A Case Study of LLVM-Based Analysis for Optimizing SIMD Code Generation

Joseph Huber

Computer Science and Mathematics

15 September 2021

SciDAC: Computational Framework for Unbiased Studies of Correlated Electron Systems (CompFUSE)

The parallel abstraction optimization was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. DOE, Office of Science, Advanced Scientific Computing Research (ASCR) and Basic Energy Sciences (BES), Division of Materials Science and Engineering.

This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Support for UO through the U.S. Department of Energy, Office of Science, ASCR, RAPIDS SciDAC Institute for Computer Science and Data under subcontract 4000159855 from ORNL.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-819815).

This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE).

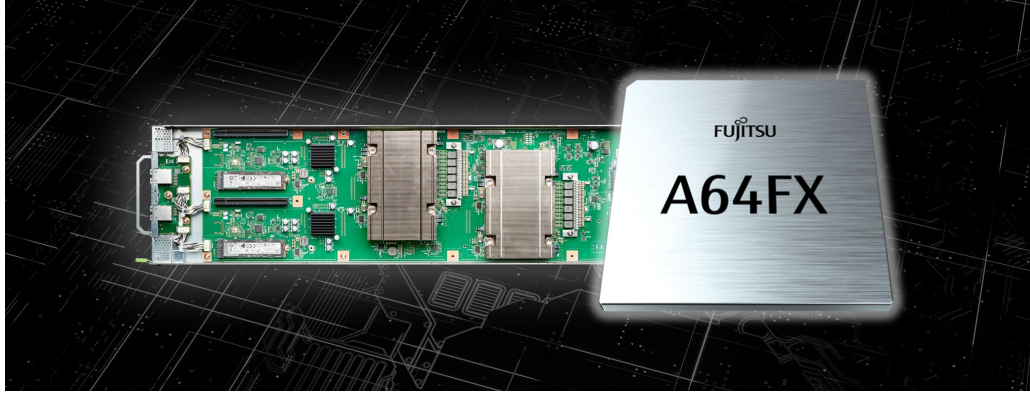
DCA++ (Dynamical Cluster Approximation)

- Scientific software for solving quantum many-body electronic correlation problems
- A numerical simulation tool to predict behaviors of co-related quantum materials (such as **superconductivity**, **magnetism**)
- Ported to world's largest supercomputers, e.g. Titan, Summit, Cori, Piz Daint (CSCS) sustaining many petaflops of performance
- **Gordon Bell Prize Winner 2008**, a highly scalable application
- Open-source software written in morden C++ (800K+ lines of code)

[1] DCA++ 2019. Dynamical Cluster Approximation. https://github.com/CompFUSE/DCA [Licensing provisions: BSD-3-Clause]
[2] Urs R. Hähner, Gonzalo Alvarez, Thomas A. Maier, Raffaele Solcà, Peter Staar, Michael S. Summers, and Thomas C. Schulthess, DCA++: A software framework to solve correlated electron problems with modern quantum cluster methods, Comput. Phys. Commun. 246 (2020) 106709.
[3] DCA++ ran on Titan – 18600 nodes at16 Petaflop rate (peak), sustained 1.3 Petaflop rate [Gordon Bell 2008]

Slide reused & modified from Wei. P3HPC 20'.

Background



- Port DCA++ to run efficiently on an A64FX cluster
  - Run on the Wombat cluster at ORNL OLCF
  - https://www.olcf.ornl.gov/olcf-resources/compute-systems/wombat/
- Target SVE vectorization to improve performance
- Evaluate impact of vectorization on the A64FX architecture
- Evaluation was done using Arm compiler 20.3

Baseline Performance

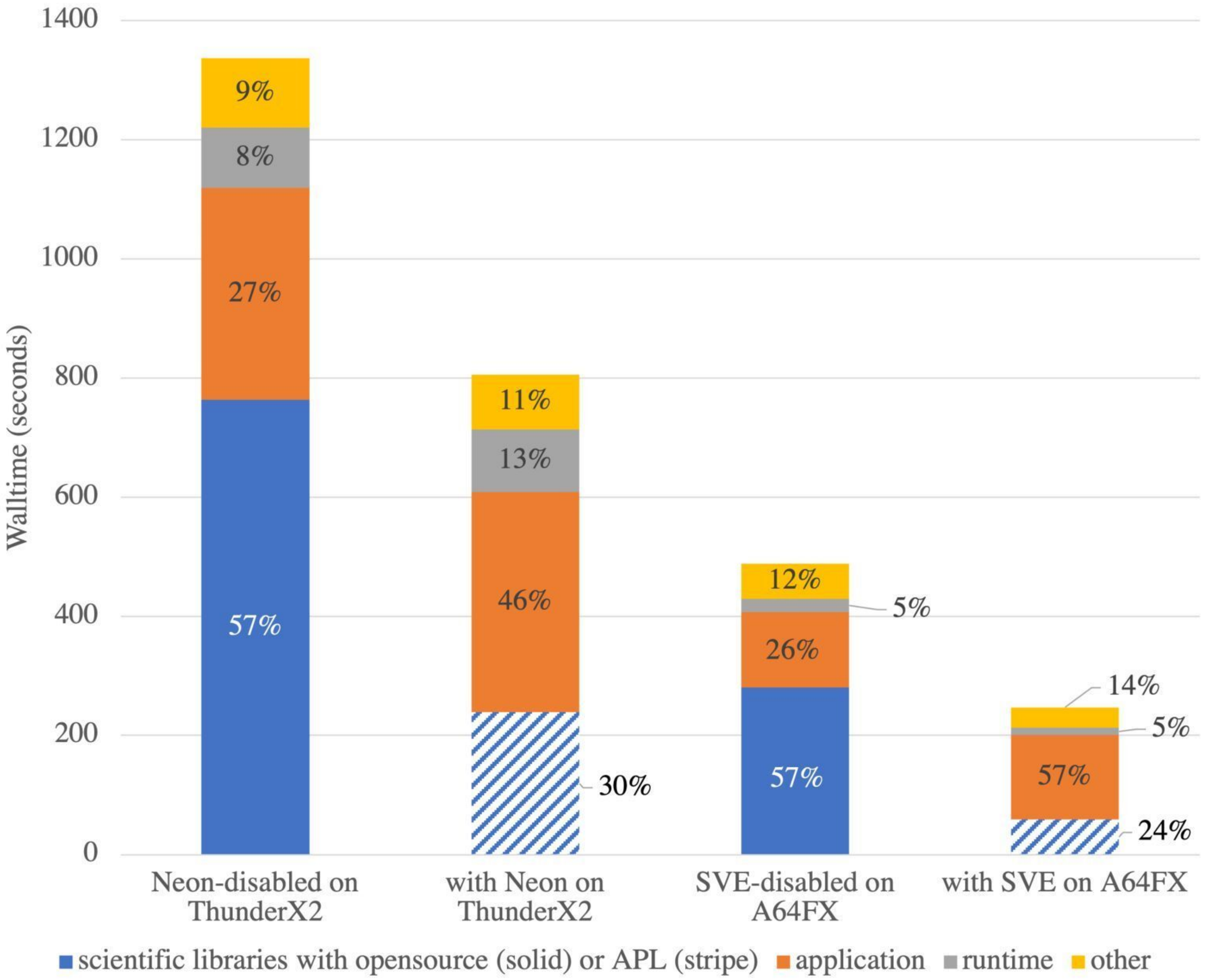
- Measure impact of SVE vectorization on the application
  - Run with vectorization explicitly disabled using netlib LAPACK and FFTW compiled without vectorization
  - Run with vectorization enabled using the Arm Performance Library

- Run on both ThunderX2 with NEON and an A64FX with SVE

- Testing done with 48 walker / accumulator threads with 100K iterations

	vectorization	walltime (seconds) ± standard deviation	speedup	Gflop/s
A64fx	no	488.42±3.09	-	17
	yes	246.98±0.48	1.98	78
ThunderX2	no	1336.61±178.09	-	14
	yes	805.53±24.06	1.66	27

Timing Breakdown



OAK RIDGE
National Laboratory

Timing Breakdown

- Remaining time is spent in the application code
- Efforts should be focusing on identifying SIMD parallelism in the application

Efficiently Porting DCA++ Code to A64FX

- A64FX performance relies heavily on vectorization
- Need to identify important loops not being vectorized by the compiler
- The Arm compiler is based on LLVM so we can use LLVM tools

Compiler Diagnostics

- LLVM provides diagnostics for its vectorization pass

```
$ armclang -O2 -Rpass-analysis=loop-vectorize -Rpass-missed=loop-vectorize code.c
```

- This will generate a lot of output, e.g.

```
$ make 2>&1| grep -e 'remark: loop not vectorized' | sort | uniq | wc -l
3335
```

- Find a way to filter out the unimportant ones

Using Profile Guided Optimization

- Build code with instrumentation

```
$ armclang -O2 -fprofile-instr-generate=profile-%m.profraw code.c -o code
```

- Run and convert the raw profile data for all the runs

```
$ ./code && llvm-profdata merge -output=code.profdata profile-*.profraw
```

- Use for the next compilation

```
$ armclang -O2 -fprofile-instr-use=code.profdata -o code
```

- Profile data provides "hotness" information

An LLVM-Based Methodology for Efficient Vectorization

- Identify only the hot loops that were not vectorized

- Generate profile information for the application with PGO
- Get hotness information for the diagnostics

  -fprofile-instr-use=code.profdata
  -fdiagnostics-show-hotness
  -fdiagnostics-hotness-threshold=100000
  -Rpass-analysis=loop-vectorize -Rpass-missed=loop-vectorize

- Sort remarks from hottest to coldest and examine

Reduction Loop Example

Problems

- IEEE Floating point numbers are not commutative
- Parallel reductions reorder the operations
- IEEE compliance must be explicitly disabled

```
for (int i = 0; i < j; i++)
  x_val −= x_ptr[i] ∗ G_ptr[i]
```

remark: loop not vectorized: cannot prove it is safe to reorder floating−point operations; allow reordering by specifying ’#pragma clang loop vectorize(enable)’ before the loop or by providing the compiler option ’−ffast−math’

Reduction Loop Example

Solution

- Explicitly enable relaxed IEEE semantics
- OpenMP SIMD supports explicit reductions to make intent clear
- Cross-platform

```
#pragma omp simd reduction(−:x_val)
for (int i = 0; i < j; i++)
  x_val −= x_ptr[i] ∗ G_ptr[i]
```

remark: vectorized loop

Gather Loop Example

Problems

- This loop performs a non-continuous load from memory, a gather.
- SVE supports fast gathering operations
- The compiler cannot statically determine the access bounds
- If pointer aliasing is present vectorization will create incorrect results
- Pointer aliasing can be checked at runtime if the bounds are statically known
- Compiler cannot statically determine the array bounds for this gather

```
for (int j = start_index_right[orb_j]; j < end_index_right[orb_j]; ++j) {
  const int out_j = j - start_index_right[orb_j];
  for (int i = start_index_left[orb i]; i < end_index_left[orb_i]; ++i) {
    const int out_i = i - start_index_left [orb_i];
    M_ij(out_i, out_j) = M(config_left[i].idx, config_right[j].idx);
  }
}
```

remark: loop not vectorized: Unknown array bounds

Gather Loop Example

Solution

- We can use OpenMP SIMD to assert that pointer aliasing doesn't occur
- Must be verified by the user that the two arrays do not overlap

```
for (int j = start_index_right[orb_j]; j < end_index_right[orb_j]; ++j) {
  const int out_j = j - start_index_right[orb_j];
  #pragma omp simd
  for (int i = start_index_left[orb i]; i < end_index_left[orb_i]; ++i) {
    const int out_i = i - start_index_left [orb_i];
    M_ij(out_i, out_j) = M(config_left[i].idx, config_right[j].idx);
  }
}
```

remark: vectorized loop:

Math Library Example

Problems

- This loop contains calls to math library functions
- Calls to functions cannot be vectorized unless a special vectorized version is provided
- Remarks suggest using **-ffast-math** or **-fno-math-errno** for relaxed error handling
- This will only allow the loop to be vectorized, without a math library the function calls will not be vectorized

```
for (int j = 0; j < n_v; ++j) {
  for (int i = 0; i < n_w; ++i) {
    const ScalarType x = configuration[j].get tau() * w_[i];
    T_[0](i, j) = std::cos(x);
    T_[1](i, j) = std::sin(x);
  }
}
```

remark: loop not vectorized : library call cannot be vectorized. Try compiling with −fno−math−errno, −ffast−math, -fsimdmath or similar flags

OAK RIDGE National Laboratory

Math Library Example

Solution

- Compile with **-fsimdmath** on Arm or **-fveclib=libmvec** using LLVM
- This tells the compiler to use the vectorized math-library

```
for (int j = 0; j < n_v; ++j) {
#pragma omp simd
  for (int i = 0; i < n_w; ++i) {
    const ScalarType x = configuration[j].get tau() * w_[i];
    T_[0](i, j) = std::cos(x);
    T_[1](i, j) = std::sin(x);
  }
}

remark: vectorized loop.
```

Loop Transformation

Problems

- This loop cannot be vectorized efficiently with being transformed
- The matrices are stored in **column-major** while this loop iterates across a **row**
- This creates non-contiguous memory accesses
- Each conditional must be computed unconditionally and selected
- The diagonal update is uncommon, but computed every time

```
for (int i = 0; i < Gamma.Rows(); i++) {
  for (int j = 0; j < Gamma.Cols(); j++) {
    int spin_idx i = random_vertex_vector[i];
    int spin_idx j = random_vertex_vector[j];

    if (spin_idx_j < vertex_index) {
      Real delta = (spin_idx_i == spin_idx_j) ? 1. : 0.;
      Real N_ij = N(spin_idx_i, spin_idx_j);
      Gamma(i, j) = (N_ij * exp_V[j] - delta) / (exp_V[j] - 1.);
    } else
      Gamma(i, j) = G_precomputed(spin_idx_i, spin_idx_j -
                                               vertex_index);
    if (i == j) {
      Real gamma_k = exp_delta_V[j];
      Gamma(i, j) -= (gamma_k) / (gamma_k - 1.);
    }
  }
}
```

Loop Transformation

Problems after Transformation

- Transposed loop for contiguous memory accesses
- Hoisted conditional to the end of the loop
- Contains a gather so still cannot be vectorized automatically
- Potential Division by zero prevents conditional masking

remark: loop not vectorized: Unknown array bounds

remark: loop not vectorized: Control flow cannot be substituted for a select

```
for (int j = 0; j < Gamma.Cols(); j++) {
  for (int i = 0; i < Gamma.Rows(); i++) {
    int spin_idx i = random_vertex_vector[i];
    int spin_idx j = random_vertex_vector[j];

    if (spin_idx_j < vertex_index) {
      Real delta = (spin_idx_i == spin_idx_j) ? 1. : 0.;
      Real N_ij = N(spin_idx_i, spin_idx_j);
      Gamma(i, j) = (N_ij * exp_V[j] - delta) / (exp_V[j] - 1.);
    } else
      Gamma(i, j) = G_precomputed(spin_idx_i, spin_idx_j -
                                  vertex_index);
  }
  Real gamma_k = exp_delta_V[j];
  Gamma(j, j) -= (gamma_k) / (gamma_k - 1.);
}
```

Loop Transformation

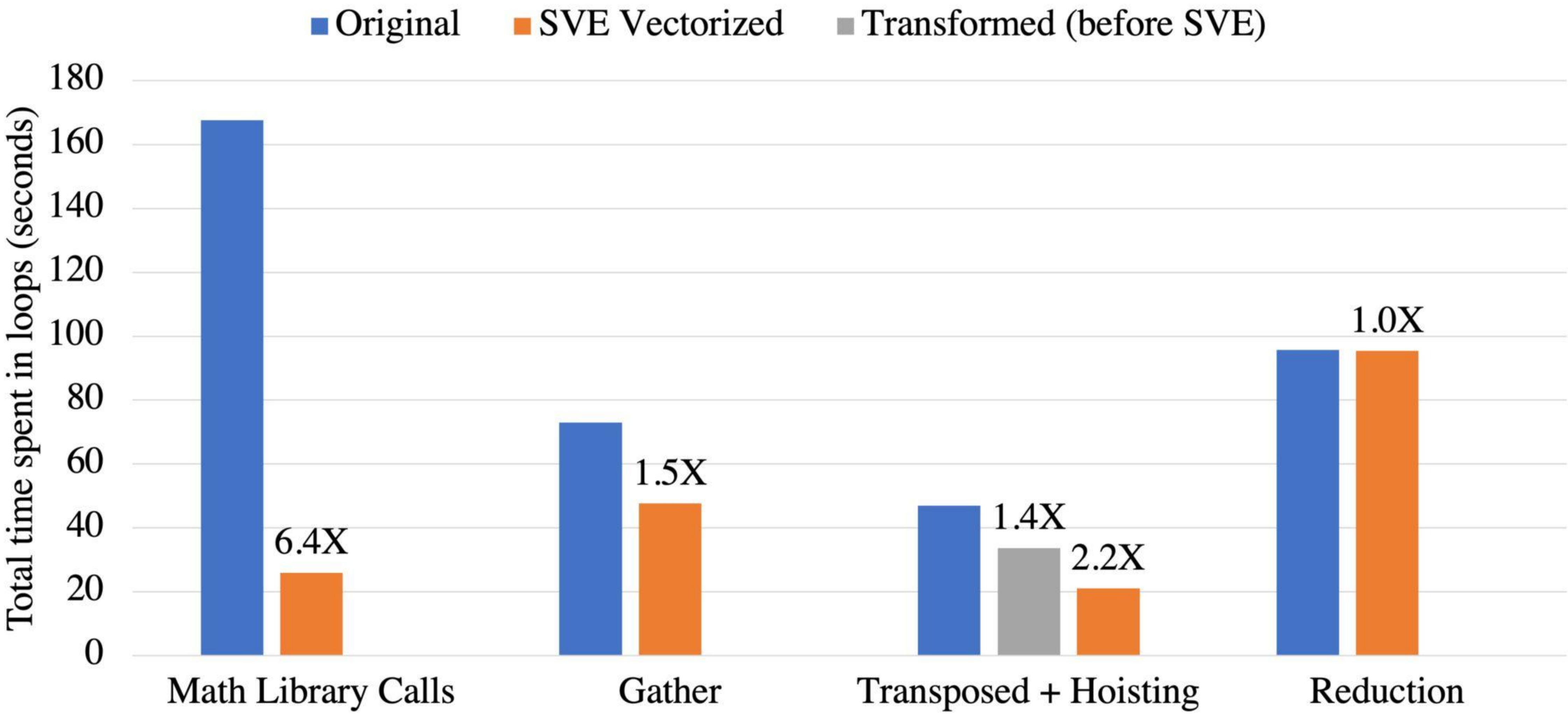
Solution

- Use OpenMP SIMD to prevent aliasing
- Division by zero can be ignored by compiling with **-ffast-math** or using OpenMP SIMD.

```
for (int j = 0; j < Gamma.Cols(); j++) {
  #pragma omp simd
  for (int i = 0; i < Gamma.Rows(); i++) {
    int spin_idx i = random_vertex_vector[i];
    int spin_idx j = random_vertex_vector[j];

    if (spin_idx_j < vertex_index) {
      Real delta = (spin_idx_i == spin_idx_j) ? 1. : 0.;
      Real N_ij = N(spin_idx_i, spin_idx_j);
      Gamma(i, j) = (N_ij * exp_V[j] - delta) / (exp_V[j] - 1.);
    } else
      Gamma(i, j) = G_precomputed(spin_idx_i, spin_idx_j -
                                              vertex_index);
  }
  Real gamma_k = exp_delta_V[j];
  Gamma(j, j) -= (gamma_k) / (gamma_k - 1.);
}
remark: vectorized loop
```

Loop Level Timing Results



Going Further: Assumptions

Assumptions

- LLVM supports built-in assumptions that can generate more efficient code
- This is not portable

```
void foo(double *X, int N) {
    __builtin_assume(N > 32 && N % 32 == 0);
    for (int i = 0; i < N; ++i)
        X[i] = X[i] * X[i];
}
```

- OpenMP 5.1 assumptions should allow this to be made portable.
- This is not implemented in LLVM yet.

```
void foo(double *X, int N) {
#pragma omp assume holds(N > 32 && N % 32 == 0);
    for (int i = 0; i < N; ++i)
        X[i] = X[i] * X[i];
}
```

Conclusions & Future Work

- Improved performance portability for DCA++ using OpenMP and LLVM tools
- Used profiling and diagnostics to filter for candidate loops

- This workflow could be automated, creating an LLVM-based vectorization Advisor tool
- Improve remarks and OpenMP support in the LLVM framework
  - Create more documentation to explain remarks like LLVM's OpenMPOpt

Questions?