void derix(double *phi, double *dfi, double &xlx, cl::sycl::event dependent, cl::sycl::event &main, cl::sycl::event &sub){
    double udx=nx/(2*xlx);

    main = q.submit([&](auto &h) {
        h.depends_on(dependent);
        h.parallel_for(cl::sycl::range(ny, nx-2), [=](auto idx) {
            int i = idx[1]+1;
            int j = idx[0];
            dfi[i+nx*j]=phi[nx*j+i+1]-phi[nx*j+i-1];
            dfi[i+nx*j]=dfi[i+nx*j] * udx;
        });
    });
    sub = q.submit([&](auto &g) {
        g.depends_on(dependent);
        g.parallel_for(cl::sycl::range(ny), [=](auto idx) {
            int j = idx[0];
            dfi[nx*j]=udx*(phi[nx*j+1]-phi[nx*(j+1)-1]);
            dfi[nx*(j+1)-1]=udx*(phi[nx*j]-phi[nx*(j+1)-2]);
        });
    });
    return;
}
```

Figure 14: GPU offloaded SYCL implementation using explicit USM and event dependencies

While there is significant scope for program improvements beyond this implementation, it was this version of the code which was provided as the project deliverable. Testing confirmed that this was efficient. Figure 15 illustrates the speed increase when using this implementation to offload to a GPU when compared to G++ serial code with optimisations. This test compared performance for a 1500x1500 domain. A speed increase of 3.8 times may seem slower than those measured previously, although it is important to note that the G++ compiler performs particularly well with optimisations. It should also be noted that this speed increase is specific to the machine used for testing, which consisted of an Intel Devlcoud node with a powerful i9 CPU, and a P630 GPU. Other combinations are likely to result in different performance measurements. Because explicit USM offloads most of the data at the beginning of the program, it is possible to find a critical number of timesteps beyond which the speed up remains constant for the chosen computational domain. In this case, that number is approximately 200 timesteps.

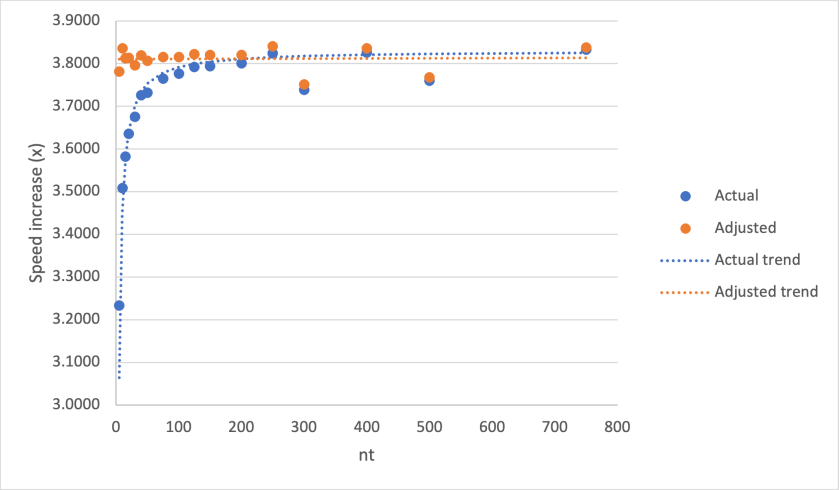


Figure 15: Speed increase relative to number of timesteps for a 1500x1500 domain

By estimating the time spent offloading at the start of the program, which should be constant with domain size, it was possible to adjust the data in figure 15 to measure the speed increase due to parallelisation, negating the effect of data offloading which becomes negligible as the runtime increases.

The final process investigated in this project was use of a sycl::reduction construct to calculate field averages using the GPU, reducing time spent transferring data between the host and the device every iteration. Figure 16 shows how this was implemented (with explicit dependencies), although improvements to this approach are believed to be possible.

```
//============================================================
//  Mean value of 2D field
void average(double* uuu, double &um, cl::sycl::event eDep, cl::sycl::event eDep2, cl::sycl::event &eSend){
    um = 0;
    //  Reduction scheme allows for parallel calculation of averages
    double* utm = cl::sycl::malloc_shared<double>(1, q);
    *utm = 0;
    eSend = q.submit([&](cl::sycl::handler &h) {
        h.depends_on(eDep);
        h.depends_on(eDep2);
        h.parallel_for(cl::sycl::nd_range<1>{nx*ny, sizeof(double)},
         cl::sycl::reduction(utm, cl::sycl::plus<double>()),
         [=](cl::sycl::nd_item<1> idx, auto& utm)
         {
            int i = idx.get_global_id(0);
            utm += uuu[i];
          });
    });
    um = *utm;
    um /= nx*ny;
    return;
}
```

Figure 16: Reduction method used for parallel calculation of averages

4.5 Function Profiling

While testing the implementations discussed above, profiling of the code was conducted using Intel oneAPI Advisor and Vtune. One particularly useful set of results were attained from a deeper investigation into the performance of 'Semi-explicit USM' code for a 3000x3000 domain over 100 timesteps. This was compared to results for other tests, and has been found to represent the general performance of the offloaded solver for most large domains.

This test found that the loop in 'adams' performed particularly poorly, spending a significant time stalled. Additionally, 'adams' produced a much higher L3 miss ratio than any other functions. The smaller loop in 'deryy' appeared to perform poorly, but it was later found that an erroneous .wait() command had resulted in unnecessary idle time. The 'initl' subroutine appeared to be the most efficient by far, with 98.4% active time. A summary of the performance of each function is shown in figure 17,

although it should be noted again that the 'deryy' idle time is erroneous.

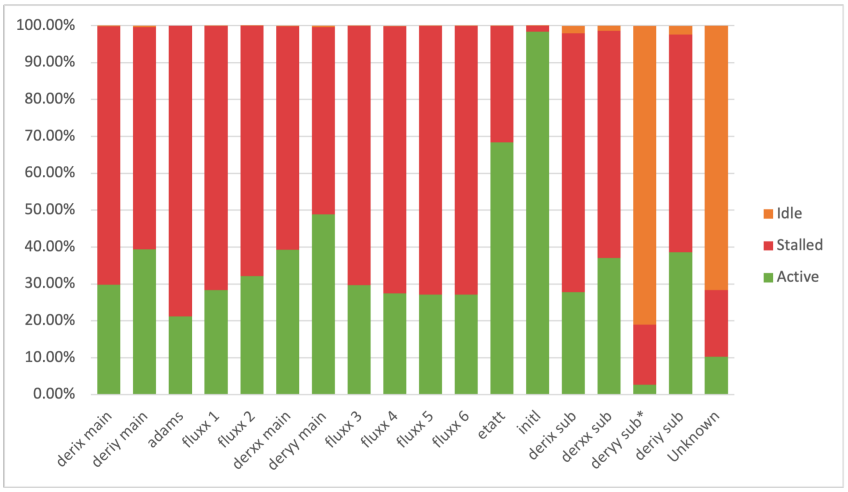


Figure 17: Efficiency of each function on a GPU

Figure 18 shows a diagram produced by Vtune which illustrates the bottlenecks in the 'adams' function. It was clear in this instance that making calculations on six different arrays for each loop iteration was inefficient in parallel, resulting in the excessive miss ratio. By splitting the loop in 'adams' into six separate work groups, it was easier for the GPU to manage data, resulting in an approximate 2% improvement in overall program performance.

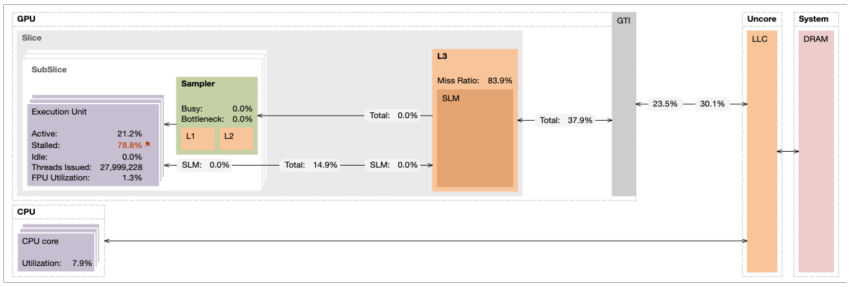


Figure 18: Profile of the 'adams' function in GPU parallel

Vtune results confirmed that dependencies were correctly implemented, with the work queue behaving as expected. The Intel Advisor report suggested that all loops were memory bound, with most bound by GTI bandwidth. Some derivative loops were also bound by L3 bandwidth. It might be possible to improve this by using ND-Range kernels, which give the user even more control over data movements, although that is not covered in the scope of this project.

4.6 Optimisation and Compatibility

It is relatively easy to implement GPU offloading using SYCL, but achieving a significant improvement is a complicated process. Most GPU offloading frameworks such as CUDA and OpenCL are designed to improve performance through hardware-specific optimisations, although these can significantly reduce the portability of such code. SYCL offers several methods to optimise performance, such as ND-Range kernels and Hierarchical kernels. These were not investigated thoroughly, although their importance and potential should be noted.

One surprising issue throughout the GPU parallelisation process was GPU compatibility. SYCL promotes itself as highly portable, but two local machines could not properly offload to the GPU using SYCL due to several support issues. SYCL supports any GPU capable of CUDA offloading, and theoretically any GPU with OpenCL 1.2 support [4]. However, GPGPU drivers and compatibility complicate this. Five GPU devices were available to the researcher, each of which had unique nuances:

1. **AMD Radeon Pro 5300 (Navi-14 Die)**. This GPU supports OpenCL 2.1, and is relatively powerful, having been released less than two years prior to the project. However, AMD do not support

Navi GPUs under ROCm [11], which is required for compilation using hipSYCL and other compilers [6]. Ultimately, this GPU could not be utilised because the manufacturer does not currently support general-purpose computing on it.

2. **Intel HD Graphics 5000 (integrated)**. This integrated GPU is a part of Intel's Gen 7 product line, but their oneAPI software only officially supports Gen 9+ GPUs[8]. It was not recognised by DPC++ as a gpu device, but using the sycl::default_selector, it could be utilised to some extent, using intel_gpu_top to verify that it was in use. This was successfully used as an FPGA emulator device.
3. **Intel HD Graphics P630**. These GPUs are available through Intel Devcloud, and work very well with the DPC++ compiler.
4. **Intel Corp' Device 4905**. These GPUs are also available through Intel Devcloud, but they do not natively support double-precision operations. Because the 2D solver uses double-precision math, DP-Emulation had to be enabled to successfully use these devices. It is not known whether this significantly impacted computational performance or accuracy.
5. **Intel Corp' Device 4905 Dual- and Quad- GPU**. Intel Devcloud nodes also support dual-gpu and quad-gpu devices, but these could not be utilised with this SYCL implementation because each queue only supports one device. While offloading to multiple GPUs may perform better than one GPU (depending on memory bandwidth and management), this is difficult to implement in the 2D solver code.

While SYCL may not be compatible with 'all' GPUs, it is no less compatible than any other framework. Additionally, compilers have been developed to support most operating systems, with hipSYCL support for Linux and MacOS, as well as an experimental Windows version [6]. One advantage to SYCL is that, if a GPU is incompatible, the code can still be run in CPU parallel with no material changes.

A brief test using an Intel Devcloud FPGA node revealed that FPGA offloading performs much faster than offloading to a single GPU. However, it is understood that offloading to multiple FPGAs is very poor when compared to GPU cluster computing.

5 The SYCL Framework

It is evident from the data discussed in previous sections that SYCL is capable of improving the performance of some parallel tasks through CPU or GPU parallelisation, depsite still being in its infancy. SYCL is not currently the most ideal choice for CPU parallelisation or GPU offloading [1], but for those wishing to explore heterogeneous computing with a combination of CPU and GPU parallelism, this may be the most appropriate framework. The SYCL 2020 specification makes it clear that significant changes and improvements are on the way, and the specification looks set to become much more useful in the near future.

5.1 Advantages

- SYCL supports the widest range of hardware of any heterongeneous computing framework
- SYCL uses single-source code
- No changes are required to run SYCL code on any machine, using any combination of devices (provided the code is well-written, and devices are supported by the chosen compiler)
- SYCL gives the user control over memory management with buffers or USM
- GPU offloading results in an increase in speed of more than 4 times when tested on a 2D CFD Solver, with a maximum increase measured at 25 times
- Several SYCL implementations are under development, meaning that it is widely supported and compiler performance is steadily improving
- The SYCL standard is under significant development, with many excellent features promised in future versions
- SYCL interacts with OpenMP, OpenCL, and CUDA, which are existing standards with known behaviours, making it easy to understand
- Several high-performance computing systems already support SYCL [12]

5.2 Limitations

- SYCL only supports C or C++ code, so high-performance Fortran code is not supported
- Even with the greatest range of hardware support, SYCL still lacks a significant proportion of hardware devices. Support from AMD is particularly poor
- Optimising for specific hardware may require changes to the source code, so portability could be reduced if maximum performance is desired
- Current SYCL compilers are all relatively slow
- Adding SYCL to existing code is not as easy as a directive-based process such as OpenMP or OpenACC
- SYCL parallelism adds significant overhead to programs, regardless of compiler choice
- GPU offloading, in this test case, could not be made more efficient than CPU parallelisation

6 Conclusions

6.1 Testing Results

It is clear that choice of compiler and optimisations will always make a significant difference to performance, and stack-based C++ code was found to perform much worse than the original Fortran 90 code. Further testing using heap-based C++ code did confirm that C++ performs similarly to Fortran90 in serial, when similar compilers are used.

CPU parallelisation using SYCL (C++) and OpenMP (Fortran) revealed that a speed increase of at least 1.5 times (up to 2.7 times) can be achieved using relatively crude parallelism. It found that using hipSYCL to execute in CPU parallel is almost half as effective as using pure OpenMP in Fortran90, despite hipSYCL using OpenMP to launch CPU parallel kernels. While the same framework is used for both methods, the additional overhead involved in using SYCL, as well as the inherent speed decrease due to use of stack memory, made SYCL much less useful.

Testing using buffers and accessors in stack-based C++ code was incredibly slow. There are several changes which might improve this, including moving buffer creation outside of some functions and using heap memory instead of stack. Ultimately, however, it is not believed that buffers and accessors could be made effective in this context.

Testing using implicit USM was much more successful. When compared to the slow DPC++ compiler in serial, a speed increase of more than 20 times was recorded (although this number should be treated with caution). When compared to CPU parallel performance on a 24-processor Intel Devcloud node, no significant improvement was found by offloading to the GPU but there was also no measured decrease in performance caused by offloading for large domains.

Explicit USM was found to perform up to 1.13 times faster than implicit USM, although this could increase with greater timesteps and different hardware. Testing compared to the powerful G++ compiler in serial revealed a speed increase of 3.8 times when GPU offloading for a 1500x1500 domain over a suitably large number of timesteps. Further testing for larger computational domains would be recommended.

6.2 Recommendations regarding GPU offloading

Before considering GPU offloading or heterogeneous parallelism in a CFD solver, it is vital that the developer understands where bottlenecks are likely to occur by identifying regions of serial tasks. For example, the 2D solver calculated field averages after each timestep, so the more complicated reduction class was needed to offload this to GPU devices. Additionally, hardware compatibility should be considered. Where is the program intended to run, and is SYCL the most appropriate method? As of summer 2021, if it is possible to use OpenMP or OpenCL for a program instead, that would be recommended [1].

To successfully implement SYCL, the Intel DPC++ textbook [10] is highly recommended for new users. Starting with basic kernels is an excellent way to test the potential of a program in parallel, and these can then be expanded into ND-Range or Hierarchical kernels for improved performance. This project found explicit USM to be the best memory management tool, but users would be encouraged to explore use of buffers and accessors in different code samples. Explicit event dependencies are an excellent tool for improving performance, but extreme care must be taken to prevent race errors, which can be very difficult to identify and resolve. Some race errors will not be obvious on certain hardware/ compiler combinations, so testing on the widest possible range of hardware is also encouraged.

6.3 Recommendations regarding SYCL

SYCL is not just a tool for GPU offloading, but it is designed for heterogeneous systems. The framework certainly does achieve this goal, by allowing users to write highly portable code which utilises CPU and GPU resources. The final C++ implementation was successfully run in CPU parallel on three different machines, GPU parallel on two different devices, and FPGA parallel on two different devices, all without any changes to the code itself. There are few, if any, other frameworks which would allow developers to achieve this with such ease.

Therefore, SYCL meets most modern requirements for portability, but its current limitations lie in performance. It is also more difficult to implement than some frameworks such as OpenMP or OpenACC, although there is a trade-off between ease-of-use and user control. If portability is of principal concern, SYCL may be appropriate, but if ultimate performance is more important and the code is only to be run on a select number of machines, it may be more appropriate to use conventional methods for now.

6.4 Final thoughts

This project barely scratches the surface of the capabilities, applications, and limitations of SYCL. The framework is currently used in HPC settings [12], and certainly has its place for those wishing to explore portable heterogeneous computing. As a framework still in its infancy, SYCL performs well, and implementations are clearly developing into more efficient and useful tools.

It is believed that SYCL will continue to develop and grow into a truly useful tool for heterogeneous computing, eventually providing superior performance portability to many of its current 'competitors'. To achieve this, further unification is necessary between device manufacturers and developers. It is expected that SYCL will, at some point, gain the performance needed to truly compete with alternative parallelism methods.

References

[1] H C D Silva, F Pisani, E Borin. 2016. *A Comparative Study of SYCL, OpenCL, and OpenMP.* International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA. pp. 61-66. doi: 10.1109/SBAC-PADW.2016.19.

[2] J A Herdman et al. 2012. *Accelerating Hydrocodes with OpenACC, OpenCL and CUDA.* SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 465-471. doi: 10.1109/SC.Companion.2012.66.

[3] R Burns. 5 November 2020. *Codeplay Supporting SYCL at SuperComputing 2020.* Codeplay. Available at: https://www.codeplay.com/portal/news/2020/11/05/codeplay-supporting-sycl-at-supercomputing-2020.html [Last accessed: 13.08.2021]

[4] Khronos. 2021. *SYCL 202 is Here!* Available at: https://www.khronos.org/sycl/ [Last accessed: 13.08.2021]

[5] Khronos. 19 March 2014. *Khronos Releases SYCL 1.2 Provisional Specification.* Available at: https://www.khronos.org/news/press/khronos-releases-sycl-1.2-provisional-specification [Last accessed: 13.08.2021]

[6] A Alpay, V Heuveline. 2020. *SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL.* In Proceedings of the International Workshop on OpenCL (IWOCL '20). Association for Computing Machinery, New York, NY, USA, Article 8, 1. DOI:https://doi.org/10.1145/3388333.3388658

[7] triSYCL. 2021. *triSYCL.* Available at: https://github.com/triSYCL/triSYCL [Last accessed: 13.08.2021]

[8] Intel. 2021. *IntelÂ® oneAPI DPC++/C++ Compiler.* Available at: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html#gs.8ul3u0 [Last accessed: 13.08.2021]

[9] B Randal, D O'Hallaron. 2016. *Computer Systems: A Programmer's Perspective.* 3 ed, Pearson Education, ISBN 978-1-488-67207-1. p. 58.

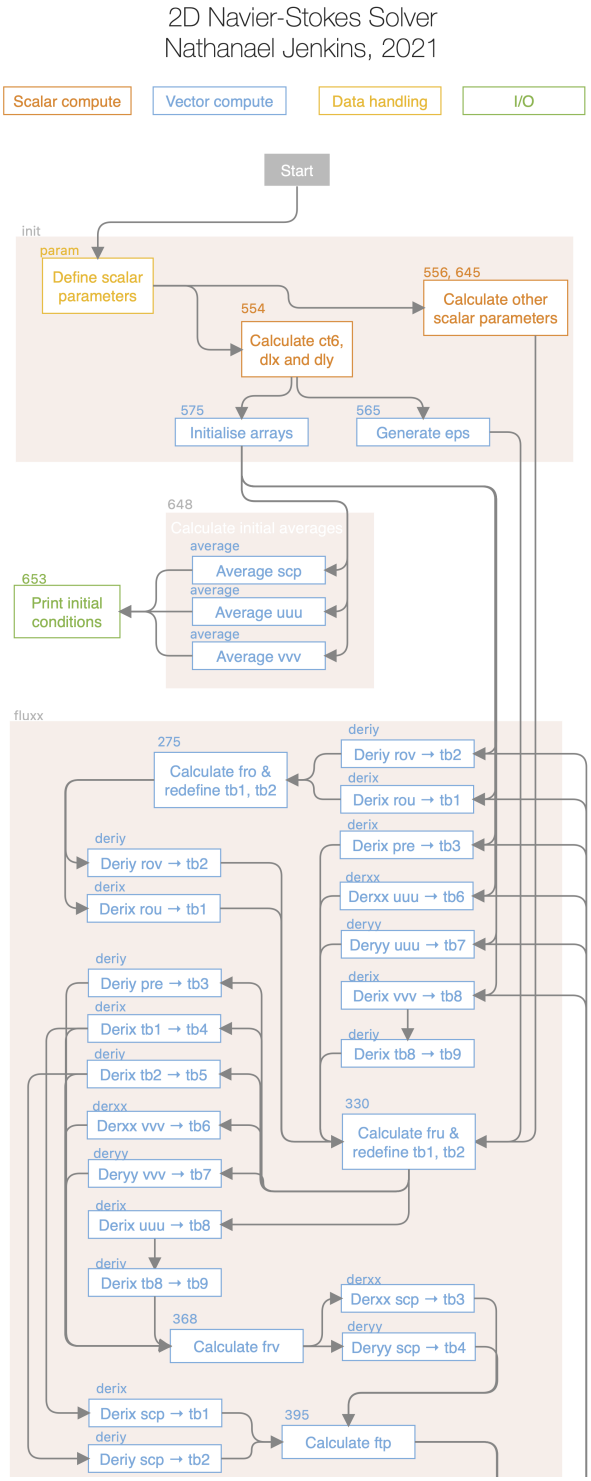
[10] J Reinders, et al. 2021. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL.* Intel Corporation, Apress Open, New York. doi: https://doi.org/10.1007/978-1-4842-5574-2

[11] ianferreira. 5 August 2021. *ROCM support for Navi (GFX10).* GitHub Issue, ROCm, #1547. Available at: https://github.com/RadeonOpenCompute/ROCm/issues/1547 [Last accessed: 13.08.2021]

[12] N Hemsoth. 3 February 2021. *Can SYCL Slice into Broader Supercomputing?* The Next Platform. Available at: https://www.nextplatform.com/2021/02/03/can-sycl-slice-into-broader-supercomputing/ [Last accessed: 13.08.2021]

Appendix A

A dependency tree, as described by Intel [10] provides an overview of operations inside a program, and their dependencies on the completion of other operations. This can be implemented in SYCL using event-based dependencies to maximise the performance of parallel code, by allowing independent tasks to run out of order.



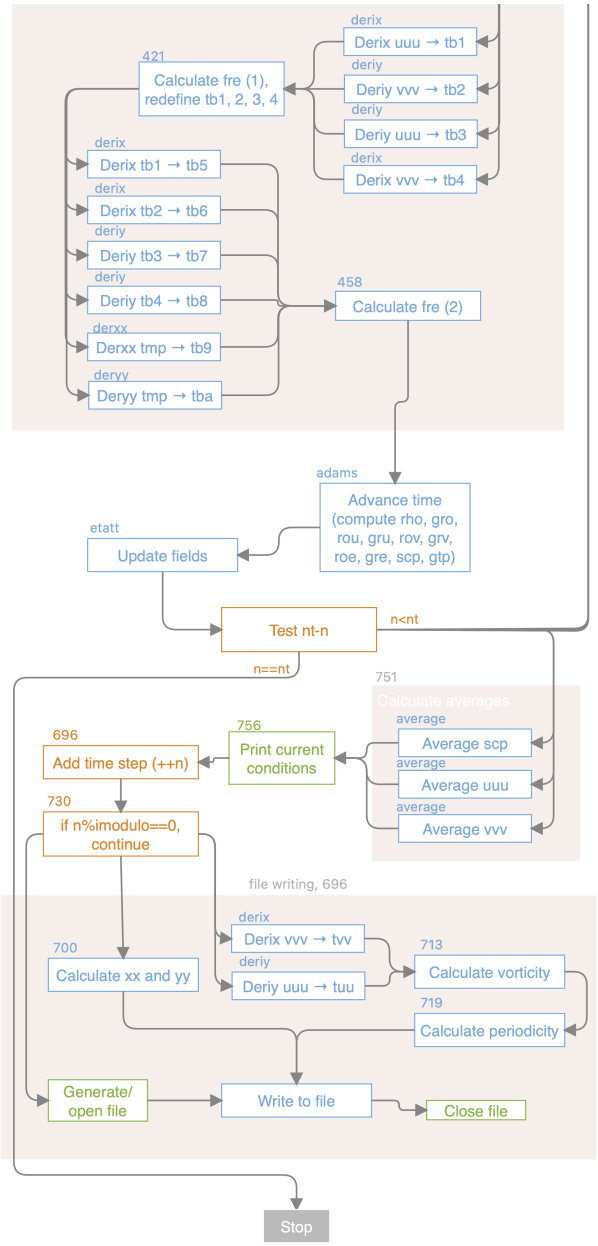


Figure 19: Dependency tree for a 2D Finite-Difference Navier-Stokes Solver

Appendix B

Code samples and data are available to download at these links [last updated August 2021]:
Original Fortran90 code, nathanaelj.github.io/PrivateSamples/Original.f90
Original C++ code, nathanaelj.github.io/PrivateSamples/Original.cpp
Final C++/SYCL code, nathanaelj.github.io/PrivateSamples/Final.cpp
Makefile for final C++/SYCL code, nathanaelj.github.io/PrivateSamples/Makefile
Testing data in full, nathanaelj.github.io/PrivateSamples/TestingData.xlsx