Friedrich-Alexander-Universität
Erlangen-Nürnberg

Master Computational and Applied Mathematics

Programming Techniques For Supercomputers

Modelling 2D Steady-State Heat Equation

Project Report

Sharif Asumah Umar

Student ID: 23242255

Semuthu H. Don

Student ID: 23169793

Summer 2025

1 Introduction

We aim to model a 2D steady-tate heat dissipation on a rectangular plate . The set differential equation is solve on the discretized setting using the Finite Difference Method (FDM). The algebraic linear system of equations was solved with Conjugate Gradent (CG) and Preconditioned Conjugate Gradient (PCG) with symmetric Gauss-Seidel preconditioning. IN particular , this project aims to measure the roofline model performances. The project has two main parts: firstly, we clone our code from Github, perform some basic .and .. In the second part, we continue with parallelization using OpenMP.

2 Tasks Part 1

2.1 Code clone

We started by cloning the code from Github, which is stored under the file name PTfS-CAM-Project:

```
 ptfs300h@fritz3:~$ git clone https://github.com/RRZE-HPC/PTfS-CAM-Project.git\\
Cloning into 'PTfS-CAM-Project'...
remote: Enumerating objects: 51, done.
remote:  Counting objects: 100% (21/21), done.
remote: Compressing\quad objects: 100% (12/12), done.
remote: Total 51 \quad(delta 13), reused 9 (delta 9), pack-reused 30 (from 1)
Receiving objects: 100% (51/51), 52.71 KiB | 7.53 MiB/s, done.
Resolving deltas: 100% (17/17), done.
```

Listing 1: Clone Code

2.2 Makefile and LIkWID

Using the given Makefile we build the code and the LIKWID file was loaded :

```
    ptfs300h@f0319:~/PTfS-CAM-Project$ CXX=icpx make
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Iinclude -fp-model=precise -o test_test.o src/test.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Iinclude -fp-model=precise -o Grid_test.o src/Grid.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Iinclude -fp-model=precise -o PDE_test.o src/PDE.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Iinclude -fp-model=precise -o Solver_test.o src/Solver.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Iinclude -fp-model=precise -o timer_test.o src/timer.cpp
icpx -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -fp-model=precise -o test test_test.o Grid_test.o PDE_test.o
    Solver_test.o timer_test.o -lstdc++fs
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -Iinclude src/perf.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -Iinclude src/Grid.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -Iinclude src/PDE.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -Iinclude src/Solver.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -Iinclude src/timer.cpp
icpx -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare   -Wno-tautological-constant-compare -o perf perf.o Grid.o PDE.o
    Solver.o timer.o -lstdc++fs
```

Listing 2: Makefile Likwid

```
    ptfs300h@f0319:~/PTfS-CAM-Project$ LIKWID=on CXX=icpx make
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Iinclude -fp-model=
    precise -o test_test.o src/test.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Iinclude -fp-model=
    precise -o Grid_test.o src/Grid.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Iinclude -fp-model=
    precise -o PDE_test.o src/PDE.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Iinclude -fp-model=
    precise -o Solver_test.o src/Solver.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Iinclude -fp-model=
    precise -o timer_test.o src/timer.cpp
icpx -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -fp-model=precise -o
     test test_test.o Grid_test.o PDE_test.o Solver_test.o timer_test.o -lstdc++fs  -L/
    apps/likwid/5.4.1/lib -llikwid
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -Iinclude src/perf.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -Iinclude src/Grid.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -Iinclude src/PDE.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -Iinclude src/Solver.cpp
icpx -c -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -Iinclude src/timer.cpp
icpx -std=c++0x -Wall -Winline -Wshadow -W -O3 -qopenmp -xHOST -Wno-tautological-
    constant-compare -DLIKWID_PERFMON -I/apps/likwid/5.4.1/include  -Wno-tautological-
    constant-compare -o perf perf.o Grid.o PDE.o Solver.o timer.o -lstdc++fs  -L/apps/
    likwid/5.4.1/lib -llikwid
```

2.3 Test

We then run ./test for code correctness and ./perf to run our code using a grid size of 10000 x 10000

```
    ptfs300h@f0662:~/PTfS-CAM-Project$ ./test
TESTS started
There are 13 tests to pass
Test[ 1/13] dotProduct sum test success
Test[ 2/13] dotProduct product test success
Test[ 3/13] dotProduct product test success
Test[ 4/13] axpy success
Test[ 5/13] axpy success
Test[ 6/13] PDE::applyStencil success
Test[ 7/13] PDE::GSPreCon success
CG iterations for Sine problem = 724
Test[ 8/13] Solver::CG - sine residual check success
Test[ 9/13] Solver::CG - sine error check success
Writing solution file...
Directory created: "./results"
CG iterations for Linear problem = 724
Test[10/13] Solver::CG - linear residual check success
Writing solution file...
```

```
PCG iterations for Sine problem = 247
Test[11/13] Solver::PCG - sine residual check success
Test[12/13] Solver::PCG - sine error check success
Writing solution file...
PCG iterations for Linear problem = 251
Test[13/13] Solver::PCG - linear residual check success
Writing solution file...
Congrats !!!, You did it !
All tests passed
```

Listing 3: Test

```
    Running without Marker API. Activate Marker API with -m on commandline.
Total number of threads active = 72
TESTS started
There are 4 tests to pass
CG iterations = 20
Performance CG = 180.604095 [MLUP/s]
Test[ 1/ 4] Solver::CG - residual check success
Test[ 2/ 4] Solver::CG - error check success
PCG iterations = 20
Performance PCG = 84.056358 [MLUP/s]
Test[ 3/ 4] Solver::PCG - residual check success
Test[ 4/ 4] Solver::PCG - error check success
Congrats !!!, You did it !
All tests passed

---------------------------------------------------------------------------------
                                  Timing Summary

---------------------------------------------------------------------------------
       FUNCTION |             COUNTS |     TOTAL TIME (s) |        TIME/COUNT (s) |
---------------------------------------------------------------------------------
          AXPBY |                128 |    1.308538600e+01 |       1.022295781e-01 |
    DOT_PRODUCT |                109 |    8.506153000e+00 |       7.803810092e-02 |
  APPLY_STENCIL |                 45 |    4.525739000e+00 |       1.005719778e-01 |
             CG |                  1 |    1.107394600e+01 |       1.107394600e+01 |
     GS_PRE_CON |                 21 |    1.128325700e+01 |       5.372979524e-01 |
            PCG |                  1 |    2.379356000e+01 |       2.379356000e+01 |
---------------------------------------------------------------------------------
```

Listing 4: Perf

We found out that with 20 iterations CG had a performance of 180.604095 [MLUP/s] and the Performance of PCG was 84.056358 [MLUP/s]. in the next section we optimized by parallelizing our code using OpenMP to achieve a better performance.

3 Parallelization with OpenMP

This section demonstrates the performance of CG and PCG when tasked with parallelizing the Grid.cpp file and PDE.cpp file.

3.1 Grip.cpp

```
    //Calculates lhs[:] = a*x[:] + b*y[:]
void axpby(Grid *lhs, double a, Grid *x, double b, Grid *y, bool halo)
{
    START_TIMER(AXPBY);
#ifdef DEBUG
    assert((lhs->numGrids_y(true)==x->numGrids_y(true)) && (lhs->numGrids_x(true)==x->
        numGrids_x(true)));
    assert((y->numGrids_y(true)==x->numGrids_y(true)) && (y->numGrids_x(true)==x->
        numGrids_x(true)));
#endif

    int shift = halo?0:HALO;

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("AXPBY");
#endif
    #pragma omp parallel for schedule(static)
    for(int yIndex=shift; yIndex<lhs->numGrids_y(true)-shift; ++yIndex)
    {
        for(int xIndex=shift; xIndex<lhs->numGrids_x(true)-shift; ++xIndex)
        {
            (*lhs)(yIndex,xIndex) = (a*(*x)(yIndex,xIndex)) + (b*(*y)(yIndex,xIndex));
        }
    }
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_STOP("AXPBY");
#endif

    STOP_TIMER(AXPBY);
}
```

Listing 5: AXPBY Parallelized

```
    //Calculates lhs[:] = a*rhs[:]
void copy(Grid *lhs, double a, Grid *rhs, bool halo)
{
    START_TIMER(COPY);
#ifdef DEBUG
    assert((lhs->numGrids_y(true)==rhs->numGrids_y(true)) && (lhs->numGrids_x(true)==rhs
        ->numGrids_x(true)));
#endif

    int shift = halo?0:HALO;

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("COPY");
#endif
     #pragma omp parallel for schedule(static)
    for(int yIndex=shift; yIndex<lhs->numGrids_y(true)-shift; ++yIndex)
    {
        for(int xIndex=shift; xIndex<lhs->numGrids_x(true)-shift; ++xIndex)
        {
            (*lhs)(yIndex,xIndex) = a*(*rhs)(yIndex,xIndex);
        }
    }

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_STOP("COPY");
#endif


    STOP_TIMER(COPY);
```

Listing 6: COPY Parallelized

```
    //Calculate dot product of x and y
//i.e. ; res = x'*y
double dotProduct(Grid *x, Grid *y, bool halo)
{
    START_TIMER(DOT_PRODUCT);
#ifdef DEBUG
    assert((y->numGrids_y(true)==x->numGrids_y(true)) && (y->numGrids_x(true)==x->
        numGrids_x(true)));
#endif

    int shift = halo?0:HALO;

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("DOT_PRODUCT");
#endif
     #pragma omp parallel for reduction(+:dot_res)
    double dot_res = 0;
    for(int yIndex=shift; yIndex<x->numGrids_y(true)-shift; ++yIndex)
    {
        for(int xIndex=shift; xIndex<x->numGrids_x(true)-shift; ++xIndex)
        {
            dot_res += (*x)(yIndex,xIndex)*(*y)(yIndex,xIndex);
        }
    }

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_STOP("DOT_PRODUCT");
#endif


    STOP_TIMER(DOT_PRODUCT);
    return dot_res;
}
```

Listing 7: Dot Product Parallelized

The OpenMP parallelization is to speed up the computation, and the schedule(static) clause is used to distribute the iterations evenly among threads. The reduction($+ : dot - res$) clause is used to sum up the results from all threads. This is necessary because each thread computes a partial sum of the dot product.

3.2 PDE.cpp

We perform common parallelization on the Apply Stencil within the outer loop, as shown below. This, together with the inner loop pragma omp parallel for schedule(static), helps to distribute the work for loop iterations, resulting in easier and faster runs.

```
  //Applies stencil operation on to x
//i.e., lhs = A*x
void PDE::applyStencil(Grid* lhs, Grid* x)
{
    START_TIMER(APPLY_STENCIL);

#ifdef DEBUG
    assert((lhs->numGrids_y(true)==grids_y) && (lhs->numGrids_x(true)==grids_x));
    assert((x->numGrids_y(true)==grids_y) && (x->numGrids_x(true)==grids_x));
#endif
    const int xSize = numGrids_x(true);
    const int ySize = numGrids_y(true);

    const double w_x = 1.0/(h_x*h_x);
    const double w_y = 1.0/(h_y*h_y);
    const double w_c = 2.0*w_x + 2.0*w_y;

#pragma omp parallel
{
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("APPLY_STENCIL");
#endif
    #pragma omp parallel for schedule(static)
    for ( int j=1; j<ySize-1; ++j)
    {
        for ( int i=1; i<xSize-1; ++i)
        {
            (*lhs)(j,i) = w_c*(*x)(j,i) - w_y*((*x)(j+1,i) + (*x)(j-1,i)) - w_x*((*x)(j,
                i+1) + (*x)(j,i-1));
        }
    }

#ifdef LIKWID_PERFMON
    LIKWID_MARKER_STOP("APPLY_STENCIL");
#endif
}

    STOP_TIMER(APPLY_STENCIL);
}
```

Listing 8: Apply Stencil

We then proceeded to parallelize the GSPreCon, and it was observed that a data dependency exists; a direct parallel method, such as pragma omp parallel, cannot be used. There was a dependency on the forward and backward substitution. To solve this, we employed wavefront parallelization in conjunction with Gauss-Seidel parallelization. We add a pragma omp barrier at the end of each substitution. This ensures that the forward and backward substitutions are sequential thread updates, thereby guaranteeing that the thread completes its respective rows before proceeding to the next iteration of the outer loop.

```
    //GS preconditioning; solving for x: A*x=rhs
void PDE::GSPreCon(Grid* rhs, Grid *x)
{
    START_TIMER(GS_PRE_CON);

#ifdef DEBUG
    assert((rhs->numGrids_y(true)==grids_y) && (rhs->numGrids_x(true)==grids_x));
    assert((x->numGrids_y(true)==grids_y) && (x->numGrids_x(true)==grids_x));
#endif
    const int xSize = x->numGrids_x(true);
    const int ySize = x->numGrids_y(true);


    const double w_x = 1.0/(h_x*h_x);
    const double w_y = 1.0/(h_y*h_y);
    const double w_c = 1.0/static_cast<double>((2.0*w_x + 2.0*w_y));
// initialization before preconditioning parallelization
int nthreads, threadId, istart, j, k, iend, i, jj;

#pragma omp parallel private(nthreads, threadId, istart, j, iend, i, jj)
{
    nthreads = omp_get_num_threads();
    threadId = omp_get_thread_num();
    k=(xSize-2) / nthreads;
    // Calculate the start and end indices for each thread
    istart = k * threadId + 1;
    iend = (threadId == nthreads - 1) ? xSize - 2 : k * (threadId - 1) ;
    // Ensure that the last thread processes the last element


    //forward substitution
    for ( j=1; j<ySize-1 + nthreads - 1; ++j)
    { jj = j -threadId;
        if (jj >= 1 && jj <= ySize-1);
        for ( int i=istart; i<=iend; ++i)
        {
            (*x)(jj,i) = w_c*((*rhs)(jj,i) + (w_y*(*x)(jj-1,i) + w_x*(*x)(jj,i-1)));
        }
    }
    #pragma omp barrier
}
    //backward substitution

    for (j=ySize + nthreads - 3; j>0; --j)
    { jj = j - threadId;
        if (jj >= 1 && jj <= ySize-2){
            for ( int i=iend; i>=istart; --i)
            {
                (*x)(jj,i) = (*x)(jj,i) + w_c*(w_y*(*x)(jj+1,i) + w_x*(*x)(jj,i+1));
            }
        }
#pragma omp barrier
    }
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_STOP("GS_PRE_CON");
#endif
    STOP_TIMER(GS_PRE_CON);
}
```

Listing 9: Gauss-Seidal Parallelization(Wavefront parallelization)

```
    //Constructor
PDE::PDE(int len_x_, int len_y_, int grids_x_, int grids_y_):len_x(len_x_), len_y(len_y_
    ), grids_x(grids_x_+2*HALO), grids_y(grids_y_+2*HALO)
{
    h_x = static_cast<double>(len_x)/(grids_x-1.0);
    h_y = static_cast<double>(len_y)/(grids_y-1.0);

    initFunc = zeroFunc;

    //by default all boundary is Dirichlet
    // optimization using loop fusion
    for (int i=0; i<4; ++i)
    {
        boundary[i] = Dirichlet;
        boundaryFunc[i] = zeroFunc;
}
}
```

Listing 10: Loop Fusion

We needed to perform loop fusion on the initialization of the boundary, with no data dependencies.

3.2.1 New performance after OpenMP

```
    All tests passed
ptfs300h@f0365:~/PTfS-CAM-Project$ ./perf 2000 20000
Total number of threads active = 72
TESTS started
There are 4 tests to pass
CG iterations = 20
Performance CG = 563.583129 [MLUP/s]
Test[ 1/ 4] Solver::CG - residual check success
Test[ 2/ 4] Solver::CG - error check success
PCG iterations = 20
Performance PCG = 341.509501 [MLUP/s]
Test[ 3/ 4] Solver::PCG - residual check success
Test[ 4/ 4] Solver::PCG - error check success
Congrats !!!, You did it !
All tests passed

-----------------------------------------------------------------------------------
                                   Timing Summary

-----------------------------------------------------------------------------------
       FUNCTION |             COUNTS |     TOTAL TIME (s) |        TIME/COUNT (s) |
-----------------------------------------------------------------------------------
          AXPBY |                128 |    1.479398012e+00 |       1.155779697e-02 |
    DOT_PRODUCT |                109 |    8.165471554e-01 |       7.491258306e-03 |
  APPLY_STENCIL |                 45 |    4.609785080e-01 |       1.024396684e-02 |
             CG |                  1 |    1.419488907e+00 |       1.419488907e+00 |
      GS_PRE_CON |                 21 |    1.291566610e+00 |       6.150317192e-02 |
            PCG |                  1 |    2.342540979e+00 |       2.342540979e+00 |
-----------------------------------------------------------------------------------
```

Listing 11: Perf

The new performance after OpenMP parallelization significantly improved the performance of the CP and PCG solvers; thus, the CG achieved a jump from 180.6 MLUP/s to 563.58 MLUP/s, and the PCG performance increased from 84 MLUP/s to 341.51 MLUP/s.

4 Roofline calculation for CG and PCG

In this section, we examined the calculations and analysis of the roofline model predictions and performance for CP and PCG on a one ccNUMA domain of 18 cores of Fritz. To achieve this, we first listed our knowns and the computations required for the three grid sizes. The Fritz system has an Ice Lake core with AVX512 instruction sets.

4.1 Solvers with different kernels and grid size performance

Considering the grid sizes, a) Grid 1: 2000 x 20,000 b) Grid 2: 2000 x 20,000 and c) Grid 3: 1000 x 4000000

we need to calculate computational intensity,I = FLOPS/ Bytes. and the maximum achievable performance P_{max} in LUP/s for all kernels individually and then combine them using chaoned roofline model for CG and PCG to be able to estimate the roofline predictions.

The roofline model is given as:

$P = min(P_{max}, I \times Bs)$

where:

P_{max} is the maximum performance.

I is the computational intensity.

Bs is the bandwidth.

we know that

$$P_{max} = n_{core} \times \frac{FLOPs}{cycles} \times n_{SIMD} \times f$$

and Also given the memory bandwidth and the Cache size for one ccNUMA to be $b_s = 82GB/s$ and 54MB , Since the peak performance and the memory bandwidth are decided by the hardware, we calculated the computational intensity I of PCG solver, CG solvers, and the time of every kernel through the following formula below

$$I = \frac{N}{v}$$

$$T = \frac{\text{Total Flops}}{P}$$

```
    START_TIMER(CG);

    while( (iter<niter) && (alpha_0>tol*tol) && (IS_VALID(alpha_0)) )
    {
        pde->applyStencil(v,p);
        lambda =  alpha_0/dotProduct(v,p);
        //Update x
        axpby(x, 1.0, x, lambda, p);
        //Update r
        axpby(r, 1.0, r, -lambda, v);
        alpha_1 = dotProduct(r,r);
        //Update p
        axpby(p, 1.0, r, alpha_1/alpha_0, p);
        alpha_0 = alpha_1;
#ifdef DEBUG
        printf("iter = %d, res = %.15e\n", iter, alpha_0);
#endif
```

```
        ++iter;
    }

    STOP_TIMER(CG);
```

```
    START_TIMER(PCG);

    while( (iter<niter) && (res_norm_sq>tol*tol) && (IS_VALID(res_norm_sq)) )
    {
        pde->applyStencil(v,p);
        lambda =  alpha_0/dotProduct(v,p);
        //Update x
        axpby(x, 1.0, x, lambda, p);
        //Update r
        axpby(r, 1.0, r, -lambda, v);
        res_norm_sq = dotProduct(r,r);
        //Update z
        pde->GSPreCon(r, z);
        alpha_1 = dotProduct(r,z);
        //Update p
        axpby(p, 1.0, z, alpha_1/alpha_0, p);
        alpha_0 = alpha_1;

#ifdef DEBUG
        printf("iter = %d, res = %.15e\n", iter, res_norm_sq);
#endif
        ++iter;
    }

    STOP_TIMER(PCG);
```

Listing 12: PCG Solver

The algorithms above gives us the kernels in each solver. We proceeded with taking the three Grid cases for each kernel,

We first looked at the topology of our Fritz system to know our cache capacity. It has two sockets with 36 cores each , while every CPU has it local L1 cache (48KB) and L2 cache(1.25MB) they share L3 cache(54MB). We need only 18 cores for one ccNUMA so we get 27MB for our main memory.

CG

(a)Applied stencil(v,p)

Case 1 Grid 1 2000 by 20000

$$\begin{aligned}\text{Layer condition} &= \text{nthreads} \times 3 \times \text{xSize} \times 8\,\text{Bytes} < 27\,\text{MB}\\ &= 18 \times 3 \times 20000 \times 8\,\text{Bytes} < 27\,\text{MB}\\ &= 8.64\,\text{MB} < 27\,\text{MB}\end{aligned}$$

This implies that our layer condition is satisfied, and we can fit data in the L3 Cache. In addition to the one load, we need one more load, write-allocation. The first load is the only element on the right-hand side of the stencil operator kernel per new lattice update that needs to be loaded from the main memory (the element in the upper row). Also, there will be one store. So in summary, 2 LOADS and 1 STORE.

Performance of ApplyStencil, we will calculate the intensity and P_max, to apply the following The formula of the performance for the roofline model:

$$P = min(P_max, I \times b_s)$$

We assume a memory bandwidth of 82 GB/s. The data transfer is 2 LOADs and 1 STORE, and we observe that the bottleneck is ADD according to the . We have 7 Flops(4 ADD and 3 MULT).

Table 1: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD	ADD	MUL	MUL			LD		ST
ADD	ADD	MUL					LD	

Then,

Arithmetic Intensity (I)

$$I = \frac{\text{Flops}}{\text{Bytes}} = \frac{7}{3 \times 8} = \frac{7}{24} \frac{\text{Flops}}{\text{Bytes}}$$

Given the bandwidth $b_s = 82\,\text{GB/s}$, the bandwidth-bound performance is:

$$I \times b_s = \frac{7}{24} \frac{\text{Flops}}{\text{Bytes}} \times 82 \frac{\text{GB}}{\text{s}} = \frac{574}{24}\,\text{GFlops/s} \approx 23.92\,\text{GFlops/s}$$

Peak Performance (P_{max}) For the `ApplyStencil` operation, the theoretical peak performance is:

$$P_{\text{max}} = \text{No. of cores} \times \frac{\text{FLOPs}}{\text{cycle}} \times n_{\text{SIMD}} \times f$$

$$P_{\text{max}} = 18 \times \frac{7}{2} \times 8 \times 2.2\,\text{GHz} = 1108.8\,\text{GFlops/s}$$

Roofline Model

The achievable performance P is the minimum of P_{max} and the bandwidth-bound performance:

$$\begin{aligned} P &= \min\left(P_{\text{max}}, I \times b_s\right) \\ &= \min\left(1108.8\,\text{GFlops/s}, 23.92\,\text{GFlops/s}\right) \\ &= 23.92\,\text{GFlops/s} \end{aligned}$$

Case 2 Grid 2 20000 by 2000

$$\begin{aligned} \text{Layer condition} &= \text{nthreads} \times 3 \times \text{xSize} \times 8\,\text{Bytes} < 27\,\text{MB} \\ &= 18 \times 3 \times 2000 \times 8\,\text{Bytes} < 27\,\text{MB} \\ &= 0.864\,\text{MB} < 27\,\text{MB} \end{aligned}$$

This implies also that our layer condition is satisfied, and we can fit data in the L3 cache. In addition to the one load, we need one more load, write-allocation. The first load is the only element on the right-hand side of the stencil operator kernel per new lattice update that needs to be loaded from the main memory (the element in the upper row). Also, there will be one store. So in summary, 2 LOADS and 1 STORE.

Performance of ApplyStencil, we will calculate the intensity and P_max, to apply the following The formula of the performance for the roofline model:

$$P = min(P_max, I \times b_s)$$

We assume a memory bandwidth of 82 GB/s. The data transfer is 2 LOADs and 1 STORE, and we observe that the bottleneck is ADD. We have 7 Flops(4 ADD and 3 MULT). Then,

Arithmetic Intensity (I)

$$I = \frac{\text{Flops}}{\text{Bytes}} = \frac{7}{3 \times 8} = \frac{7}{24}\,\frac{\text{Flops}}{\text{Bytes}}$$

Given the bandwidth $b_s = 82\,\text{GB/s}$, the bandwidth-bound performance is:

$$I \times b_s = \frac{7}{24}\,\frac{\text{Flops}}{\text{Bytes}} \times 82\,\frac{\text{GB}}{\text{s}} = \frac{574}{24}\,\text{GFlops/s} \approx 23.92\,\text{GFlops/s}$$

Peak Performance ($P_{\mathbf{max}}$) For the `ApplyStencil` operation, the theoretical peak performance is:

$$P_{\text{max}} = \text{No. of cores} \times \frac{\text{FLOPs}}{\text{cycle}} \times n_{\text{SIMD}} \times f$$

$$P_{\text{max}} = 18 \times \frac{7}{2} \times 8 \times 2.2\,\text{GHz} = 1108.8\,\text{GFlops/s}$$

Roofline Model

The achievable performance P is the minimum of P_{max} and the bandwidth-bound performance:

$$\begin{aligned} P &= \min\left(P_{\text{max}}, I \times b_s\right) \\ &= \min\left(1108.8\,\text{GFlops/s}, 23.92\,\text{GFlops/s}\right) \\ &= 23.92\,\text{GFlops/s} \end{aligned}$$

Case 3 Grid 3 1000 by 400000

$$\begin{aligned} \text{Layer condition} &= \text{nthreads} \times 3 \times \text{xSize} \times 8\,\text{Bytes} < 27\,\text{MB} \\ &= 18 \times 3 \times 400000 \times 8\,\text{Bytes} < 27\,\text{MB} \\ &= 172.8\,\text{MB} > 27\,\text{MB} \end{aligned}$$

This implies that the layer condition is not satisfied, and we can not fit the data in the L3 cache. Therefore, three elements on the right-hand side of the stencil kernel (top, right, and bottom elements) per new lattice update need to be loaded from the main memory. Moreover, we need one more load (write allocate) and one store . So in summary, 4 LOADS and 1 STORE.

Performance of ApplyStencil, we will calculate the intensity and P_max, to apply the following The formula of the performance for the roofline model:

$$P = min(P_{max}, I \times b_s)$$

We assume a memory bandwidth of 82 GB/s. The data transfer is 4 LOADs and 1 STORE, and we observe that the bottleneck is ADD. We have 7 Flops(4 ADD and 3 MULT).

Table 2: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD	ADD	MUL	MUL			LD	LD	ST
ADD	ADD	MUL				LD	LD	

Then,

Arithmetic Intensity (I)

$$I = \frac{\text{Flops}}{\text{Bytes}} = \frac{7}{5 \times 8} = \frac{7}{40}\,\frac{\text{Flops}}{\text{Bytes}}$$

Given the bandwidth $b_s = 82\,\text{GB/s}$, the bandwidth-bound performance is:

$$I \times b_s = \frac{7}{40}\,\frac{\text{Flops}}{\text{Bytes}} \times 82\,\frac{\text{GB}}{\text{s}} = \frac{574}{40}\,\text{GFlops/s} \approx 14.35\,\text{GFlops/s}$$

Peak Performance (P_{max})

For the `ApplyStencil` operation, the theoretical peak performance is:

$$P_{\text{max}} = \text{No. of cores} \times \frac{\text{FLOPs}}{\text{cycle}} \times n_{\text{SIMD}} \times f$$

$$P_{\text{max}} = 18 \times \frac{7}{2} \times 8 \times 2.2\,\text{GHz} = 1108.8\,\text{GFlops/s}$$

Roofline Model

The achievable performance P is the minimum of P_{max} and the bandwidth-bound performance:

$$\begin{aligned} P &= \min\left(P_{\text{max}}, I \times b_s\right) \\ &= \min\left(1108.8\,\text{GFlops/s}, 14.35\,\text{GFlops/s}\right) \\ &= 14.35\,\text{GFlops/s} \end{aligned}$$

Time Required for Execution of ApplyStencil

We compute the time required to execute the kernel using the following formula:

$$T_{\text{kernel}} = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P_{\text{kernel}} \times 10^9}$$

Case 1: Grid 1 2000 x 20000

$$T = \frac{7 \times 2000 \times 20000}{23.92 \times 10^9} = \frac{280{,}000{,}000}{23.92 \times 10^9} = 11.71 \times 10^{-3}\,\text{s} = 11.71\,\text{ms}$$

Case 2: Grid 2 20000 x 2000

$$T = \frac{7 \times 20000 \times 2000}{23.92 \times 10^9} = \frac{280{,}000{,}000}{23.92 \times 10^9} = 11.71 \times 10^{-3}\,\text{s} = 11.71\,\text{ms}$$

Case 3: Grid 3 1000 x 400000

$$T = \frac{7 \times 1000 \times 400000}{14.35 \times 10^9} = \frac{2{,}800{,}000{,}000}{14.35 \times 10^9} = 195.12 \times 10^{-3}\,\text{s} = 195.12\,\text{ms}$$

(b)Dot product (v,p)

GRID 1 and GRID 2 and GRID 3 Similarly, for the dot product, we have to compute for all grid cases; we skip the layer conditions as it is the same for the dot product. However, we have 1 LD and 2 Flops (add and mult) from the dot product of different variable kernels.

Table 3: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD		MUL				LD	LD	

following the same formula we compute the computational intensity as:

Arithmetic Intensity (I)

$$I = \frac{2\text{Flops}}{2 * 8\text{Bytes}} = \frac{1\text{Flop}}{8\text{Bytes}}$$

We already know the value of B_s, so we can compute $I * B_s$:

$$I \times b_s = \frac{1}{8}\,\frac{\text{Flops}}{\text{Bytes}} \times 82\,\frac{\text{GB}}{\text{s}} = \frac{82}{8}\,\text{GFlops/s} \approx 10.25\,\text{GFlops/s}$$

Now, we can see that we got 2 Flops per 1 cycle from the table 3. Also, we conclude from the table 3 that this kernel performs 1 AVX iteration per 1 cycle. Here, the Bottleneck is LD. given

$$P_{max} = n_{core} * \frac{\text{Flops}}{\text{cycles}} * n_{SIMD} * f$$

so,

$$P_{max} = 18 * \frac{2}{1} * 8 * 2.2 = 633.6 \text{ GFlops/s}$$

Now,

$$P = \min(P_{max}, I * B_s)$$

$$= \min(633.6 \text{ GFlops/s}, 10.25 \text{ GFlops/s})$$

$$= 10.25 \text{ GFlops/s}$$

Time Required for Execution of dot product(v,p)

We compute the time required to execute the kernel using the following formula:

$$T_{\text{kernel}} = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P_{\text{kernel}} \times 10^9}$$

Case 1: Grid 1 2000 $\times$ 20000

$$T = \frac{2 \times 2000 \times 20000}{10.25 \times 10^9} = \frac{80{,}000{,}000}{10.25 \times 10^9} = 7.80 \times 10^{-3}\,\text{s} = 7.80\,\text{ms}$$

Case 2: Grid 2 20000 $\times$ 2000

$$T = \frac{2 \times 20000 \times 2000}{10.25 \times 10^9} = \frac{80{,}000{,}000}{10.25 \times 10^9} = 7.80 \times 10^{-3}\,\text{s} = 7.80\,\text{ms}$$

Case 3: Grid 3 1000 $\times$ 400000

$$T = \frac{2 \times 1000 \times 400000}{10.25 \times 10^9} = \frac{800{,}000{,}000}{10.25 \times 10^9} = 78.05 \times 10^{-3}\,\text{s} = 78.05\,\text{ms}$$

(c)Dotproduct(r,r)

Here, we also have the dot product, but for the same vectors (r,r), also called the pointer aliasing; therefore, we get one load, and 2 Flops on the RHS side. we considered the three GRID sizes.

Table 4: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD	ADD_{2nd}	MUL_{2nd}	MUL			LD	LD_{2nd}	

Arithmetic Intensity (I)

So, the computational intensity can be given as:

$$I = \frac{2\text{Flops}}{1 * 8\text{Bytes}} = \frac{1\text{Flop}}{4\text{Bytes}} \tag{1}$$

We already know the value of B_s, so we can compute $I * B_s$:

$$I * B_s = \frac{1\text{Flop}}{4\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{82\text{ GFlops}}{4s} \approx 20.5\text{GFlops/s}$$

Now, we can see that we got 2 Flops per 1 cycle and 4 Flops per 2 cycles from the table 4. Also, we conclude from the Table 4 that this kernel performs 2 AVX iterations per 1 cycle. Here, the Bottleneck is LD, ADD and MUL.

$$P_{max} = n_{core} * \frac{\text{Flops}}{\text{cycles}} * n_{SIMD} * f$$

$$P_{max} = 18 \times \frac{4}{1} \times 8 \times 2.2 = 1267.2\text{ GFlops/s} \tag{2}$$

Now,

$$P = \min(P_{max}, I * B_s)$$

$$= \min(1267.2\text{GFlops/s}, 20.5\text{GFlops/s})$$

$$= 20.5\text{GFlops/s}$$

Time Required for Execution of Dot Product (r, r)

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{3}$$

Case 1: Grid 1 2000 x 20000

$$T = \frac{2 \times 2000 \times 20000}{20.5 \times 10^9} = \frac{80{,}000{,}000}{20.5 \times 10^9} = 3.902 \times 10^{-3}\,\text{s} = 3.902\,\text{ms}$$

Case 2: Grid 2 20000 x 2000

$$T = \frac{2 \times 20000 \times 2000}{20.5 \times 10^9} = \frac{80{,}000{,}000}{20.5 \times 10^9} = 3.902 \times 10^{-3}\,\text{s} = 3.902\,\text{ms}$$

Case 3: Grid 3 1000 x 400000

$$T = \frac{2 \times 1000 \times 400000}{20.5 \times 10^9} = \frac{800{,}000{,}000}{20.5 \times 10^9} = 39.02 \times 10^{-3}\,\text{s} = 39.02\,\text{ms}$$

(d) axpby(x, 1.0, x, lambda, p)

Here, we have 2 loads and 1 Store on the LHS side and on the RHS side, we have 1 ADD and 2 MUL, 3 Flops.

Table 5: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD		MUL	MUL			LD	LD	ST

Arithmetic Intensity (I)
So, the computational intensity can be given as:

$$I = \frac{3 \text{ Flops}}{3 * 8\text{Bytes}} = \frac{1 \text{ Flop}}{8\text{Bytes}} \tag{4}$$

We already know the value of B_s, so we can compute $I * B_s$:

$$I * B_s = \frac{1\text{Flop}}{8\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{82 \text{ GFlops}}{8s} \approx 10.25 \tag{5}$$

Peak Performance $(P_{\textbf{max}})$

$$P_{max} = n_{core} * \frac{\text{Flops}}{\text{cycles}} * n_{SIMD} * f$$

$$P_{max} = 18 \times \frac{3}{1} \times 8 \times 2.2 = 950.4 \text{ GFlops/s}$$

Now,

$$P = \min(P_{max}, I * B_s) \tag{6}$$

$$= \min(950.4 \text{ GFlops/s}, 10.25 \text{ GFlops/s})$$

$$= 10.25 \text{ GFlops/s}$$

Time Required for Execution of axpby(x, 1.0, x, lambda, v)

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{7}$$

Case 1: Grid 1 2000 $\times$ 20000

$$T = \frac{3 \times 2000 \times 20000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 2: Grid 2 20000 $\times$ 2000

$$T = \frac{3 \times 20000 \times 2000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 3: Grid 3 1000 $\times$ 400000

$$T = \frac{3 \times 1000 \times 400000}{10.25 \times 10^9} = \frac{1{,}200{,}000{,}000}{10.25 \times 10^9} = 117.07 \times 10^{-3}\,\text{s} = 117.07\,\text{ms}$$

(e)axpby(r, 1.0, r, -lambda, v)

Similarly like above, here, we have two loads and 1 Store on the LHS side and On the RHS side, we have 1 ADD and 2 MUL, 3 Flops.

Table 6: Instruction Distribution

ADD	ADD	MUL	MUL	FMA	FMA	LD	LD	ST
ADD		MUL	MUL			LD	LD	ST

Arithmetic Intensity (I)
So, the computational intensity can be given as:

$$I = \frac{3 \text{ Flops}}{3 * 8\text{Bytes}} = \frac{1 \text{ Flop}}{8\text{Bytes}} \tag{8}$$

We already know the value of B_s, so we can compute $I * B_s$:

$$I * B_s = \frac{1\text{Flop}}{8\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{82 \text{ GFlops}}{8s} \approx 10.25 \tag{9}$$

Peak Performance ($P_{\mathbf{max}}$)

$$P_{max} = n_{core} * \frac{\text{Flops}}{\text{cycles}} * n_{SIMD} * f$$

$$P_{max} = 18 \times \frac{3}{1} \times 8 \times 2.2 = 950.4 \text{ GFlops/s}$$

Now,

$$P = \min(P_{max}, I * B_s) \tag{10}$$

$$= \min(950.4 \text{ GFlops/s}, 10.25 \text{ GFlops/s})$$

$$= 10.25 \text{ GFlops/s}$$

Time Required for Execution of axpby(r, 1.0, r, -lambda, v)

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{11}$$

Case 1: Grid 1 2000 $\times$ 20000

$$T = \frac{3 \times 2000 \times 20000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 2: Grid 2 20000 $\times$ 2000

$$T = \frac{3 \times 20000 \times 2000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 3: Grid 3 1000 $\times$ 400000

$$T = \frac{3 \times 1000 \times 400000}{10.25 \times 10^9} = \frac{1{,}200{,}000{,}000}{10.25 \times 10^9} = 117.07 \times 10^{-3}\,\text{s} = 117.07\,\text{ms}$$

(e) axpby(p, 1.0, r, alpha$_1$ $/alpha_0, v)$

similarly like above d and e , Here, we have 2 loads and 1 Store on the LHS side and on the RHS side, we have 1 ADD and 2 MUL, 3 Flops.

Table 7: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD		MUL	MUL			LD	LD	ST

Arithmetic Intensity (I)
So, the computational intensity can be given as:

$$I = \frac{3 \text{ Flops}}{3 * 8\text{Bytes}} = \frac{1 \text{ Flop}}{8\text{Bytes}} \tag{12}$$

We already know the value of B_s, so we can compute $I * B_s$:

$$I * B_s = \frac{1\text{Flop}}{8\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{82 \text{ GFlops}}{8s} \approx 10.25 \tag{13}$$

Peak Performance ($P_{\max}$)

$$P_{max} = n_{core} * \frac{\text{Flops}}{\text{cycles}} * n_{SIMD} * f$$

$$P_{max} = 18 \times \frac{3}{1} \times 8 \times 2.2 = 950.4 \text{ GFlops/s}$$

Now,

$$P = \min(P_{max}, I * B_s) \tag{14}$$

$$= \min(950.4 \text{ GFlops/s}, 10.25 \text{ GFlops/s})$$

$$= 10.25 \text{ GFlops/s}$$

Time Required for Execution of axpby(p, 1.0, r, alpha$_1/alpha_0, v)$

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{15}$$

Case 1: Grid 1 2000 $\times$ 20000

$$T = \frac{3 \times 2000 \times 20000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 2: Grid 2 20000 $\times$ 2000

$$T = \frac{3 \times 20000 \times 2000}{10.25 \times 10^9} = \frac{120{,}000{,}000}{10.25 \times 10^9} = 11.707 \times 10^{-3}\,\text{s} = 11.707\,\text{ms}$$

Case 3: Grid 3 1000 $\times$ 400000

$$T = \frac{3 \times 1000 \times 400000}{10.25 \times 10^9} = \frac{1{,}200{,}000{,}000}{10.25 \times 10^9} = 117.07 \times 10^{-3}\,\text{s} = 117.07\,\text{ms}$$

Summary Outline of Different Grids and Performances for the CG solver

4.1.1 Expected chain roofline performance of GRID 1 and GRID 2

Given the total time $T_{total} = 58.533 \quad ms)$

$$P = \frac{2000 \times 20000 LUP}{58.533 ms} \approx 683 MLUP/s$$

Table 8: Performance Comparison Across Different Grids

kernels	$P_{\max}$ (GFlops/s)			Time Taken (ms)		
	Grid 1	Grid 2	Grid 3	Grid 1	Grid 2	Grid 3
applyStencil	23.92	23.92	14.35	11.71	11.71	195.12
Dotproduct(v,p)	10.25	10.25	10.25	7.8	7.8	78.05
Dotproduct(r,r)	20.5	20.5	20.5	3.902	3.902	39.02
axpby(x, 1.0, x, lambda, p)	10.25	10.25	10.25	11.707	11.707	117.07
axpby(x, 1.0, x, -lambda, v)	10.25	10.25	10.25	11.707	11.707	117.07
axpby(p, 1.0, r, α_1/α_0, p)	10.25	10.25	10.25	11.707	11.707	117.07
Total	–	–	–	**58.533**	**58.533**	**663.4**

4.1.2 Expected chain roofline performance of GRID 3

$T_{total} = 663.4ms$

$$P = \frac{1000 \times 400000LUP}{663.4ms} \approx 603MLUP/s$$

5 PCG

This solver has the same kernels as the CP solver except the GSPrecon , so we computed the computational intensity and the performance for both the forward substitution and backward substitution.

5.1 (a) GSPrecon forward substitution

In the forward substitution, we get 1LD and 1ST on LHS side, and 1LD, 3MUL and 2ADD , 5 Flops. So here, the bottleneck would be MUL and LD.

Table 9: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD	ADD	MUL	MUL			LD	LD	ST
		MUL						

Arithmetic Intensity (I) for GRID1 and GRID 2
So, the computational intensity can be given as:

$$I = \frac{5\text{Flops}}{3 * 8\text{Bytes}} = \frac{5\text{Flops}}{24\text{Bytes}}$$

$$I * B_s = \frac{5\text{Flops}}{24\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{5 * 41\text{GFlops}}{12s} \approx 17.08\text{GFlops/s}$$

$$P_{max} = 18 \times \frac{5}{2} \times 8 \times 2.2 = 792 \text{ GFlops/s}$$

$$P = \min(792 \text{ GFlops/s}, 17.08 \text{ GFlops/s}) = 17.08 \text{ GFlops/s}$$

Arithmetic Intensity (I) for GRID 3 We saw from the previous kernels of CG that the layer condition is not satisfied for GRID 3, which implies here for the GSPrecon forward substation that an additional load is required, meaning 3 LD + 1 ST, then 5 Flops. Then we get

$$I = \frac{5\text{Flops}}{4 * 8\text{Bytes}} = \frac{5\text{Flops}}{32\text{Bytes}}$$

$$I * B_s = \frac{5\text{Flops}}{32\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{5 * 82\text{GFlops}}{32s} \approx 12.81\text{GFlops/s}$$

$$P_{max} = 18 \times \frac{5}{2} \times 8 \times 2.2 = 792 \text{ GFlops/s}$$

$$P = \min(792 \text{ GFlops/s}, 12.81 \text{ GFlops/s}) = 12.81 \text{ GFlops/s}$$

Time Required for Execution of GSPrecon Forward Substitution

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{16}$$

Case 1: Grid 1 2000 × 20000

$$T = \frac{5 \times 2000 \times 20000}{17.08 \times 10^9} = \frac{200{,}000{,}000}{17.08 \times 10^9} = 11.71 \times 10^{-3}\,\text{s} = 11.71\,\text{ms}$$

Case 2: Grid 2 20000 × 2000

$$T = \frac{5 \times 20000 \times 2000}{17.08 \times 10^9} = \frac{200{,}000{,}000}{17.08 \times 10^9} = 11.71 \times 10^{-3}\,\text{s} = 11.71\,\text{ms}$$

Case 3: Grid 3 1000 × 400000

$$T = \frac{5 \times 1000 \times 400000}{12.81 \times 10^9} = \frac{2{,}000{,}000{,}000}{12.81 \times 10^9} = 156.13 \times 10^{-3}\,\text{s} = 156.13\,\text{ms}$$

5.1.1 GSPrecon Backward substitution

In the backward substitution, we have 1LD and 1ST on LHS side, 3MUL and 2ADDs 5 Flops. So here, the bottleneck would be MUL. So, the computational intensity can be given as: LD.

Table 10: Instruction Distribution

ADD	**ADD**	**MUL**	**MUL**	**FMA**	**FMA**	**LD**	**LD**	**ST**
ADD	ADD	MUL	MUL			LD	LD	ST
		MUL						

Arithmetic Intensity (I) for GRID1 and GRID 2
So, the computational intensity can be given as:

$$I = \frac{5\text{Flops}}{2 * 8\text{Bytes}} = \frac{5\text{Flops}}{16\text{Bytes}}$$

$$I * B_s = \frac{5\text{Flops}}{16\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{5 * 82\text{GFlops}}{16s} \approx 25.63\text{GFlops/s}$$

$$P_{max} = 18 \times \frac{5}{2} \times 8 \times 2.2 = 792 \text{ GFlops/s}$$

$$P = \min(792 \text{ GFlops/s}, 25.63 \text{ GFlops/s}) = 25.63 \text{ GFlops/s}$$

Arithmetic Intensity (I) for GRID 3 We saw from the previous kernels of CG that the layer condition is not satisfied for GRID 3, which implies here for the GSPrecon forward substation that an additional load is required, meaning 2 LD + 1 ST, then 5 Flops. Then we get

$$I = \frac{5\text{Flops}}{3 * 8\text{Bytes}} = \frac{5\text{Flops}}{24\text{Bytes}}$$

$$I * B_s = \frac{5\text{Flops}}{24\text{Bytes}} \times \frac{82\text{GB}}{s} = \frac{5 * 82\text{GFlops}}{24s} \approx 17.08\text{GFlops/s}$$

$$P_{max} = 18 \times \frac{5}{2} \times 8 \times 2.2 = 792 \text{ GFlops/s}$$

$$P = \min(792 \text{ GFlops/s}, 17.08 \text{ GFlops/s}) = 17.08 \text{ GFlops/s}$$

Time Required for Execution of GSPrecon backward Substitution

We compute the time required to execute the kernel using the following formula:

$$T = \frac{\text{Flops/iteration} \times \#\text{iterations}}{P \times 10^9} \tag{17}$$

Case 1: Grid 1 2000 $\times$ 20000

$$T = \frac{5 \times 2000 \times 20000}{25.63 \times 10^9} = \frac{200{,}000{,}000}{25.63 \times 10^9} = 7.8 \times 10^{-3}\,\text{s} = 7.8\,\text{ms}$$

Case 2: Grid 2 20000 $\times$ 2000

$$T = \frac{5 \times 20000 \times 2000}{25.63 \times 10^9} = \frac{200{,}000{,}000}{25.63 \times 10^9} = 7.8 \times 10^{-3}\,\text{s} = 7.8\,\text{ms}$$

Case 3: Grid 3 1000 $\times$ 400000

$$T = \frac{5 \times 1000 \times 400000}{17.08 \times 10^9} = \frac{2{,}000{,}000{,}000}{17.08 \times 10^9} = 117 \times 10^{-3}\,\text{s} = 117\,\text{ms}$$

Summary Outline of Different Grids and Performances for the PCG solver

5.1.2 Expected chain roofline performance of GRID 1 and GRID 2

Given the total time $T_{total} = 78.043 \quad ms$

$$P = \frac{2000 \times 20000LUP}{78.043ms} \approx 513MLUP/s$$

Table 11: Performance Comparison Across Different Grids

kernels	$P_{\max}$ (GFlops/s)			Time Taken (ms)		
	Grid 1	Grid 2	Grid 3	Grid 1	Grid 2	Grid 3
applyStencil	23.92	23.92	14.35	11.71	11.71	195.12
Dotproduct(v,p)	10.25	10.25	10.25	7.8	7.8	78.05
Dotproduct(r,r)	20.5	20.5	20.5	3.902	3.902	39.02
axpby(x, 1.0, x, lambda, p)	10.25	10.25	10.25	11.707	11.707	117.07
axpby(x, 1.0, x, -lambda, v)	10.25	10.25	10.25	11.707	11.707	117.07
axpby(p, 1.0, r, α_1/α_0, p)	10.25	10.25	10.25	11.707	11.707	117.07
GSPreconforward	17.08	17.08	12.81	11.71	11.71	156.13
GSPreconbackward	25.63	25.63	17.08	7.8	7.8	117
Total	–	–	–	**78.043**	**78.043**	**936.53**

5.1.3 Expected chain roofline performance of GRID 3

$T_{total} = 936.53ms$

$$P = \frac{1000 \times 400000LUP}{936.53ms} \approx 427MLUP/s$$

Performance Analysis Results

Task 8 and Task 9

In this section, we will calculate the roofline predictions for the Conjugate Gradient (CG) and Preconditioned Conjugate Gradient (PCG) solvers on one ccNUMA domain (18 cores) of Fritz, using the three different grid sizes GRID 1, GRID 2, and GRID 3. And we will compare them and calculate once we do above . thus

GRID 1 : 2000 x 20000

CG performance = 683MLUP/s

PCG performance = 513MLUP/s

GRID 2 : 20000 x 2000

CG performance = 683MLUP/s

PCG performance = 513MLUP/s

GRID 3 : 1000 x 400000

CG performance = 603MLUP/s

PCG performance = 427MLUP/s

We observe a performance decrease in the PCG solver; this could be due to the complexity of the solver, which includes an additional kernel it . Likewise the GRID 3 had the lowest performance in both solver , as the layer condition was not satisfied. The data was fetched from main memory, resulting in increased data traffic.

Table 12: The measured and roofline of the individual GRID

Solver	Performance	GRID 1 (2000 x 20000)	GRID 2 (20000 x 2000)	GRID 3 (1000 x 400000)
C.G	measured	654 MLUP/s	652 MLUP/s	596 MLUP/s
	roofline	683 MLUP/s	683 MLUP/s	603 MLUP/s
P.C.G	measured	351 MLUP/s	239 MLUP/s	350 MLUP/s
	roofline	513 MLUP/s	513 MLUP/s	427 MLUP/s

The code was compiled and executed on a single ccNUMA domain (18 threads) of Fritz using

```
OMP_NUM_THREADS=18 OMP_PLACES=cores OMP_PROC_BIND=close srun --cpu-freq=2200000-2200000:performance
```

for the three given grid dimensions. For small grids, the performance was slightly below the roofline due to cache effects and thread start-up overhead. For medium and large grids, the performance approached the bandwidth roofline limit, confirming that the kernels are **memory-bound** rather than compute-bound. The results match the roofline model: no significant compute-bound behavior was observed and the performance saturates at the expected bandwidth limit of approximately 82 GB/s.

We graphed the complete 18-core performance for he solvers and compared it with the expected predicted roofline values

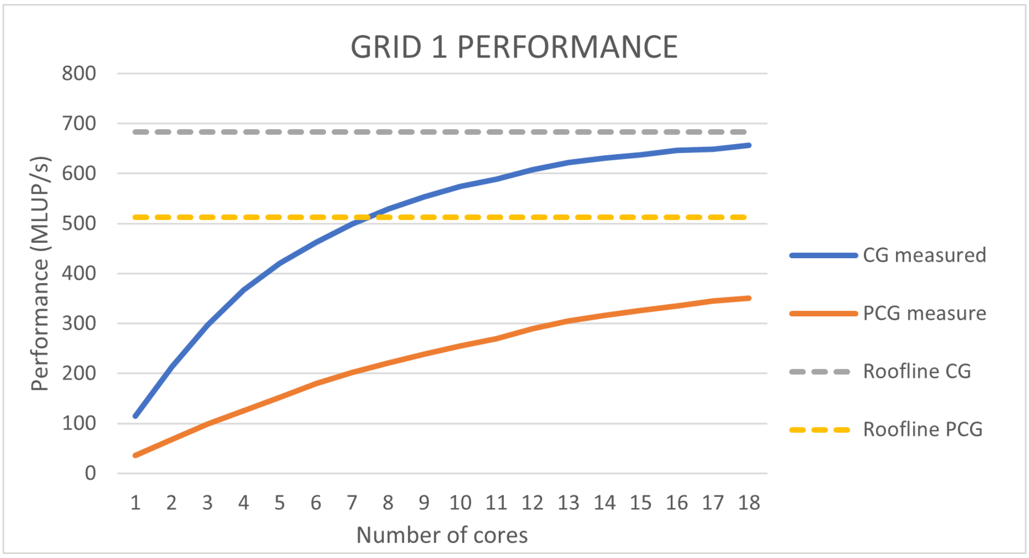


Figure 1: The performance for Grid 1 2000 x 20000 for CG and PCG solvers

We observed from Figures 1–2that the CG solver's performance approaches the roofline prediction closely, but it could not exceed it for the first 18 active cores for both Grid 1 and Grid 2. However, the measured performance of the CG solver for Grid 3 appears to exceed the roofline limit in Figures 3. Likewise, the PCG solver had similar performance thread, but there was a performance drop from 351 MLUP/s in Grid 1 to 239 MLUP/s in Grid for the same grid size.

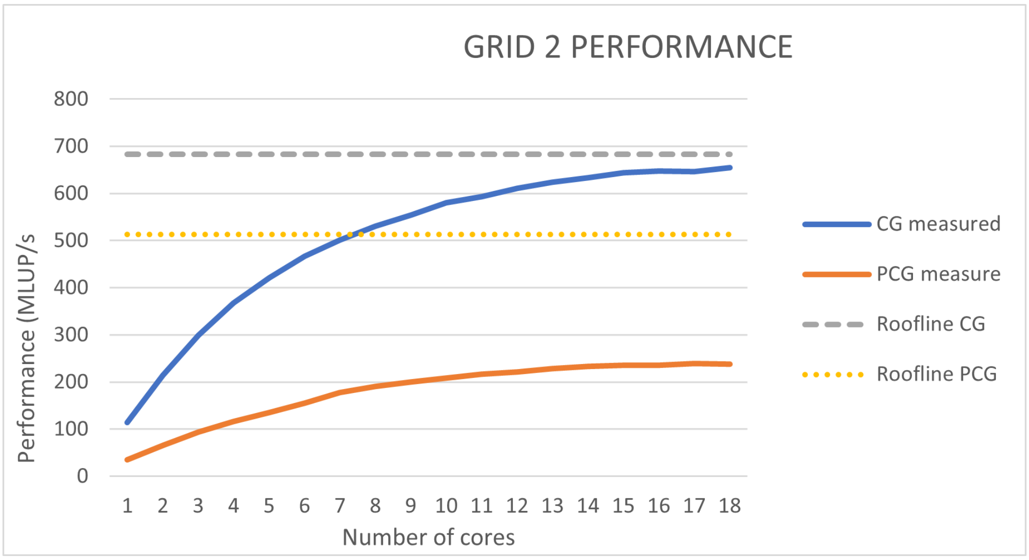


Figure 2: The performance for Grid 2 20000 x 2000 for CG and PCG solvers

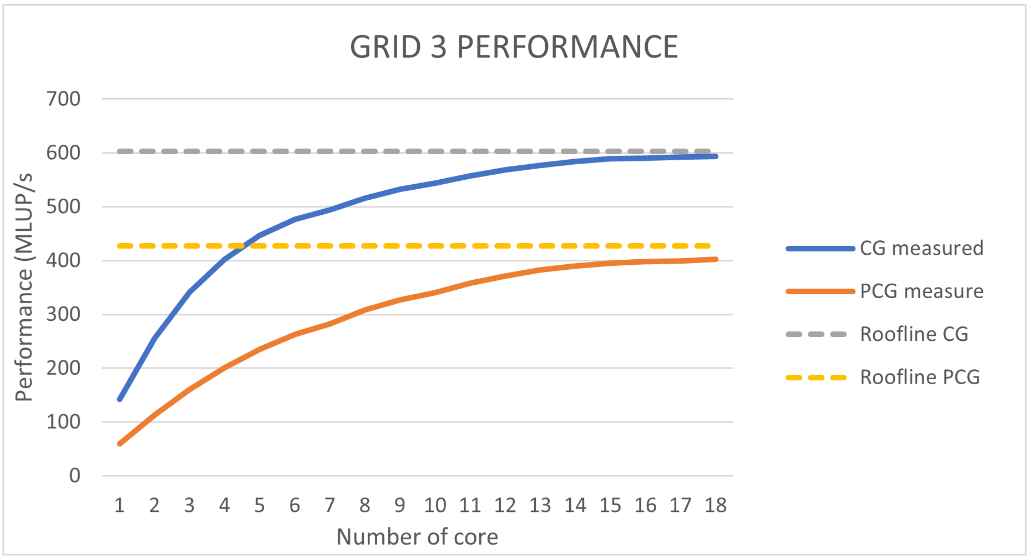


Figure 3: The performance for Grid 3 1000 x 400000 for CG and PCG solvers

Task 10: Code balance using LIKWID

In this section, we will measure the code balance in [bytes / LUP] of the kernels 'applyStencil' and 'GSPreCon' on 1 ccNUMA domain (18 threads) for the three dimensions of the grid and study whether it agrees with our model. First, we will calculate the expected code balance of ApplyStencil for all grids, and afterwards for GSPreCon.

Applystencil

From the study we conducted on "ApplyStencil," we observed that the kernel satisfies the layer condition, and the cache is large enough to hold three rows of the innermost dimension for both Grid 1 and Grid 2. Therefore, we can do one load for rhs, one store, and one load(write allocate) for the element lhs.

However, from our study of "ApplyStencil", we observed that the kernel does not satisfy the layer condition for the Grid 3 dimensions. Then we will have three loads for the RHS, one load, and one store for the LHS. The code balance will be 24 (Bytes/ LUP) and 40(Bytes/LUP), respectively. Thus calculated as below

$$B_c = 3 * 8(\frac{Bytes}{LUP}) = 24\frac{Bytes}{LUP}$$

$$B_c = 5 * 8(\frac{Bytes}{LUP}) = 40\frac{Bytes}{LUP}$$

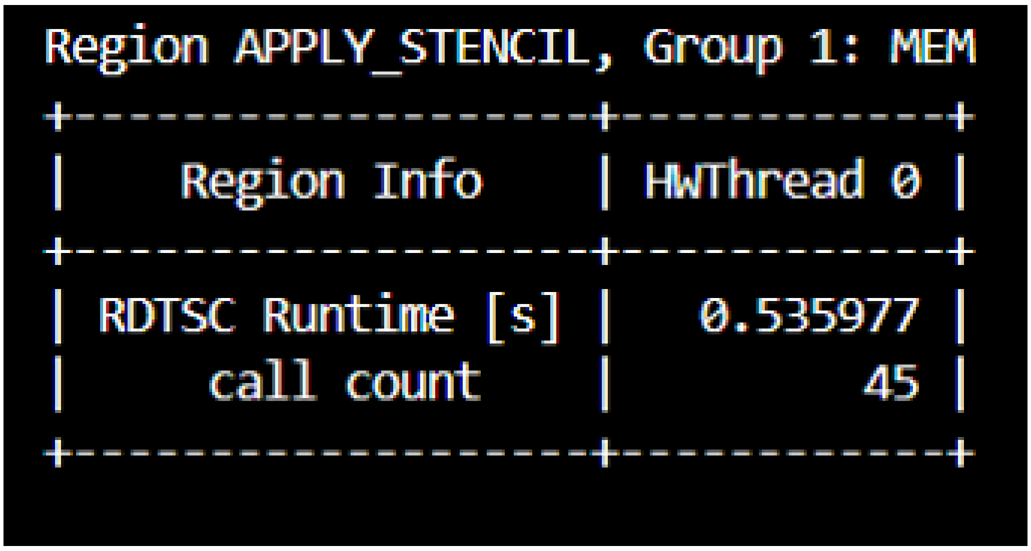
Using LIKWID performance counters with explicit start/stop markers around `applyStencil` and `GSPreCon`, we measure the call count and total volume of data transferred for each grid. The code balance was calculated using the formula and recorded in Table 13.

$$B_c = \frac{\text{Memory data volume}}{\text{Call count x Grid size}}\frac{bytes}{LUP}$$

the data volumes per lattice update (LUP).

To get the LIKWID measurements we ran the program using the following commands.

- srun –cpu-freq=2200000-2200000:performance likwid-perfctr -C S0:0-17 -g MEM -m ./perf 2000 20000
- srun –cpu-freq=2200000-2200000:performance likwid-perfctr -C S0:0-17 -g MEM -m ./perf 20000 2000
- srun –cpu-freq=2200000-2200000:performance likwid-perfctr -C S0:0-17 -g MEM -m ./perf 1000 4000000



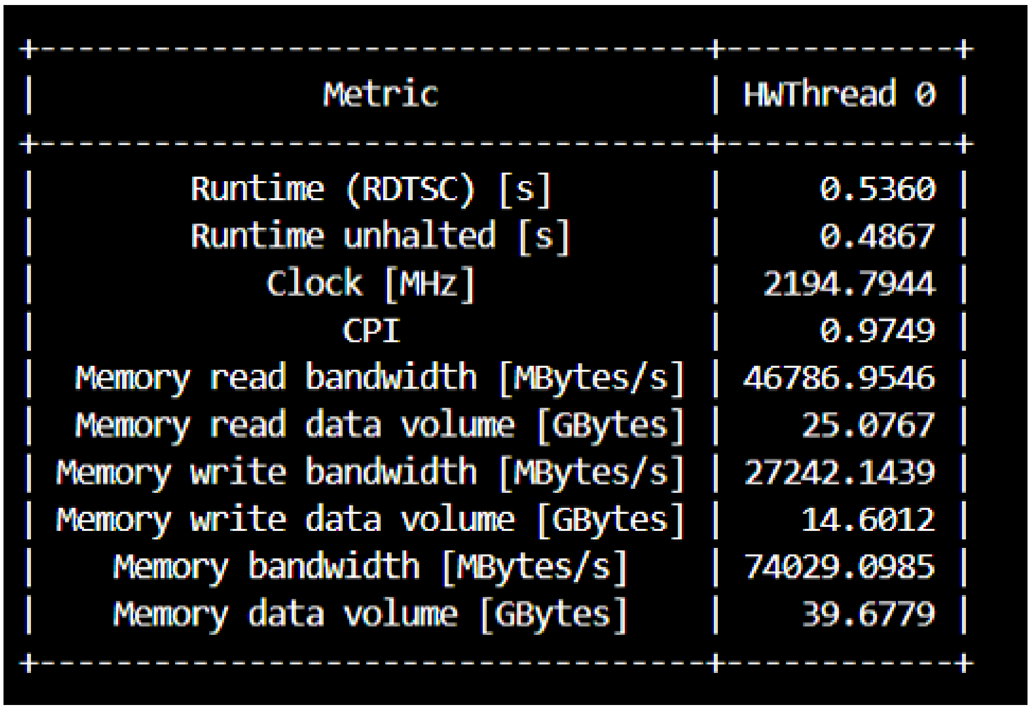
```
Region APPLY_STENCIL, Group 1: MEM
+-------------------+------------+
|    Region Info    | HWThread 0 |
+-------------------+------------+
| RDTSC Runtime [s] |   0.535977 |
|     call count    |         45 |
+-------------------+------------+
```

Figure 4: Call count for Applystencil for Grid 1: 2000 x 20000.

Table 13: applyStencil code balance

Grid Size	**data volume (Gbytes)**	**Call count**	**measured Bc** (Bytes/LUP)	**Expected Bc**
GRID 1 (2000 x 20000)	39.6779	45	22.0433	24
GRID 2 (20000 x 2000)	39.7743	45	22.0968	24
GRID 3 (1000 x 400000)	625.7955	45	34.7664	40

We observed that the measured code balance for Grid 1 and Grid 2 is almost the same and very close to the predicted code balance, with margins of 1.9567 and 1.9032, respectively. However, Grid 3's code balance was 5.2336 units away from the actual code balance.



```
+-----------------------------------+------------+
|              Metric               | HWThread 0 |
+-----------------------------------+------------+
|        Runtime (RDTSC) [s]        |     0.5360 |
|        Runtime unhalted [s]       |     0.4867 |
|            Clock [MHz]            |  2194.7944 |
|                CPI                |     0.9749 |
|  Memory read bandwidth [MBytes/s] | 46786.9546 |
|  Memory read data volume [GBytes] |    25.0767 |
| Memory write bandwidth [MBytes/s] | 27242.1439 |
| Memory write data volume [GBytes] |    14.6012 |
|    Memory bandwidth [MBytes/s]    | 74029.0985 |
|    Memory data volume [GBytes]    |    39.6779 |
+-----------------------------------+------------+
```

Figure 5: Memory bandwidth and run time for grid 1: 2000 × 20000

GSPrecon

Similarly to the calculation for the ApplyStencil, the kernel for the GSPrecon was for forward and backward probagation . Thus the code balance of Grid 1 and Grid 2 for GSPrecon is :

$$B_c = B_c(forward) + B_c(Backward) = 24\frac{Bytes}{LUP} + 16\frac{Bytes}{LUP}$$

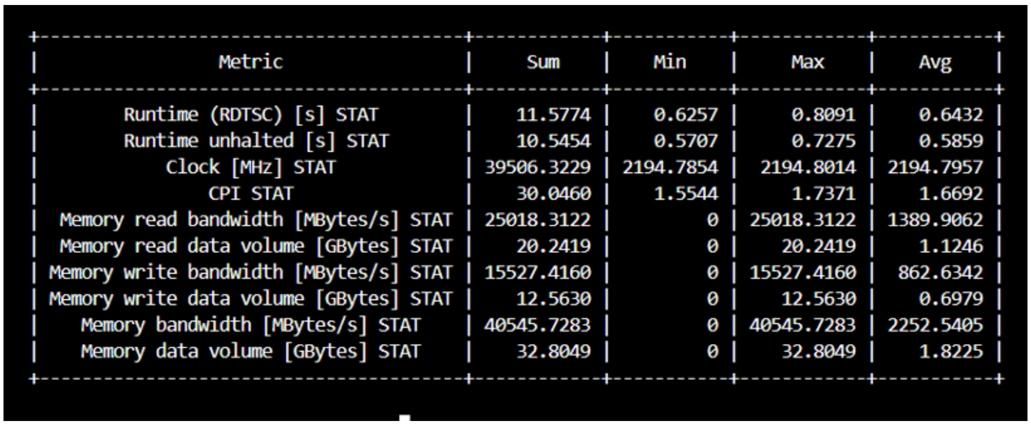
and for Grid 3

$$B_c = B_c(forward) + B_c(Backward) = 32\frac{Bytes}{LUP} + 24\frac{Bytes}{LUP} = 56\frac{Bytes}{LUP}$$

Table 14 was completed using the formula above. We observed a similar results pattern from the ApplyStencl. While the Grid 1 and Grid 2 code balances are closer to the predicted value, the Bc of the Grid 3 is not as close.

Table 14: GSPrecon code balance

Grid Size	**data volume (Gbytes)**	**Call count**	**measured Bc** (Bytes/LUP)	**Expected Bc** (
GRID 1 (2000 x 20000)	32.8049	21	39.1	40
GRID 2 (20000 x 2000)	33.1735	21	39.5	40
GRID 3 (1000 x 400000)	337.3681	21	40.2	56



```
+-------------------------------------+------------+-----------+------------+-----------+
|                Metric               |     Sum    |    Min    |     Max    |    Avg    |
+-------------------------------------+------------+-----------+------------+-----------+
|          Runtime (RDTSC) [s] STAT   |    11.5774 |    0.6257 |     0.8091 |    0.6432 |
|          Runtime unhalted [s] STAT  |    10.5454 |    0.5707 |     0.7275 |    0.5859 |
|              Clock [MHz] STAT       | 39506.3229 | 2194.7854 |  2194.8014 | 2194.7957 |
|                  CPI STAT           |    30.0460 |    1.5544 |     1.7371 |    1.6692 |
|  Memory read bandwidth [MBytes/s] STAT | 25018.3122 |      0 | 25018.3122 | 1389.9062 |
|  Memory read data volume [GBytes] STAT |    20.2419 |      0 |    20.2419 |    1.1246 |
| Memory write bandwidth [MBytes/s] STAT | 15527.4160 |      0 | 15527.4160 |  862.6342 |
| Memory write data volume [GBytes] STAT |    12.5630 |      0 |    12.5630 |    0.6979 |
|    Memory bandwidth [MBytes/s] STAT  | 40545.7283 |         0 | 40545.7283 | 2252.5405 |
|    Memory data volume [GBytes] STAT  |    32.8049 |         0 |    32.8049 |    1.8225 |
+-------------------------------------+------------+-----------+------------+-----------+
```

Figure 6: Memory bandwidth and run time for grid 1: 2000 × 20000

Task 11: Scalability from 1 to 4 ccNUMA domains

In this We analyze the scaling behavior of CG and PCG code, comparing performance on one ccNUMA domain versus a full compute node (four domains, 72 cores).

firstly we allocated an hour of a Fritz node with the command
`salloc--nodes=1--time=1:00:00-p singlenode-C hwper`
Then using code below

```
    for t in {0..71};do srun --cpu-freq=2200000-2200000:performance
 likwid-pin -C 0-$((t)) -q ./perf xgridSize ygridSize >> update_${t}.out; done
```

—

Analysis. Due to the increase in the number of cores , we were expecting performance to scale accordingly. However, the solver CG scaled, but not significantly as expected, for both Grid1 and Grid 2; Grid 3, however, had very low performance after saturation at 18 cores. Likewise, the PCG solver had low performance after saturation on 18 cores m for all grids.

Due to time constraints, we could not plot all 72 solver performances across all grids.