PC2 Hackathon Reports 2023

Contents

Quantum Optimization with Photonic Quantum Computers

Participants

Zarin Shakibaei, Zuse Institute Berlin

Report

Optimization problems with continuous variables like

$$min_{\vec{x}\in\mathbb{R}^N} f(\vec{x})$$

can be solved with photonic quantum computers using the cv-QAOA (continuous-variable quantum approximate minimization algorithm, [**?**]). Like quantum computers using superconducting qubits, there is a universal set of operations ("gates") for photonic quantum computing, that can be used to realize, in principle, every quantum operation. Such a set of universal gates is, for example

$$e^{i\frac{t}{\hbar}\hat{x}_i}, e^{i\frac{t}{2\hbar}\hat{x}_i^2}, e^{i\frac{t}{3\hbar}\hat{x}_i^3}, e^{i\frac{t}{\hbar}\hat{x}_i\hat{x}_j}, e^{i\frac{\theta}{2\hbar}(\hat{x}_i^2+\hat{p}_i^2)}$$

where $\hat{x}_i$ is the position operator for mode i and $\hat{p}_i$ is the momentum operator for mode i.

One of the main steps in the cv-QAOA is to set up a cost Hamiltonian $\hat{H}_C = f(\hat{x})$ and apply this Hamiltonian to the photonic state as

$$U_C(\alpha) = e^{-i\alpha\hat{H}_C}.$$

For example, for an objective function of $f(x) = \hat{x}_1\hat{x}_2^4 + \hat{x}_1\hat{x}_2\hat{x}_3$ this would be

$$\hat{U}_C(\alpha) = e^{-i\alpha(\hat{x}_1\hat{x}_2^4+\hat{x}_1\hat{x}_2\hat{x}_3)}.$$

Clearly, this can't be performed with the available operations from the universal gate set.

Thus, the main task of this hackathon project was to set up a Python implementation that, with the help of known decompositions like

$$e^{i\alpha\hat{x}_j\hat{x}_k\hat{x}_l} = e^{i\frac{\alpha}{6}(\hat{x}_j+\hat{x}_k+\hat{x}_l)^3}e^{-i\frac{\alpha}{6}(\hat{x}_j+\hat{x}_k)^3}e^{-i\frac{\alpha}{6}(\hat{x}_j+\hat{x}_l)^3}e^{-i\frac{\alpha}{6}(\hat{x}_k+\hat{x}_l)^3}e^{i\frac{\alpha}{6}\hat{x}_j^3}e^{i\frac{\alpha}{6}\hat{x}_k^3}e^{i\frac{\alpha}{6}\hat{x}_l^3}$$

recursively decomposes $\hat{U}_C$ to sequences of universal gates. To verify the accuracy of the decomposition, we have implemented the general 4-mode operators in Strawberry Fields,

$$e^{itx_i^a x_j^b x_k^c x_k^d}$$

with $a, b, c, d \in \{0, 1, 2, ...\}$. However, it turned out that the application of gate operations in Strawberry Fields for larger Fock space cutoffs and mode counts is very problematic with respect to memory usage and efficiency. Thus, we have evaluated other frameworks with similar functionality or a subset of the functionality, especially Perceval and QuantumOptics.jl.

The main functionalities required for cv-QAOA are:

- creation of momentum-squeezed multi-qubit initial states
- application of sequences of universal one- and two-mode gates
- measurement of position space-probability distributions in homodyne measurements

Our work with respect to the frameworks is summarized in the following table:

Aspect	Strawberry Fields	Strawberry Fields	Perceval	QuantumOptics.jl	QuantumOptics.jl
Version	mainline	our modification	mainline	mainline	our modification
sparse states	no		yes	optional	yes
sparse operators	no		yes	optional	yes
creation of single-mode squeezed states	yes	no	no	yes	
efficient creation of multi-mode squeezed states	no	no	no	no	yes
efficient application of single-mode gates	no		yes	no	yes
efficient application of two-mode gates	no		yes	no	to be done
gates with arbitrary x-polynoms of up to 4 operators	no	yes	no	no	to be done
sampled homodyne measurements	yes		no	no	to be done
distributions from homodyne measurements	no	yes	no	no	to be done

Multithreading Performance Improvements in Peridynamics.jl

Participants

Kai Partmann, Universität Siegen

Report

As part of the PC^2 hackathon, we focused on the multithreading performance of the Peridynamics.jl package.

Challenges in Existing Code:
The v0.2 codebase poses several challenges. Firstly, it shows suboptimal multithreading scaling. False sharing issues slow down the bottleneck function `compute_forcedensity!` due to using a global variable containing data for all threads. Moreover, the existing thread assignment strategy for bond partitions neglected data locality, inhibiting better data access patterns.

Improvements:
With the help of the HPC Advisers, an improved short version of the package could be obtained that addresses these issues. Each thread is now equipped with a thread local copy of data, effectively mitigating false sharing issues. A two-loop approach involves iterations over every point of the thread and the bonds of each point. The `@threads :static` directive is now consistently used, and threads are pinned using the Julia package `ThreadPinning.jl`.

Benchmarks:
These changes result in a remarkable performance boost, and it is crucial to note that the full simulation contains serial code portions that remain unaltered.

Even greater speedup improvements can be seen by focusing on a benchmark of the bottleneck function `compute_forcedensity!`. When threads are pinned to NUMA domains, the performance for 64 threads improves even more compared to pinning to cores.

Many thanks to Carsten Bauer, Xin Wu, and Alex Wiens for their valuable help and HPC insights.

Implementing the Wilson Dslash operator and its inverse in SIMULATeQCD

Participants

Jangho Kim, Forschungszentrum Jülich
Sajid Ali, Universität Bielefeld
Pavan Pavan, Universität Bielefeld
Simran Singh, Universität Bielefeld
Dibyendu Bala, Universität Bielefeld

Report

On the first day of the Hackathon, we initiated our activities by presenting a comprehensive plan for the entire Hackathon week. Subsequently, in a hands-on session, we established a new branch within our SIMULATeQCD codebase and successfully extended the spinor structure to accommodate Wilson-clover fermions, marking a pivotal step in enhancing the overall functionality of our code. For those interested, our code can be accessed via this link: https://github.com/LatticeQCD/SIMULATeQCD

Over the following days, our dedicated efforts were focused on implementing the Wilson-clover Dslash operator, which is defined as follows:

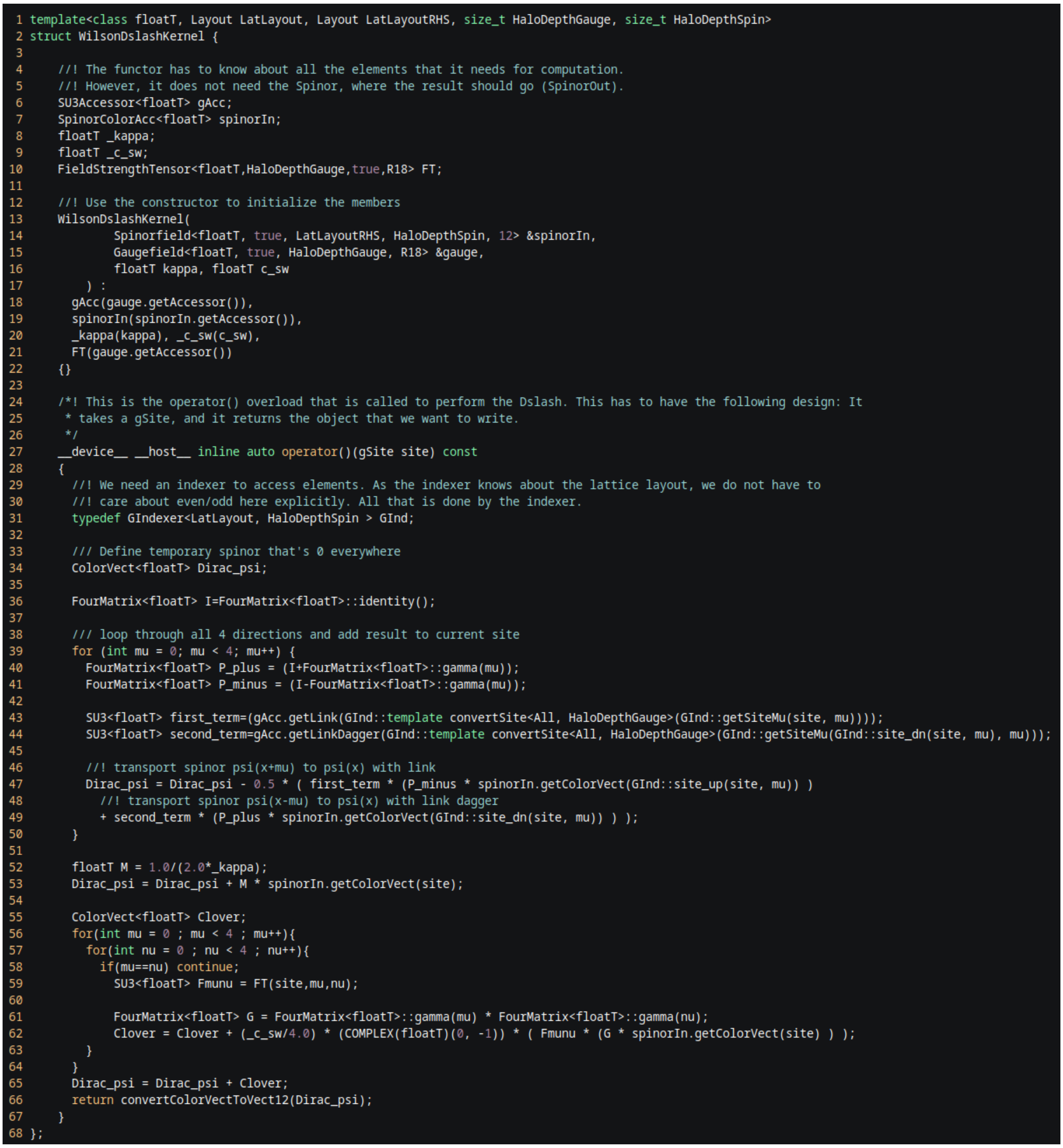
$D^{a,b}_{\alpha,\beta}(x,y)\psi^{b}_{\beta}(y) = \frac{1}{2\kappa}\psi^{a}_{\alpha}(x) - \frac{1}{2}\sum_{\mu=0}^{3}[U^{a,b}_{\mu}(x)P^{-\mu}_{\alpha,\beta}\psi^{b}_{\beta}(x+\hat{\mu}) + U^{\dagger\,a,b}_{-\mu}(x)P^{+\mu}_{\alpha,\beta}\psi^{b}_{\beta}(x-\hat{\mu})] + \frac{c_{sw}}{2}\sum_{\mu=0}^{3}\sum_{\mu<\nu}\sigma^{\mu,\nu}_{\alpha,\gamma}\hat{F}^{a,c}_{\mu,\nu}(x)\psi^{c}_{\gamma}(x)$

where $\hat{F}_{\mu\nu}(n)$ and $P^{\pm\mu}$ are defined as

$\hat{F}_{\mu\nu}(x) = -\frac{i}{8}\left[U_{\mu\nu}(x) - U_{\nu\mu}(x)\right]$ and $P^{\pm\mu} = 1 \pm \gamma^{\mu}$ are defined as such

Additionally, we implement the Bi-CGStab algorithm and use it to compute the inverse of this Wilson-clover Dslash operator.
This inverse Dslash operator, also referred to as the fermion propagator, plays a pivotal role in our numerical computations and is essential for computing observables in Lattice QCD. Specifically, we will utilize this implementation to compute the Energy-Momentum Tensor on the lattice, with a particular focus on determining Shear and Bulk Viscosity.

In summary, our team made substantial progress during the Hackathon, effectively achieving our set objectives and enhancing the capabilities of our codebase. Nevertheless, we recognize the need for further optimization and rigorous testing at a significant level. We eagerly look forward to the continued refinement of our work and the ultimate realization of our project's objectives in the future.



```cpp
template<class floatT, Layout LatLayout, Layout LatLayoutRHS, size_t HaloDepthGauge, size_t HaloDepthSpin>
struct WilsonDslashKernel {

    //! The functor has to know about all the elements that it needs for computation.
    //! However, it does not need the Spinor, where the result should go (SpinorOut).
    SU3Accessor<floatT> gAcc;
    SpinorColorAcc<floatT> spinorIn;
    floatT _kappa;
    floatT _c_sw;
    FieldStrengthTensor<floatT,HaloDepthGauge,true,R18> FT;

    //! Use the constructor to initialize the members
    WilsonDslashKernel(
            Spinorfield<floatT, true, LatLayoutRHS, HaloDepthSpin, 12> &spinorIn,
            Gaugefield<floatT, true, HaloDepthGauge, R18> &gauge,
            floatT kappa, floatT c_sw
        ) :
      gAcc(gauge.getAccessor()),
      spinorIn(spinorIn.getAccessor()),
      _kappa(kappa), _c_sw(c_sw),
      FT(gauge.getAccessor())
    {}

    /*! This is the operator() overload that is called to perform the Dslash. This has to have the following design: It
     * takes a gSite, and it returns the object that we want to write.
     */
    __device__ __host__ inline auto operator()(gSite site) const
    {
      //! We need an indexer to access elements. As the indexer knows about the lattice layout, we do not have to
      //! care about even/odd here explicitly. All that is done by the indexer.
      typedef GIndexer<LatLayout, HaloDepthSpin > GInd;

      /// Define temporary spinor that's 0 everywhere
      ColorVect<floatT> Dirac_psi;

      FourMatrix<floatT> I=FourMatrix<floatT>::identity();

      /// loop through all 4 directions and add result to current site
      for (int mu = 0; mu < 4; mu++) {
        FourMatrix<floatT> P_plus = (I+FourMatrix<floatT>::gamma(mu));
        FourMatrix<floatT> P_minus = (I-FourMatrix<floatT>::gamma(mu));

        SU3<floatT> first_term=(gAcc.getLink(GInd::template convertSite<All, HaloDepthGauge>(GInd::getSiteMu(site, mu))));
        SU3<floatT> second_term=gAcc.getLinkDagger(GInd::template convertSite<All, HaloDepthGauge>(GInd::getSiteMu(GInd::site_dn(site, mu), mu)));

        //! transport spinor psi(x+mu) to psi(x) with link
        Dirac_psi = Dirac_psi - 0.5 * ( first_term * (P_minus * spinorIn.getColorVect(GInd::site_up(site, mu)) )
          //! transport spinor psi(x-mu) to psi(x) with link dagger
          + second_term * (P_plus * spinorIn.getColorVect(GInd::site_dn(site, mu)) ) );
      }

      floatT M = 1.0/(2.0*_kappa);
      Dirac_psi = Dirac_psi + M * spinorIn.getColorVect(site);

      ColorVect<floatT> Clover;
      for(int mu = 0 ; mu < 4 ; mu++){
        for(int nu = 0 ; nu < 4 ; nu++){
          if(mu==nu) continue;
            SU3<floatT> Fmunu = FT(site,mu,nu);

            FourMatrix<floatT> G = FourMatrix<floatT>::gamma(mu) * FourMatrix<floatT>::gamma(nu);
            Clover = Clover + (_c_sw/4.0) * (COMPLEX(floatT)(0, -1)) * ( Fmunu * (G * spinorIn.getColorVect(site) ) );
        }
      }
      Dirac_psi = Dirac_psi + Clover;
      return convertColorVectToVect12(Dirac_psi);
    }
};
```

Code snippet of Wilson Dslash Kernel

Optimizing MKL Thread Usage for Improved Performance in Julia

Participants

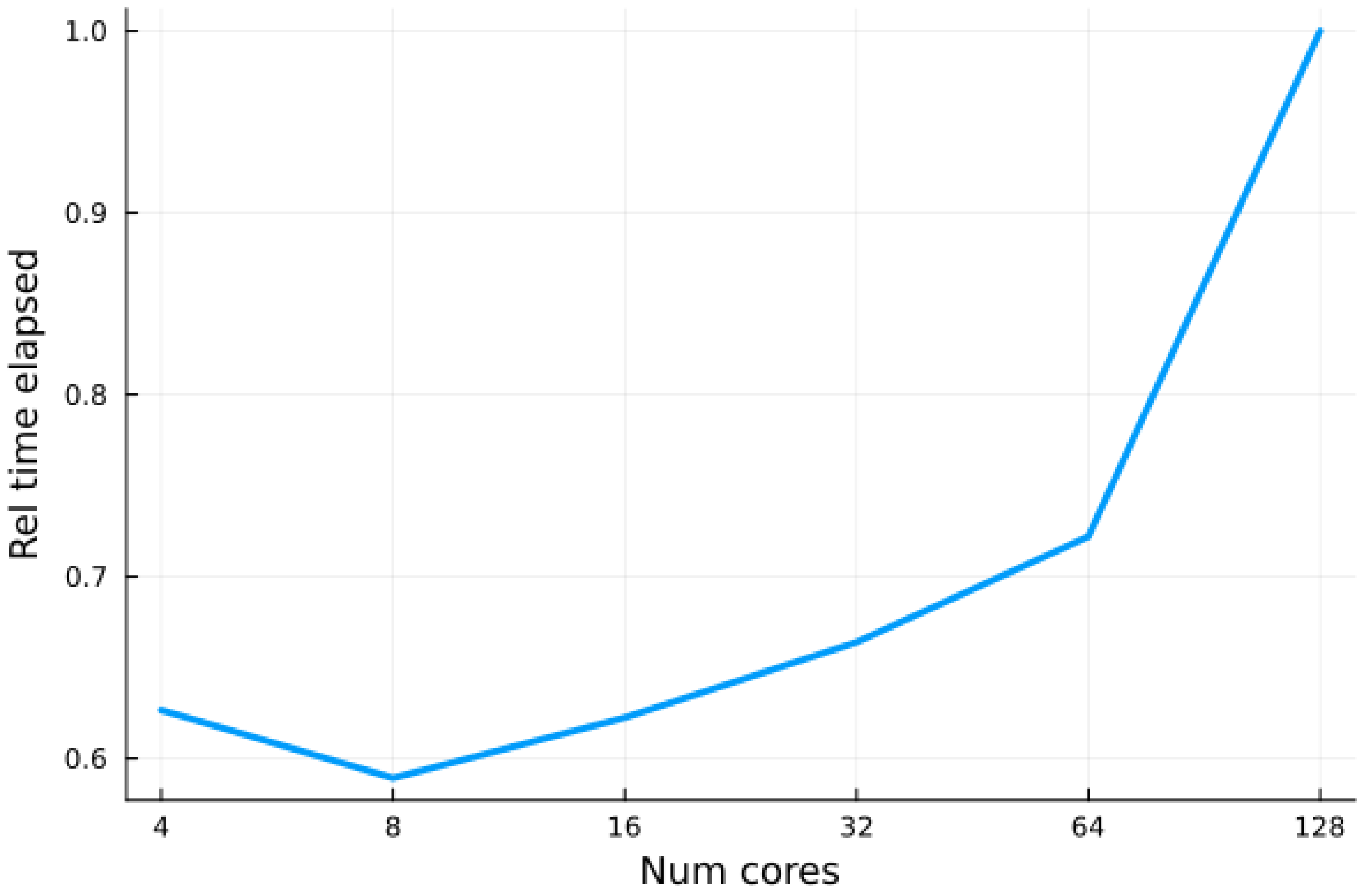
Dominik Itner, Universität Duisburg-Essen

Report

The problem: A collection of independent optimizations need to be run to collect statistical quantaties of interest to evaluate the success of optimization. A meta-optimizer uses this data to improve optimization performance by optimizing hyper-parameters and properties of the underlying simulation. To this end, the Distributed package in Julia was used to spread the workload. However, the use of MKL, required for improved linear algebra operations, lead to an overloaded use of cores by spamming too many threads. MKL, without any user specifications, tries to use as many threads as are physically possible. Each Julia worker would load its own MKL instance, as such many more threads were started than should be to utilize the hardware correctly. This leads to these threads fighting over phyiscal cores, pushing each other off and leading to cold caches.

Because individual optimizers do not communicate, the use of Distributed is not strictly necessary and Slurm Arrays can be used to efficiently call individual jobs, which may run a single optimizer or an ensemble of optimizers sequentially. This guarantees optimal use of hardware and best performance.

Conclusion: The problem was not the absence of multi-threaded MKL but rather the overuse of automatic multi-threaded MKL. In actuality, there is a sweet spot of number of cores to use for the eigenvalue solver in MKL.



FPGA Acceleration of Breadth-First Search Algorithm

Participants

Kaan Olgu, University of Bristol

Report

1. Single Source Multiple CMake File Build Reduces Performance!

The project at the public repository has 4 different folders for four different compute unit setup. The difference between each other is how many times the `explore` kernel is called and the graph related differences `offsets,number of files,etc.` In order to reduce the confusion, I tried to create a single project which would include the required files according to the user defined attributes given to the CMake during the `cmake ..` stage. I am including the snippets from the following files.

Top Level CMakeLists.txt file :

```
add_subdirectory (benchmarks/1cu)
add_subdirectory (benchmarks/2cu)
add_subdirectory (benchmarks/3cu)
add_subdirectory (benchmarks/4cu)
add_subdirectory (benchmarks/8cu)
```

The source file from the `benchmarks/4cu` folder `CMakeLists.txt`:

```
set(SOURCE_FILE ${PROJECT_SOURCE_DIR}/src/host.cpp GraphProcessing.cpp)
set(NUMBER_CU 4)
set(TARGET_NAME bfs_${NUMBER_CU})
set(EMULATOR_TARGET ${TARGET_NAME}.bfs_emu)
set(FPGA_TARGET ${TARGET_NAME}.fpga)
...
# For the hardware built added this extra line
target_include_directories(${FPGA_TARGET} PRIVATE ${PROJECT_SOURCE_DIR}/benchmarks/${
    NUMBER_CU}cu)
```

The performance difference is massive compared to different folders with different CMake for each compute unit. There is a performance decrease in single project multiple CMakes with 19.8%

---

2. OneAPI 2023.2.0 Generated Report Lags for Area Estimates

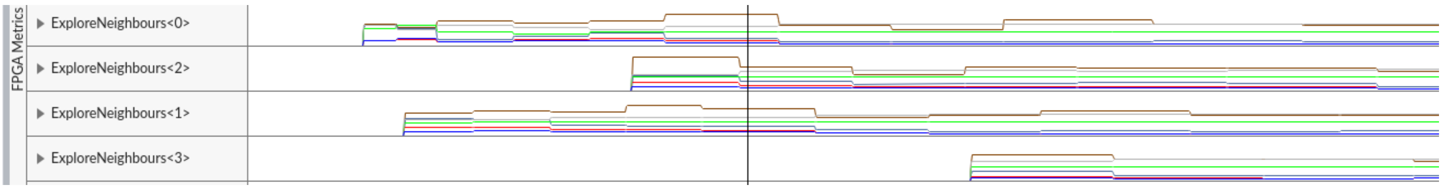
The generated report to analyse the design lags for the Area Estimates part which I assume is related with the Javascripts they added but it is super slow now. Double tested in earlier versions 2023.1.0 and 2023.0.0 does not have this issue so it is not related with the size of my project. When I try to open a kernel area estimate I click and it responds after 4-5s even highlighting appears after 2-3 seconds.

3. The Scheduler Delay in Parallel Execution of Kernels in Same Queue!

The source code for the BFS project, has 4 explore kernels submitted out of order and needs to be executed in parallel since they don't read from the same memory buffer. They access different zones with offsets in the following code part :

```
e1_1 = parallel_explorer_kernel<0>(q,h_over,offset[0],offset_inds[0],usm_nodes_start,
    usm_edges,usm_dist,usm_pipe, usm_updating_mask,usm_visited);
e1_2 = parallel_explorer_kernel<1>(q,h_over,offset[1],offset_inds[1],usm_nodes_start,
    usm_edges,usm_dist,usm_pipe, usm_updating_mask,usm_visited);
e1_3 = parallel_explorer_kernel<2>(q,h_over,offset[2],offset_inds[2],usm_nodes_start,
    usm_edges,usm_dist,usm_pipe, usm_updating_mask,usm_visited);
e1_4 = parallel_explorer_kernel<3>(q,h_over,offset[3],offset_inds[3],usm_nodes_start,
    usm_edges,usm_dist,usm_pipe, usm_updating_mask,usm_visited);
q.wait();
```

The issue could be easily seen at the AOCL profiler results:



Kernel Name and Start Time:

ExploreKernel<0> : 8.73ms

ExploreKernel<1> : 8.95ms

ExploreKernel<2> : 10.21ms

ExploreKernel<3> : 11.48ms

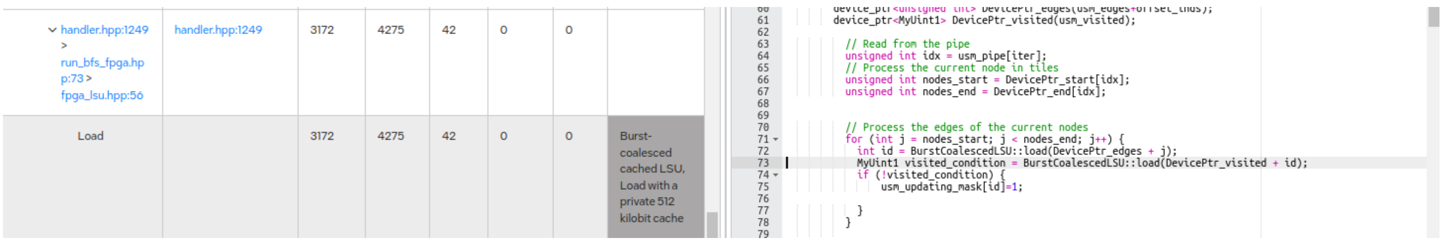
So the difference between the First and Last explore kernel submission time is tremendous compared to the execution time of the shortest kernel which runs for 8ms then the delay of starting later of 3ms actually corresponds in **32.6%** performance degrade. The time differences are observed in different datasets and the differences are always quite similar (~2.75ms) which brings us to conclusion that it is irrelevant of the dataset size or characteristics explored.

4. `ac_int<1,false>` datatype is not interpreted as 1 bit

On the second day of Hackathon, we discovered that ac_int data types like 1 bit custom datatype does not work with the Cache .

```
using CachingLSU =  ext::intel::lsu<ext::intel::burst_coalesce<true>,
                                    ext::intel::cache<1024*1024>,
                                    ext::intel::statically_coalesce<false>>;
int X = CachingLSU::load(input_ptr);
```

According to this official example, it should create 1 MByte of Cache (1024*1024) By default it is implemented as 512 kilobit private memory in reports and it does not change with the cache size changes in the code.



⌄ handler.hpp:1249 > run_bfs_fpga.hpp:73 > fpga_lsu.hpp:56	handler.hpp:1249	3172	4275	42	0	0	
Load		3172	4275	42	0	0	Burst-coalesced cached LSU, Load with a private 512 kilobit cache

```
        device_ptr<unsigned int> DevicePtr_edges(usm_edges+offset_inds);
        device_ptr<MyUint1> DevicePtr_visited(usm_visited);

          // Read from the pipe
          unsigned int idx = usm_pipe[iter];
          // Process the current node in tiles
          unsigned int nodes_start = DevicePtr_start[idx];
          unsigned int nodes_end = DevicePtr_end[idx];


          // Process the edges of the current nodes
          for (int j = nodes_start; j < nodes_end; j++) {
            int id = BurstCoalescedLSU::load(DevicePtr_edges + j);
            MyUint1 visited_condition = BurstCoalescedLSU::load(DevicePtr_visited + id);
            if (!visited_condition) {
              usm_updating_mask[id]=1;

            }
          }
```

When we changed the dataype to char (that is also an issue, so the 1 bit custom datatype is actually implemented as 8 bits anyway so had padding of 7 bits in the reports it is evident) it reacted to the changes in the cache size and we set it to 16 Megabit private memory and started compilation. So this concludes us that 1 bit is converted to char with padding and the related cache sizes are not working properly. After fixing this issue overall **20%** performance increase was observed

Hackathon Performance Results

For Real-World Datasets

- Youtube where the number of edges processed is quite low in each step which forces the design to do empty iterations Throughput improvement

  27.41%

For Synthetic Datasets

- Medium Size Edge Factor in terms of the number of processed edges per run is on average on (RMAT-24-16) 4 Bytes per edge, and the synthetic dataset had 2 million nodes where around 1/4th of it distributed to each compute unit and 500k nodes which brings us the 2 MBytes of data and our Cache size is 1 Million * 1 byte= 1MByte. We have hit rate of ~50% The improvement is again around 27.68%
- Large Size Edge Factor (RMAT-19-32) Graph where we have lots of edges to process this is where the new improvements excel we have achieved 2x higher raw performance in total kernel execution times but since we have kernel launch overheads this corresponds to overall 22.30% increase in performance

  To sum up overall on average 1.25x performance increase.

Optimisations

- Introduced caches to store the visited node data for Explore kernels
- Optimised irregular accesses at the PipeGenerator with introduction of custom buffer size 16 (for best clock frequency) kernel which was the main bottleneck for Small World datasets
- CPU results are correct now for double check purposes and print looks pretty now!

We have lots of ideas to further to boost the performance!

Acknowledgements

Thanks to Paderborn Center for Parallel Computing hosting this hackathon and Tobias Kenter, Michael Laß and Heinrich Riebler for their helps during the hackathon.

Moving from software based to FPGA accellerated RTL simulation

Participants

Robert Beck, Universität Paderborn
Tom Nellius, Universität Paderborn
Lars Luchterhandt, Universität Paderborn

Report

We started the Hackathon introducing our CPU architecture research project group GRIP-V. In the last year we implemented the grid of processing cells (GPC) proposed by Rainer Dömer from UC Irvine based on open-source RISC-V SoC generators. Part of the evaluation workflow requires RTL level simulation of the implemented design. Integrated into Chipyard framework is the Verilator RTL simulation workflow. While sufficient for small example workloads the execution time of more complex programs is increasing, slowing down the development.

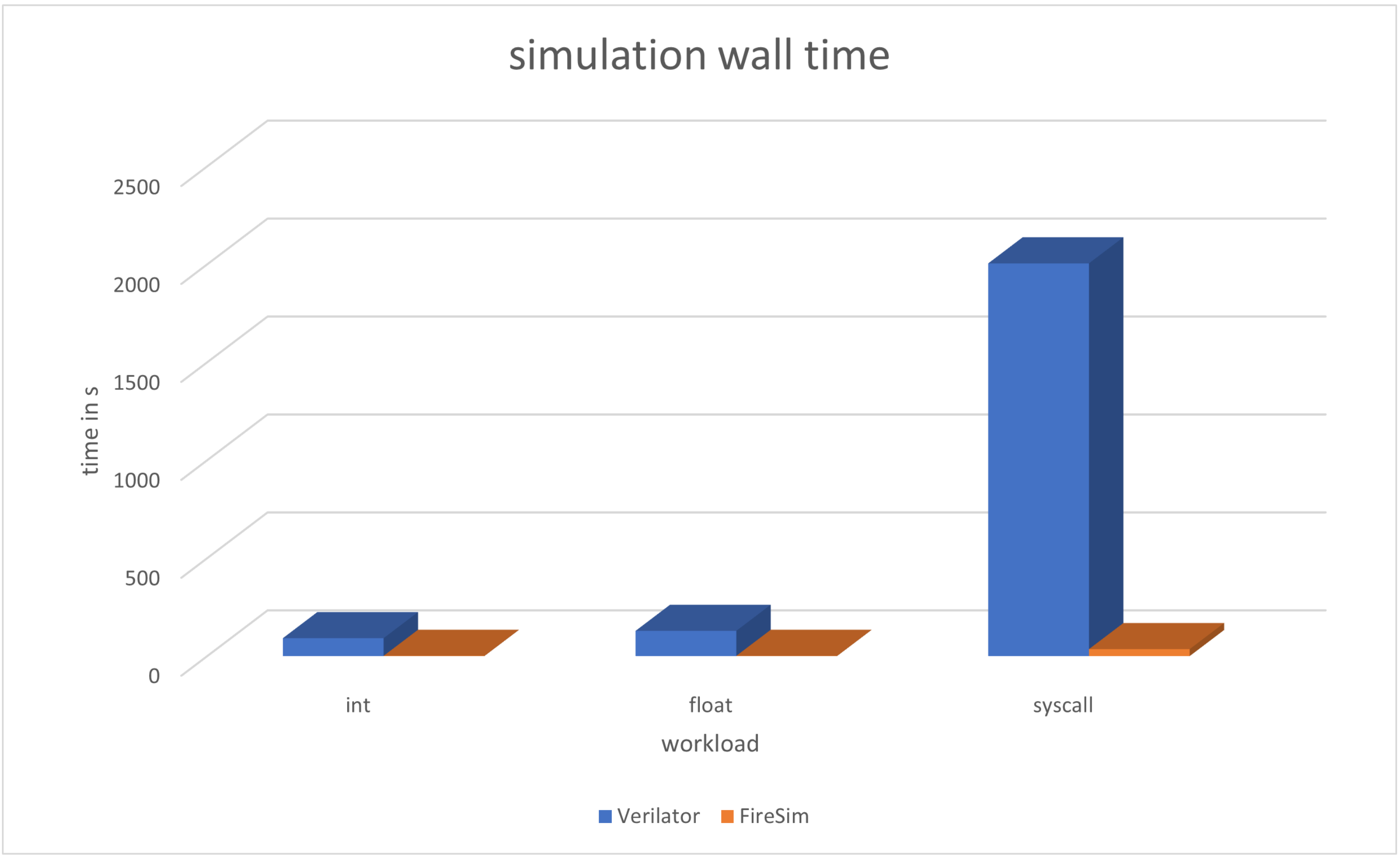
Therefore, our goal for the Hackathon was to implement FireSim, a FPGA accelerated RTL simulator, on Noctua 2 and execute bare metal binaries on hardware we synthesised using the Chipyard.

Typically, FireSim is run on AWS FPGA nodes or on-premise machines with supported FPGA cards with superuser permissions. Therefore, the system binaries that had to be executed with elevated permissions had to be identified and with the help of the FPGA advisers the permission requirement was circumvented.

Afterwards some of the simulation managing scripts were shortened to reduce the preparation time for single simulation runs.

The setup was then tested against software simulation with three different workloads. The first consisting mostly of integer math, second using primarily floating-point operations and the last with a combination of math operations and heavy syscall use. Resulting wall times are shown in the graph below. With simple workloads the accelerated FireSim simulation had a speedup of around 10 compared to the Verilator runtime. For the syscall heavy workload the speedup increased to 57, shortening the execution time from around 33 minutes to about 35 seconds.

Visualisation of simulation walltime with different workloads



Runtime Results