Implementation of the conjugate gradient algorithm for heterogeneous systems

Salvatore Calì[1], William Detmold[1], Grzegorz Korcyl[2], Piotr Korcyl[2], Phiala Shanahan[1]

Massachusetts Institute of Technology[1], Jagiellonian University in Kraków[2]

July 28, 2021

LATTICE 21
JULY 26-30 2021, ZOOM/GATHER@MIT

Table of contents

1. Motivation
2. Short introduction to SYCL/DPC++
3. Numerical details
4. Results
5. Conclusions

Motivation

Introduction

Linear systems

- One of the most common problems in computational science and linear algebra is to solve systems of linear equations.
- In lattice QCD calculations most of the computer time is typically spent in the numerical inversion of the Dirac-Wilson operator:

$$D^{AB}_{\alpha\beta}(n,m)\psi^{B}_{\beta}(m) = \eta^{A}_{\alpha}(n)$$

Iterative solvers

- Because of the very large size of the systems, trying to find the solution using a direct approach is not very practical:
  e.g.: for $V = 48^3 \times 96 \quad \Rightarrow \quad \text{size}(D^{-1}) = (V \times N_c \times N_s \times 2)^2|_{N_c=3, N_s=4} \approx 6.5 \times 10^{16}$
- We need to use iterative methods, like CG, BICG, BICG-STAB, etc., often combined with some preconditioning: AMG, Gauss-Seidel, Machine Learning (see Brian Xiao's poster), etc.

Heterogeneous Systems

- Heterogeneous systems may contain multiple types of computational devices.
  Common case: a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU)
- Collection of devices in a heterogeneous system may include: CPUs, GPUs, FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors), ASICs (Application-Specific Integrated Circuits), and AI chips (graph, neuromorphic, etc.).
- Each device has its pros/cons, so the next supercomputer architectures will probably combine goodness of different types of accelerators.
  e.g.: Cygnus at University of Tsukuba: first cluster equipped with CPUs-GPUs-FPGAs available for public use
- Dealing with multiple types of devices, with different architectures and characteristics ⇒ different programming and optimizations needs for each device. This is the motivation of the framework SYCL and its different implementations.
- Here, we make use of a SYCL implementation called DPC++ (Data Parallel C++), developed by INTEL.

Goals of this project

Single-Source code

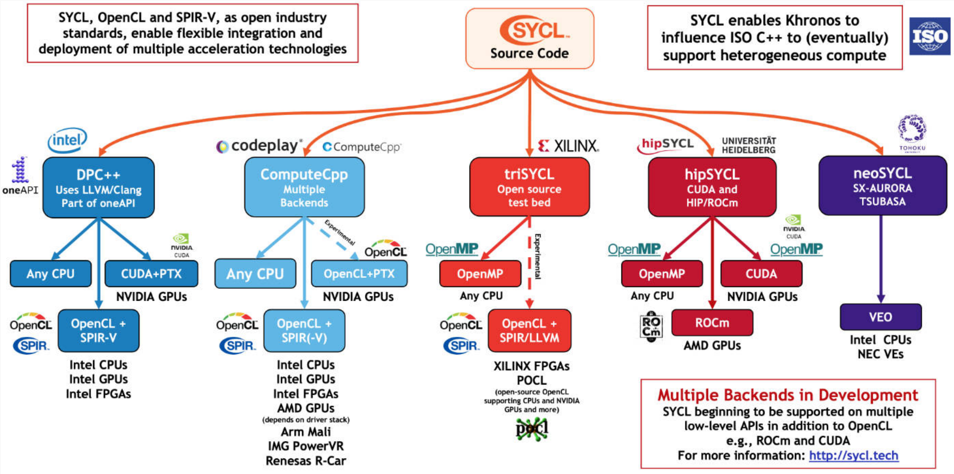
- One of the main attributes of SYCL/DPC++ is that programs can be single-source.
- The same source file can (it's not mandatory!) contain both:
    1. the code that defines the compute kernels to be executed on SYCL devices
    2. the host code that manages the execution of different compute kernels

Short project description

- We consider a single-node DPC++ implementation of one of the simplest methods to solve large and sparse linear systems, i.e. the Conjugate Gradient (CG) and we directly apply it to the Wilson-Dirac operator.
- This implementation is executed on different devices (CPUs, GPUs and FPGAs) and we test the performances.

Short introduction to SYCL/DPC++

SYCL implementations



SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute

intel
oneAPI

DPC++
Uses LLVM/Clang
Part of oneAPI

Any CPU

CUDA+PTX
NVIDIA GPUs

OpenCL + SPIR-V
Intel CPUs
Intel GPUs
Intel FPGAs

codeplay
ComputeCpp

ComputeCpp
Multiple Backends

Any CPU

Experimental

OpenCL+PTX
NVIDIA GPUs

OpenCL + SPIR(-V)
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car

XILINX

triSYCL
Open source test bed

OpenMP
Any CPU

Experimental

OpenCL + SPIR/LLVM
XILINX FPGAs
POCL
(open-source OpenCL supporting CPUs and NVIDIA GPUs and more)

hipSYCL
UNIVERSITÄT HEIDELBERG

hipSYCL
CUDA and HIP/ROCm

OpenMP
Any CPU

CUDA
NVIDIA GPUs

ROCm
AMD GPUs

TOHOKU

neoSYCL
SX-AURORA TSUBASA

VEO
Intel CPUs
NEC VEs

Multiple Backends in Development

SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL e.g., ROCm and CUDA

For more information: http://sycl.tech

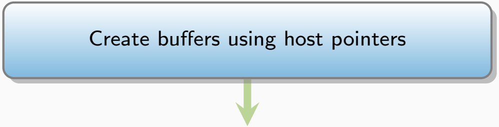
https://www.khronos.org/sycl/

Example of vector addition in DPC++

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main(){
        float A[1024], B[1024], C[1024];
        //Initialize the arrays A, B...
        {
                buffer<float, 1> bufA { A, range<1> {1024} };
                buffer<float, 1> bufB { B, range<1> {1024} };
                buffer<float, 1> bufC { C, range<1> {1024} };
                queue q;
                q.submit([&](handler& h) {
                        accessor A(bufA, h, read_only);
                        accessor B(bufB, h, read_only);
                        accessor C(bufC, h, write_only);
                        h.parallel_for(range<1> {1024}, [=](id<1> i) {
                                C[i] = A[i] + B[i];
                        });
                });
        }
        for (int i = 0; i < 1024; i++)
            std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

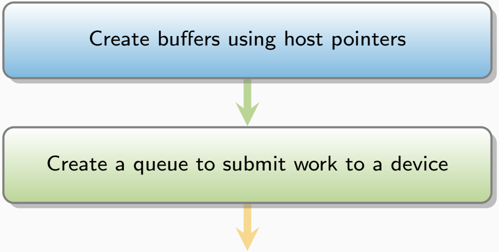
Example of vector addition in DPC++

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main(){
        float A[1024], B[1024], C[1024];
        //Initialize the arrays A, B...
        {
                buffer<float, 1> bufA { A, range<1> {1024} };
                buffer<float, 1> bufB { B, range<1> {1024} };
                buffer<float, 1> bufC { C, range<1> {1024} };
                queue q;
                q.submit([&](handler& h) {
                        accessor A(bufA, h, read_only);
                        accessor B(bufB, h, read_only);
                        accessor C(bufC, h, write_only);
                        h.parallel_for(range<1> {1024}, [=](id<1> i) {
                                C[i] = A[i] + B[i];
                        });
                });
        }
        for (int i = 0; i < 1024; i++)
            std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```


Create buffers using host pointers

Example of vector addition in DPC++

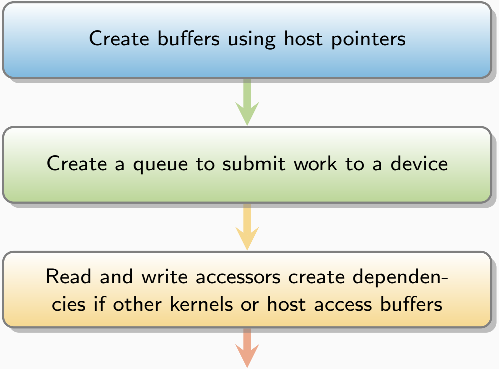
```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main(){
        float A[1024], B[1024], C[1024];
        //Initialize the arrays A, B...
        {
                buffer<float, 1> bufA { A, range<1> {1024} };
                buffer<float, 1> bufB { B, range<1> {1024} };
                buffer<float, 1> bufC { C, range<1> {1024} };
                queue q;
                q.submit([&](handler& h) {
                        accessor A(bufA, h, read_only);
                        accessor B(bufB, h, read_only);
                        accessor C(bufC, h, write_only);
                        h.parallel_for(range<1> {1024}, [=](id<1> i) {
                                C[i] = A[i] + B[i];
                        });
                });
        }
        for (int i = 0; i < 1024; i++)
            std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```


Create buffers using host pointers

Create a queue to submit work to a device

Example of vector addition in DPC++

```cpp
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main(){
        float A[1024], B[1024], C[1024];
        //Initialize the arrays A, B...
        {
                buffer<float, 1> bufA { A, range<1> {1024} };
                buffer<float, 1> bufB { B, range<1> {1024} };
                buffer<float, 1> bufC { C, range<1> {1024} };
                queue q;
                q.submit([&](handler& h) {
                        accessor A(bufA, h, read_only);
                        accessor B(bufB, h, read_only);
                        accessor C(bufC, h, write_only);
                        h.parallel_for(range<1> {1024}, [=](id<1> i) {
                                C[i] = A[i] + B[i];
                        });
                });
        }
        for (int i = 0; i < 1024; i++)
            std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```



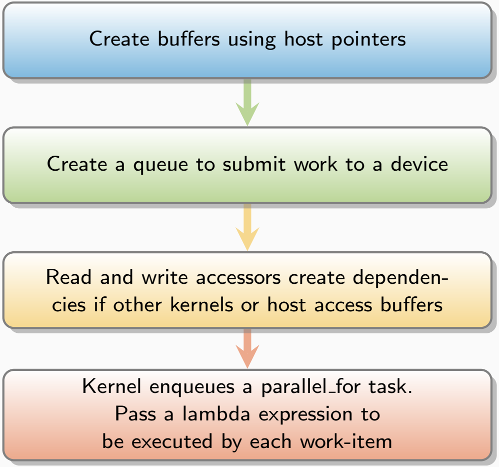
Create buffers using host pointers

Create a queue to submit work to a device

Read and write accessors create dependencies if other kernels or host access buffers

Example of vector addition in DPC++

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main(){
        float A[1024], B[1024], C[1024];
        //Initialize the arrays A, B...
        {
                buffer<float, 1> bufA { A, range<1> {1024} };
                buffer<float, 1> bufB { B, range<1> {1024} };
                buffer<float, 1> bufC { C, range<1> {1024} };
                queue q;
                q.submit([&](handler& h) {
                        accessor A(bufA, h, read_only);
                        accessor B(bufB, h, read_only);
                        accessor C(bufC, h, write_only);
                        h.parallel_for(range<1> {1024}, [=](id<1> i) {
                                C[i] = A[i] + B[i];
                        });
                });
        }
        for (int i = 0; i < 1024; i++)
            std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```



Create buffers using host pointers

Create a queue to submit work to a device

Read and write accessors create dependencies if other kernels or host access buffers

Kernel enqueues a parallel_for task.
Pass a lambda expression to
be executed by each work-item

Some types of kernel invocations

```
q.single_task([=](){
for(int i=0;i<N;i++){
// CODE THAT RUNS ON DEVICE
    }
});
```

```
q.parallel_for(range<1>{N},
[=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(
nd_range<1>{global_size, work_group_size},
[=](nd_item<1> item){
// CODE THAT RUNS ON DEVICE
});
```

Some types of kernel invocations

```
q.single_task([=](){
for(int i=0;i<N;i++){
// CODE THAT RUNS ON DEVICE
    }
});
```

```
q.parallel_for(range<1>{N},
[=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(
nd_range<1>{global_size, work_group_size},
[=](nd_item<1> item){
// CODE THAT RUNS ON DEVICE
});
```

- Single task: invoke a kernel function once on the target device.

Some types of kernel invocations

```
q.single_task([=](){
for(int i=0;i<N;i++){
// CODE THAT RUNS ON DEVICE
    }
});
```

```
q.parallel_for(range<1>{N},
[=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(
nd_range<1>{global_size, work_group_size},
[=](nd_item<1> item){
// CODE THAT RUNS ON DEVICE
});
```

- Single task: invoke a kernel function once on the target device.
- Basic parallel kernel: invoke a kernel function on each iteration of the task

Some types of kernel invocations

```
q.single_task([=](){
for(int i=0;i<N;i++){
// CODE THAT RUNS ON DEVICE
    }
});
```

```
q.parallel_for(range<1>{N},
[=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(
nd_range<1>{global_size, work_group_size},
[=](nd_item<1> item){
// CODE THAT RUNS ON DEVICE
});
```

- Single task: invoke a kernel function once on the target device.
- Basic parallel kernel: invoke a kernel function on each iteration of the task
- ND-range kernel: programmer can split the global size into smaller blocks, called work-groups.

Some types of kernel invocations

```
q.single_task([=](){
for(int i=0;i<N;i++){
// CODE THAT RUNS ON DEVICE
    }
});
```

```
q.parallel_for(range<1>{N},
[=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(
nd_range<1>{global_size, work_group_size},
[=](nd_item<1> item){
// CODE THAT RUNS ON DEVICE
});
```

- Single task: invoke a kernel function once on the target device.
- Basic parallel kernel: invoke a kernel function on each iteration of the task
- ND-range kernel: programmer can split the global size into smaller blocks, called work-groups.

Example:

- if GPU hardware consists of a bunch of compute units which has its own local memory, we can group executions so that each group executes on a single compute unit.

Numerical details

Numerical Setup

Some numerical details

- $V = [4^4, 6^4, 8^4, 10^4, 12^4, 14^4]$;
- $D =$ standard Wilson Dirac operator;
- to apply the CG algorithm, we solve for $DD^\dagger$ (hermitian) and then multiply the solution by $D^\dagger$;
- single precision (float);

Sparse Matrix format

- Coordinate format (easy to implement, but not optimal for the Dirac operator). D is stored using 3 arrays:
  1. values[N]: contains the value of the non-zero elements;
  2. row[N]: contains the row-index of the non-zero elements;
  3. col[N]: contains the column -index of the non-zero elements;
- SpMV pseudo-code:

```
for (k = 0; k < N; k = k + 1)
        output[row[k]] = result[row[k]] + values[k]*input[col[k]];
```

Algorithms and Hardware

Standard CG

```
$\psi \leftarrow \psi_0$;
$r \leftarrow \eta - DD^\dagger\psi$;
$p \leftarrow r$;
while $|r| > r_{min}$ do
    $r_{old} \leftarrow |r|$;
    $\alpha \leftarrow \frac{r_{old}}{\langle p|DD^\dagger|p\rangle}$;
    $\psi \leftarrow \psi + \alpha\psi$;
    $r \leftarrow r - \alpha DD^\dagger p$;
    $\beta \leftarrow |r|/r_{old}$;
    $p = r + \beta p$;
end
```

Kernels implemented for:

Sparse Matrix Vector Multiplication (SpMV)

Dot product

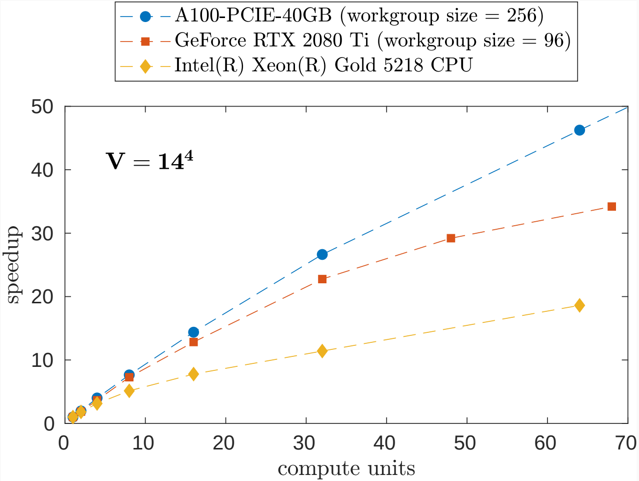
Vector addition/difference

Hardware tested

- CPUs:
  Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz
- GPUs:
  Nvidia GeForce RTX 2080 Ti
  A100-PCIE-40GB
- FPGAs: [https://devcloud.intel.com/oneapi/]
  Intel Arria 10
  Intel Stratix 10

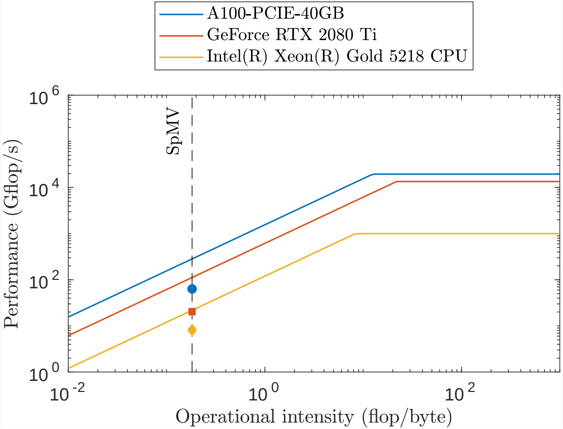
Results

CG speedup



- work group size = number of threads per compute unit
- using the same source code, we observe a better scaling for GPUs

SpMV performance - Roofline model



[S.William et al., Commun. ACM, 52:65–76]

Naive Roofline model

$$P = \min(\pi, \beta \times I)$$

- P = attainable performance;
- π = peak performance;
- β = peak bandwidth;
- I = operational intensity;

- Max performance reached on A100: $P \approx 65\text{GFlop/s}$
- Using the same source code, similar tests have been performed on FPGAs: $P \lesssim 1\text{GFlop/s}$

Conclusions

Summary and Outlook

Conclusions

- SYCL/DPC++ seems a potentially good framework to run lattice QCD codes on different devices.
- We observe acceptable performances on CPUs/GPUs.
- On FPGAs, we need to explore different kernels and optimizations for these devices: at the moment we seem far from a performing single-source code for CPU/GPU/FPGA.

Future plans

- Optimize code for Intel FPGAs and compare performances with other frameworks (OpenCL) and hardware (Xilinx cards). [For other results on FPGAs, see G. Korcyl's poster].
- Write a more specific algorithm for the Dirac operator.
- Combine CG with a preconditioner.

Thank you very much for your attention