PARALLEL PROGRAMMING FINAL PROJECT

Group 1 李秉綸 翁君牧 謝東豫

OUTLINE

- Problem description
- CPU – sequential
- GPU – different library implementation
- Performance comparison
- Conclusion

PROBLEM DESCRIPTION

GOAL

Implement multiple version of Jacobi solver, compare performance of different implementation and conduct conclusion.

Implementation

- Sequential
- OpenMP
- OpenCL
- SyCL
- CUDA

JACOBI SOLVER

Jacobi method iteratively solves the system Ax=b

$$Ax = (L + D + U)x = b \quad \Longrightarrow \quad x^{(k+1)} = (b - (L + U)x^{(k)})/D$$

- Start from random value
- Continues until the solution converges within a given tolerance.
- Computationally expensive for large N, suitable for parallelization

IMPLEMENTATION

```
while ((conv > TOLERANCE) && (iters < MAX_ITERS)) {
    iters++;

    for (int i = 0; i < Ndim; i++) {
      xnew[i] = (TYPE)0.0;
      for (int j = 0; j < Ndim; j++) {
        if (i != j)
          xnew[i] += A[i * Ndim + j] * xold[j];
      }
      xnew[i] = (b[i] - xnew[i]) / A[i * Ndim + i];
    }
    //
    // test convergenc
    //
    conv = 0.0;
    for (int i = 0; i < Ndim; i++) {
      TYPE tmp = xnew[i] - xold[i];
      conv += tmp * tmp;
    }
    conv = sqrt((double)conv);
#ifdef DEBUG
    printf(" conv = %f \n", (float)conv);
#endif

    TYPE* tmp = xold;
    xold = xnew;
    xnew = tmp;
  }
```

CPU - SEQUENTIAL

- Calculate x(k+1) from x(k)
- Test convergence

OPENMP - GPU

- Copy data in GPU
- Generate team and distribute workload
- Generate threads and distribute workload
- Same on test convergence

```
#pragma omp target enter data map(to: xold[0:Ndim], xnew[0:Ndim], \
    A[0:Ndim*Ndim], b[0:Ndim])

  while ((conv > TOLERANCE) && (iters < MAX_ITERS)) {
    iters++;

#pragma omp target
#pragma omp teams distribute parallel for simd
    for (int i = 0; i < Ndim; i++) {
      xnew[i] = (TYPE)0.0;
      for (int j = 0; j < Ndim; j++) {
        xnew[i] += A[i * Ndim + j] * xold[j] * (i != j);
      }
      xnew[i] = (b[i] - xnew[i]) / A[i * Ndim + i];
    }
    //
    // test convergenc
    //
    conv = 0.0;

#pragma omp target map(tofrom: conv)
#pragma omp teams distribute parallel for simd reduction(+: conv)
    for (int i = 0; i < Ndim; i++) {
      TYPE tmp = xnew[i] - xold[i];
      conv += tmp * tmp;
    }
    conv = sqrt((double)conv);

#ifdef DEBUG
    printf(" conv = %f \n", (float)conv);
#endif

    TYPE* tmp = xold;
    xold = xnew;
    xnew = tmp;
  }

#pragma omp target exit data map(from : xold[0 : Ndim], xnew[0 : Ndim])
```

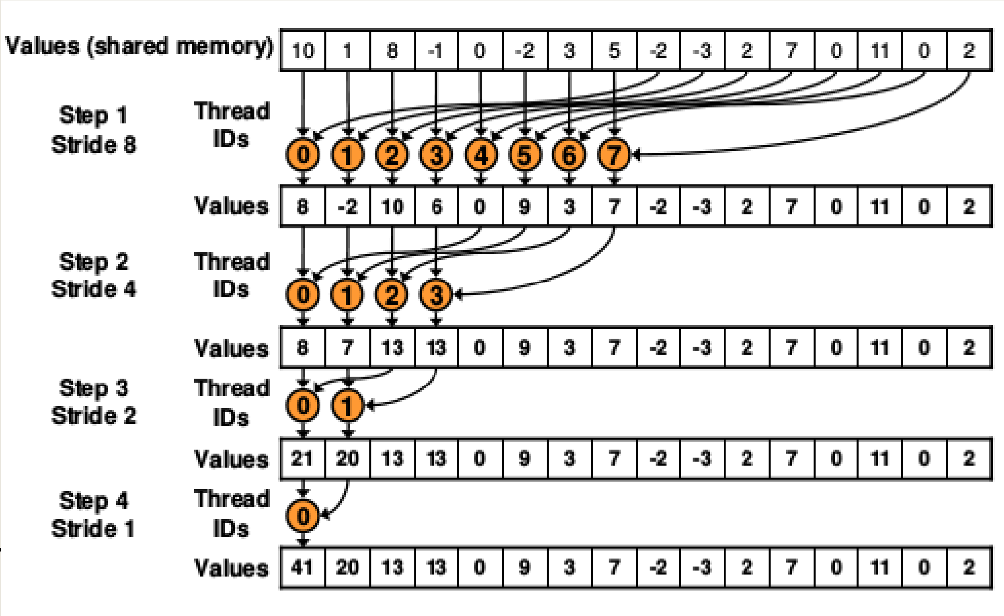
OPENCL - GPU

- Create context, command queue corresponding to device
- Issue buffer read/write and kernel launch to command queue
- Build kernel in runtime
- Set appropriate workgroup (block) size

```
kernel void jacobi(const unsigned Ndim, global TYPE *A, global TYPE *
b, global TYPE *xold, global TYPE *xnew) {
  size_t i = get_global_id(0);

  xnew[i] = 0.0;
  for (int j = 0; j < Ndim; j++)
    xnew[i] += A[i * Ndim + j] * xold[j] * (i != j);
  xnew[i] = (b[i] - xnew[i]) / A[i * Ndim + i];
}

kernel void convergence(global TYPE *xold, global TYPE *xnew, global TYPE *
conv, local TYPE *conv_loc) {
  size_t i = get_global_id(0);
  TYPE tmp = xnew[i] - xold[i];
  conv_loc[get_local_id(0)] = tmp * tmp;

  barrier(CLK_LOCAL_MEM_FENC
         E);
  for (int offset = get_local_size(0) >> 1; offset; offset >>= 1) {
    if (get_local_id(0) < offset)
      conv_loc[get_local_id(0)] += conv_loc[get_local_id(0) + offset];
    barrier(CLK_LOCAL_MEM_FENC
  }        E);
  if (get_local_id(0) == 0)
    conv[get_group_id(0)] = conv_loc[0];
}
```

SYCL - GPU

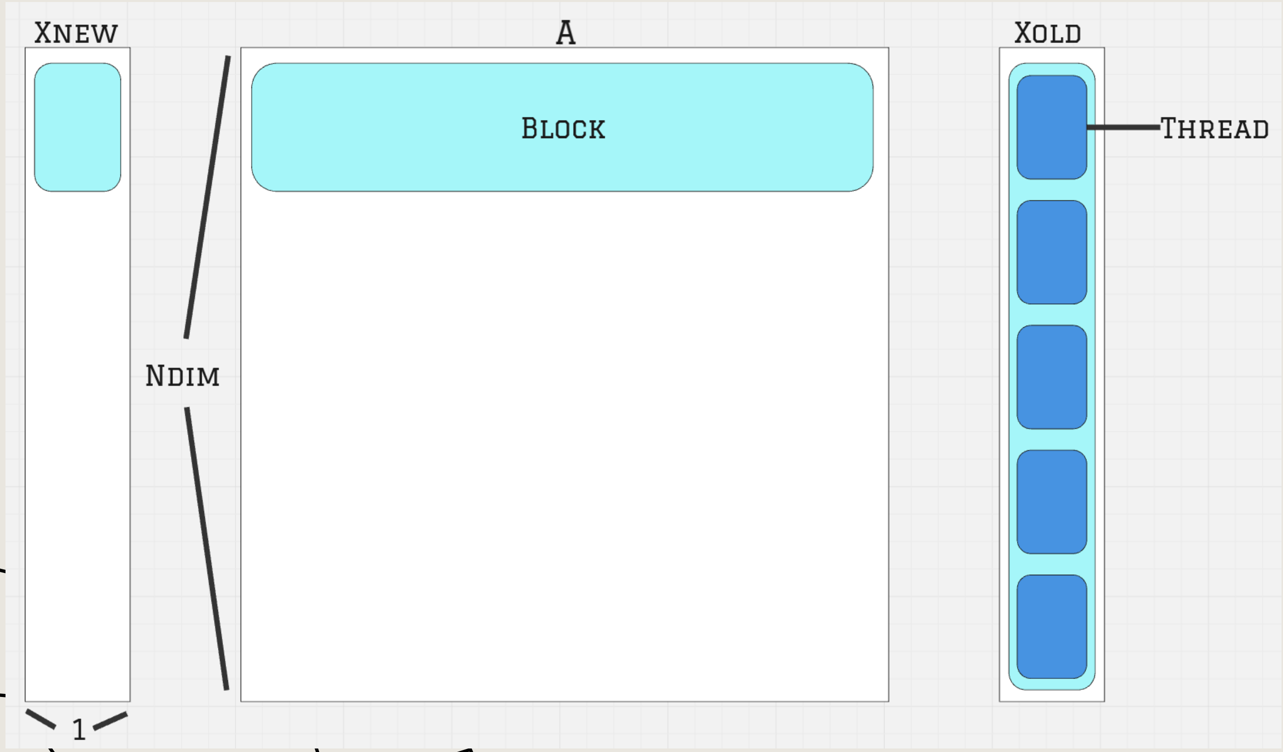
- Allocate a thread for a single scalar multiplication
- Use a work group to perform reduction
- Run convergence calculation on device




```
    while ((conv > TOLERANCE) && (iters < MAX_ITERS))
    {
        iters++;

        if ((iters & 1) == 0)
        {
            q.submit([&](sycl::handler &h)
            {
                // sycl::stream out(1024, 256, h); //output buffe
r
                h.parallel_for(sycl::nd_range<1>(blocks * threads, threads),  =](sycl::nd_item
<1> item) [[sycl::reqd_sub_group_size(3                [
        2)]]   {
                    jac_solver_mai
(item, Ndim, Ndim_round, d_A, d_b, d_xnew, d_xold, d_A_reduce, d_conv_reduce, blocks_per_grid
* threads_per_bloc
 k);
                    // out << d_A_reduce[0] <<
"\n";           });
            });

            q.submit([&](sycl::handler &h)
            {
                // sycl::stream out(1024, 256, h); //output buffe
r               // sycl::local_accessor<TYPE> shm_acc(sycl::range<1>(BLOCKS_PER_GRID),
h);
                h.parallel_for(sycl::nd_range<1>(reduce_blocks_per_grid / 2
, reduce_blocks_per_grid / 2), [=](sycl::nd_item<1> item) [[sycl::reqd_sub_group_size(3
                {                                                      2)]]
                    reduce(item, d_conv, d_conv_reduce, reduce_blocks_per_gri
                });        d);
            });
        }
```

CUDA - GPU

- Share memory
- Reduce
- Grid layout <Ndim / block size, block size>




```
__global__ void jacobi(const unsigned Ndim, TYPE *const
__restrict__ A, TYPE *const __restrict__ b, TYPE *const
__restrict__ xold, TYPE *const __restrict__ xnew) {
  const size_t i = blockIdx.x * blockDim.x + threadIdx.x;
  extern __shared__ TYPE xold_loc[];
  int chunk = Ndim / blockDim.x;
  for (int j = 0; j < chunk; j++)
    xold_loc[threadIdx.x * chunk + j] = xold[threadIdx.x * chunk
 + j];

  __syncthreads();

  xnew[i] = 0.0;
  for (int j = 0; j < Ndim; j++)
    xnew[i] += A[i * Ndim + j] * xold_loc[j] * (i != j);
  xnew[i] = (b[i] - xnew[i]) / A[i * Ndim + i];
}

__global__ void convergence(TYPE *const __restrict__ xold, TYPE
*const __restrict__ xnew, TYPE *const __restrict__ conv) {
  extern __shared__ TYPE conv_loc[];
  const size_t i = blockIdx.x * blockDim.x + threadIdx.x;
  TYPE tmp = xnew[i] - xold[i];
  conv_loc[threadIdx.x] = tmp * tmp;

  __syncthreads();

  for (int offset = blockDim.x >> 1; offset; offset >>= 1) {
    if (threadIdx.x < offset)
      conv_loc[threadIdx.x] += conv_loc[threadIdx.x + offset];
    __syncthreads();
  }
  if (threadIdx.x == 0)
    conv[blockIdx.x] = conv_loc[0];
}
```

ENVIRONMENTS

TESTBEDS

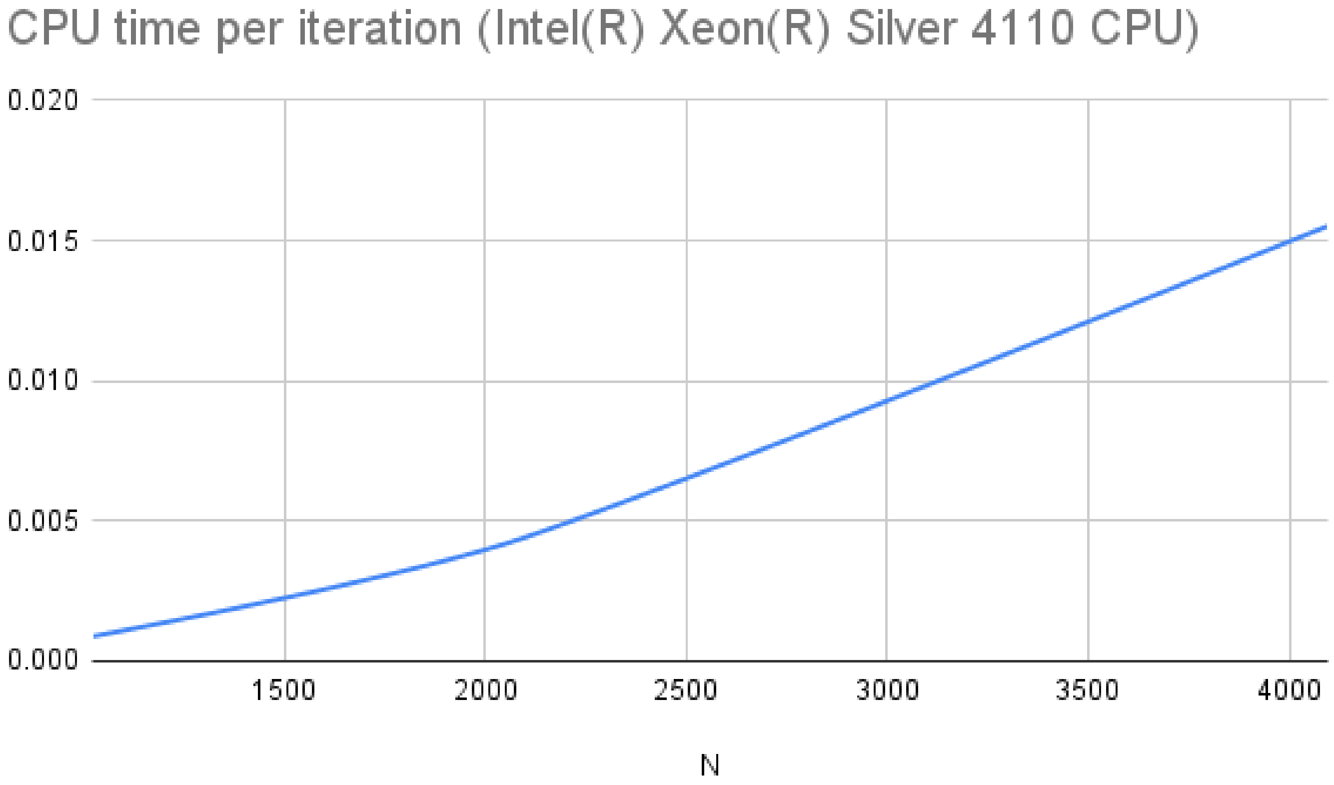
- Apollo GPU
  - NVIDIA Geforce GTX 1080
  - AMD Instinct MI210
- Titan
  - NVIDIA Tesla P100 PCIe 16GiB
  - NVIDIA Tesla V100 PCIe 16GiB

COMPILERS

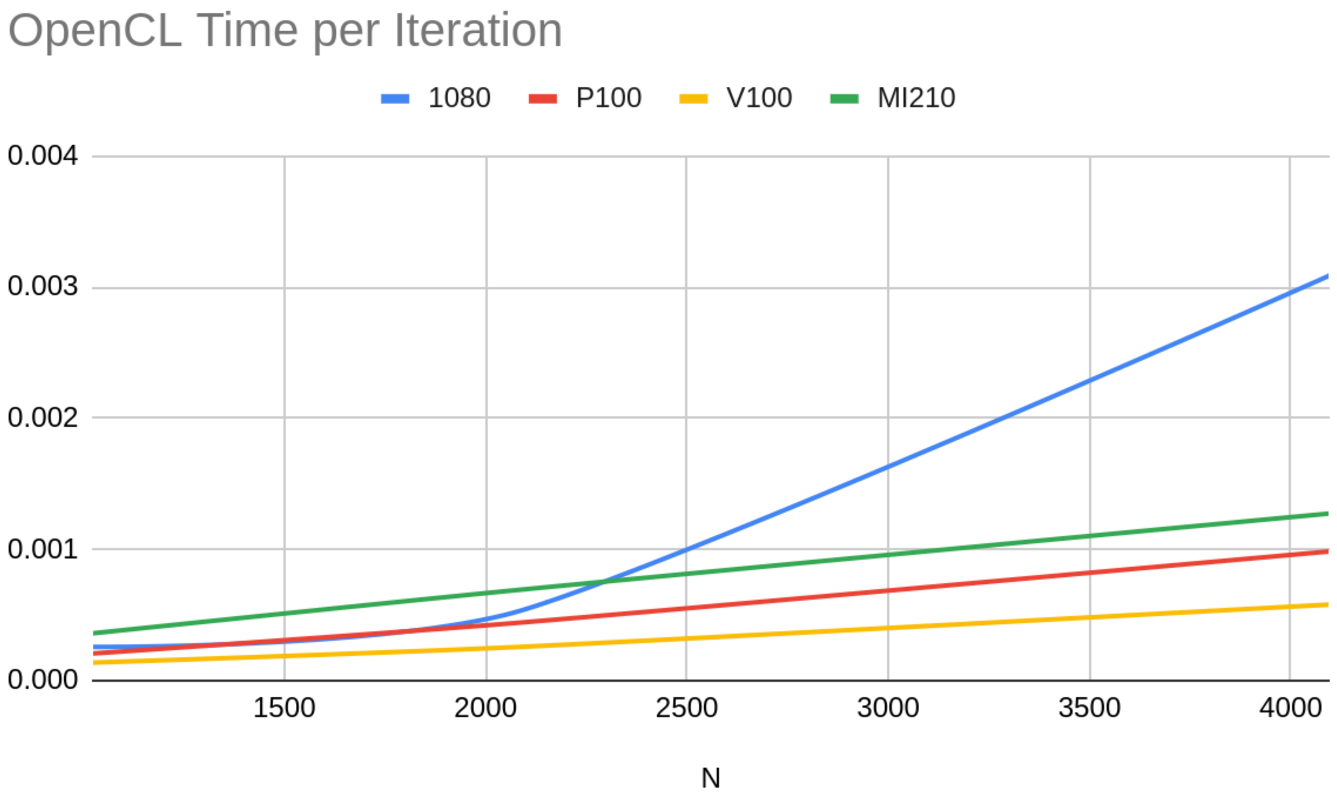
- OpenMP GPU
  - amdclang
  - nvc
- OpenCL
  - g++ (dynamically linked to NVIDIA's or AMD's libOpenCL.so)
- SyCL
  - Icpx + CUDA / ROCm
- CUDA
  - nvcc
  - hipify-clang + hipcc for MI210

PERFORMANCE

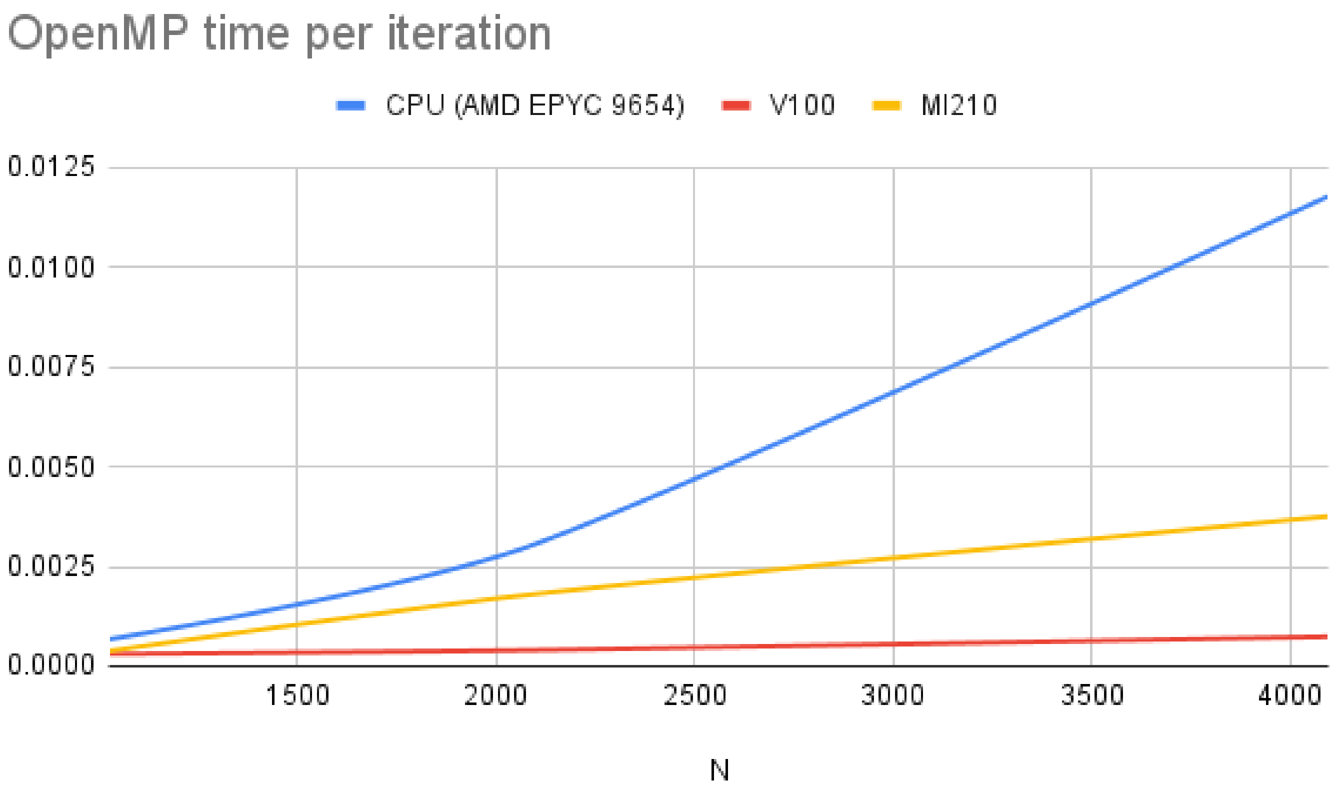
CPU - PERFORMANCE



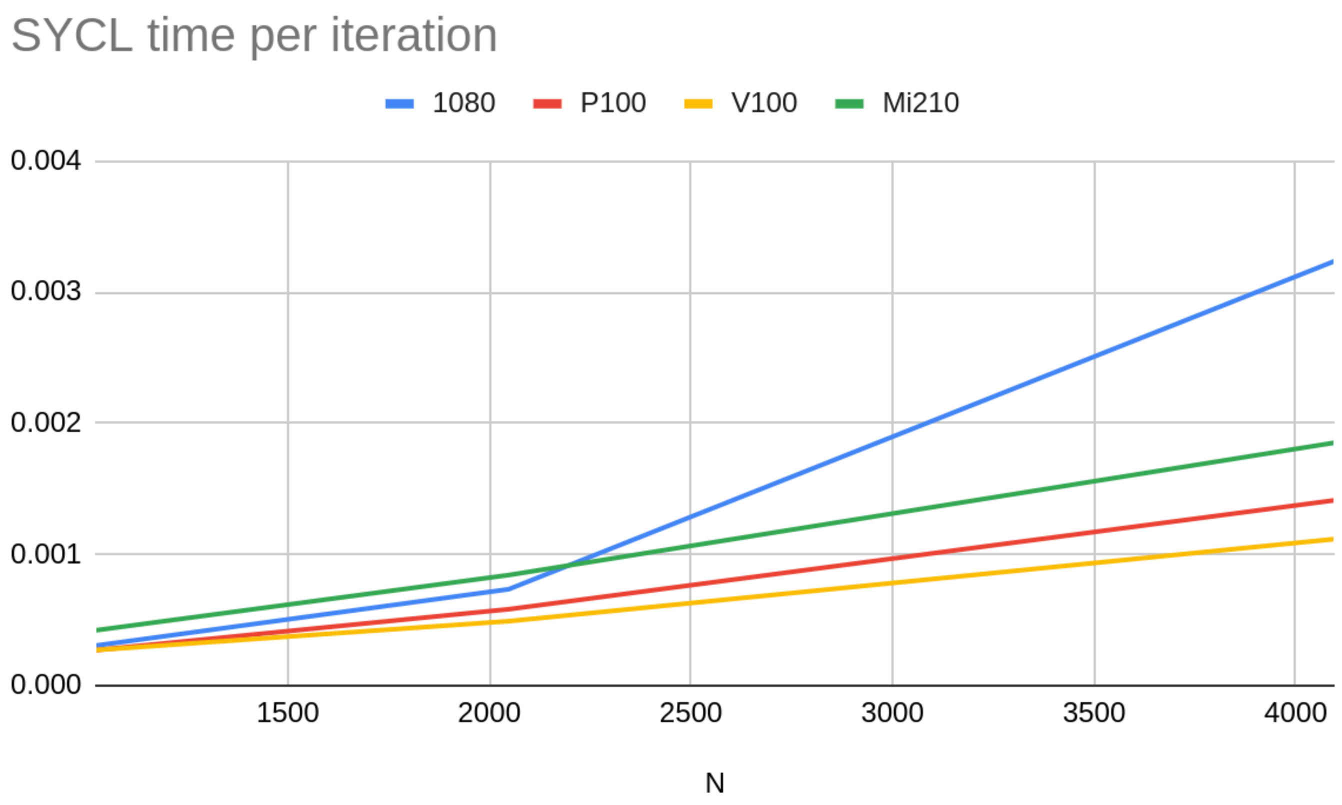
OPENCL - PERFORMANCE



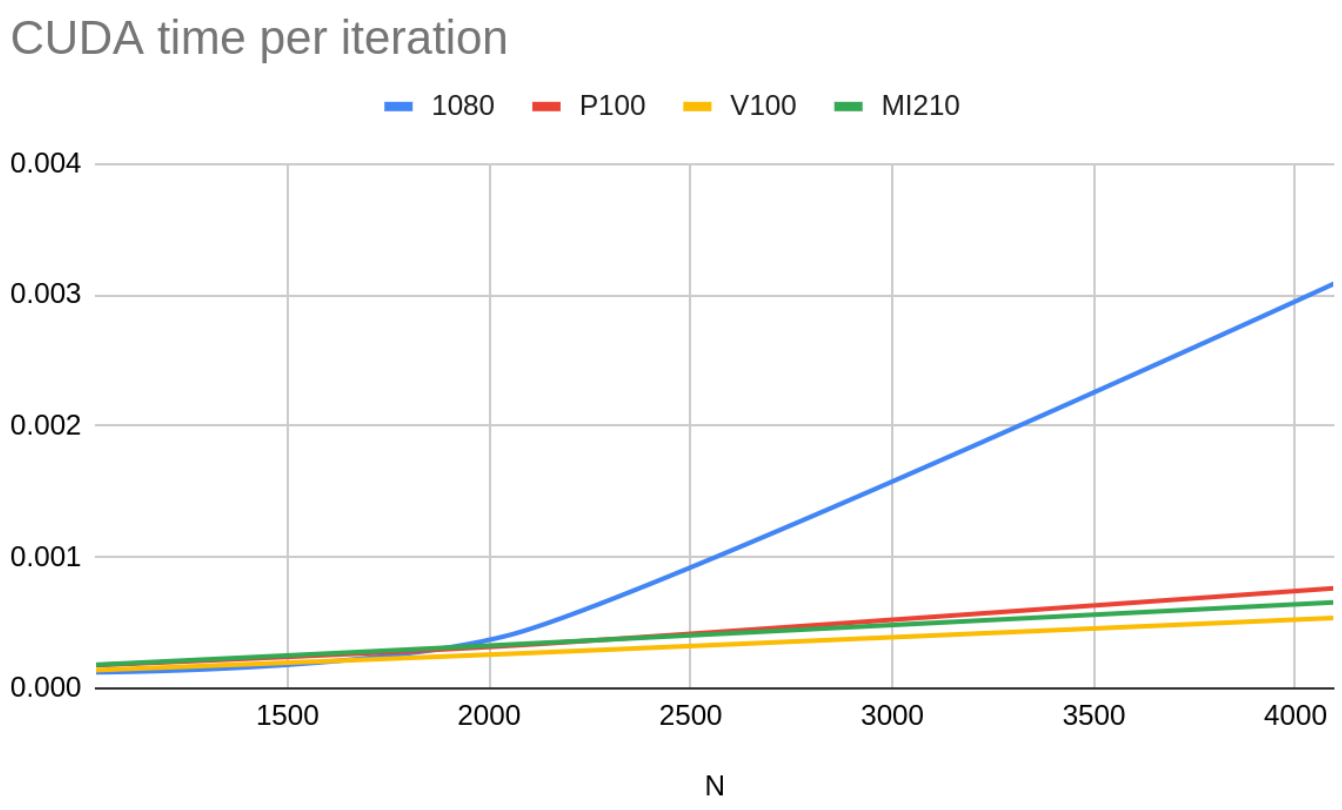
OPENMP - PERFORMANCE



SYCL - PERFORMANCE



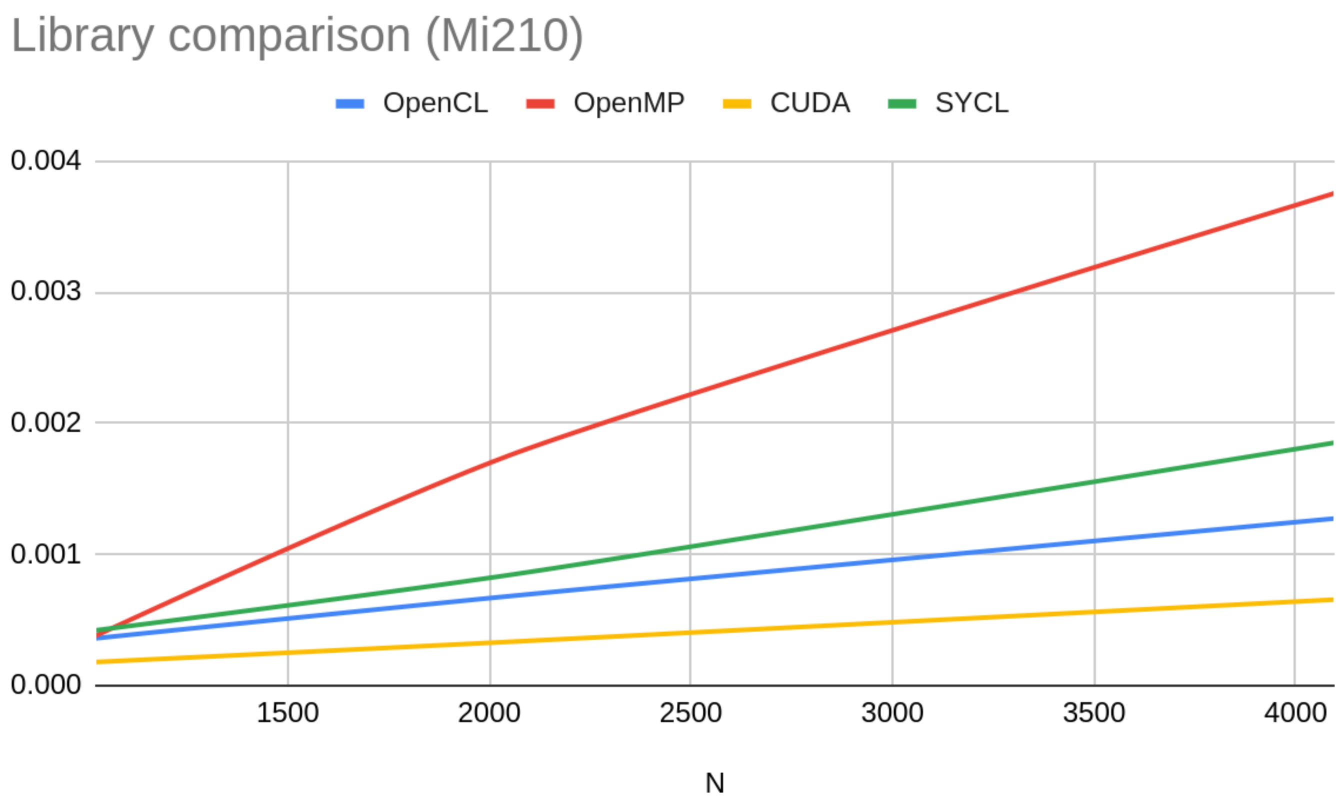
CUDA - PERFORMANCE



PERFORMANCE COMPARISON

LIBRARY – PERFORMANCE (V100)

LIBRARY – PERFORMANCE (MI210)



CONCLUSION

CONCLUSION

OpenCL

- Cross platform (AMD, NVIDIA, Smartphone GPU, FPGA, ...)
- Runtime kernel building
- Fast for performance

OpenMP

- Easy programming
- Cross platform
- Fairly good performance
- Difficult for optimization

SyCL

- Cross platform
- Higher dispatch overhead

CUDA

- More programming effort
- Ultimate performance
- All in control
- Vendor specific

REFERENCE

REFERENCES

- Programming Your GPU with OpenMP: A "Hands-On" Introduction (Tutorial at SC’18 ~ ‘24)
- Portable Programs for Heterogeneous Computing: A Hands-On Introduction (Tutorial at SC’15)
- “Hands On OpenCL” Project by UoB HPC

THANK YOU