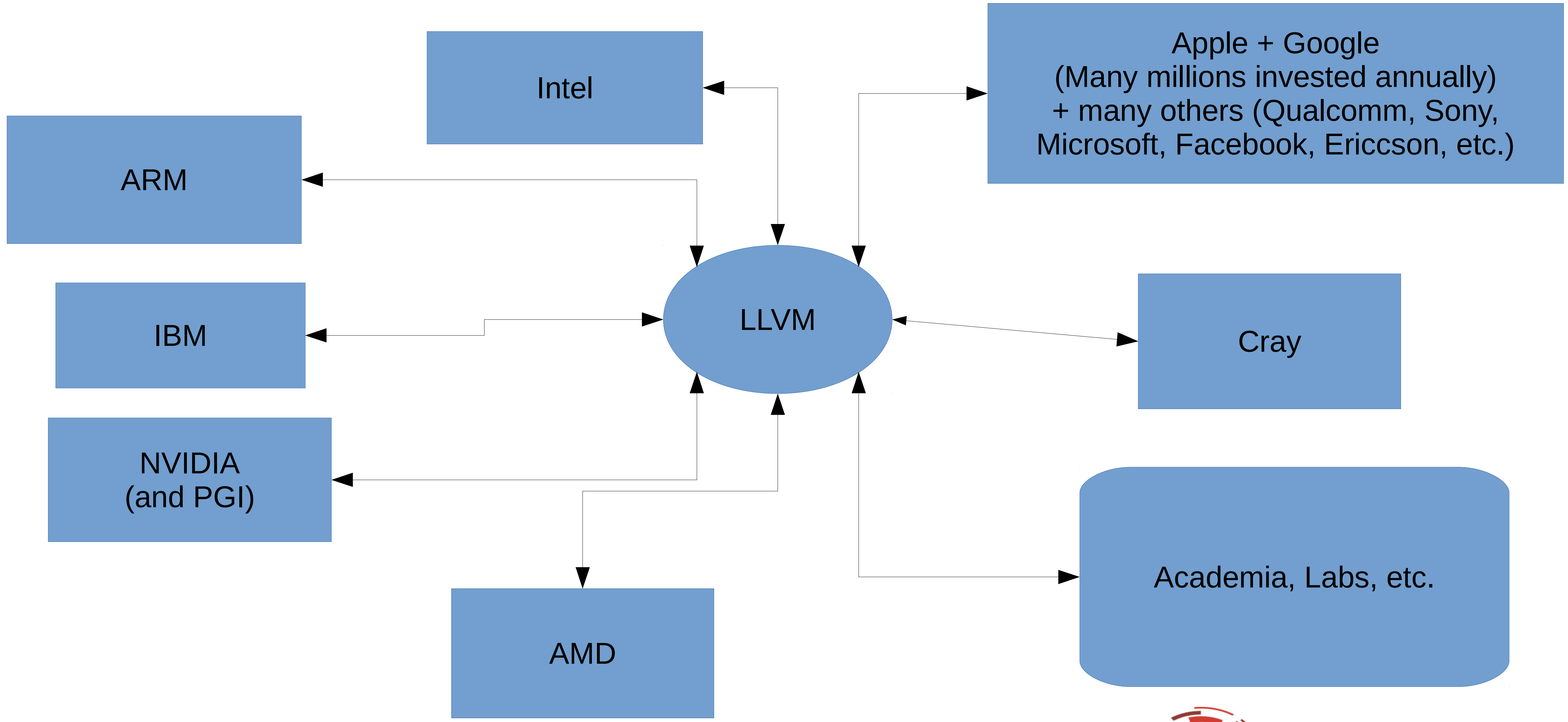
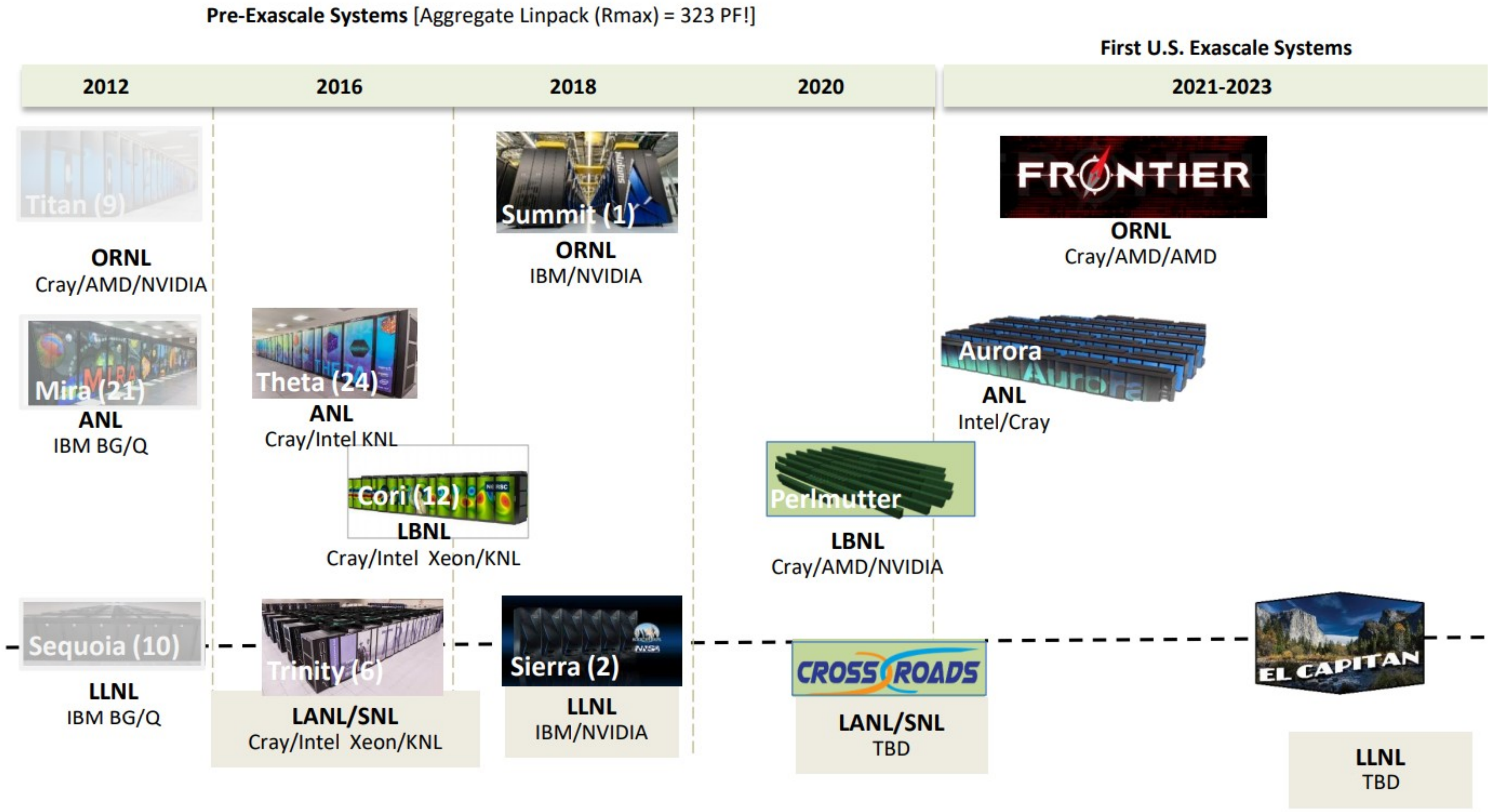
Thoughts on LLVM in DOE/ECP and MLIR

Hal Finkel
Leadership Computing Facility
Argonne National Laboratory

MLIR4HPC
10/21/2019

A role in exascale? Current/Future HPC vendors are already involved (plus many others)...


Intel
ARM
IBM
NVIDIA (and PGI)
AMD
LLVM
Apple + Google (Many millions invested annually) + many others (Qualcomm, Sony, Microsoft, Facebook, Ericcson, etc.)
Cray
Academia, Labs, etc.



(https://science.osti.gov/-/media/ascr/ascac/pdf/meetings/201909/20190923_ASCAC-Helland-Barbara-Helland.pdf)

ECP ST Projects Developing LLVM-Based Technology

SOLLVE: OpenMP (WBS 2.3.1.13)

- Enhancing the implementation of OpenMP in LLVM:
  - Developing support for unified memory (e.g., from NVIDIA), kernel decomposition and pipelining, automated use of local memory, and other enhancements for accelerators.
  - Developing optimizations of OpenMP constructs to reduce overheads (e.g., from thread startup and barriers).
    - Building on LLVM parallel-IR work in collaboration with Intel.
- Using LLVM, Clang, and Flang to prototype new OpenMP features for standardization.
- Developing an OpenMP test suite, and as a result, testing and improving the quality of OpenMP in LLVM, Clang, and Flang.

PROTEAS: Parallel IR & More (WBS 2.3.2.09)

- Developing extensions to LLVM's intermediate representation (IR) to represent parallelism.
  - Strong collaboration with Intel and several academic groups.
  - Parallel IR can target OpenMP's runtime library among others.
  - Parallel IR can be targeted by OpenMP, OpenACC, and other programming models in Clang, Flang, and other frontends.
  - Building optimizations on parallel IR to reduce overheads (e.g., merging parallel regions and removing redundant barriers).
- Developing support for OpenACC in Clang, prototyping non-volatile memory features, and integration with Tau performance tools.

Y-Tune: Autotuning (WBS 2.3.2.07)

- Enhancing LLVM to better interface with autotuning tools.
- Enhancing LLVM's polyhedral loop optimizations and the ability to drive them using autotuning.
- Using Clang, and potentially Flang, for parsing and semantic analysis.

Kitsune: LANL ATDM Dev. Tools (WBS 2.3.2.02)

- Using parallel IR to replace template expansion in FleCSI, Kokkos, RAJA, etc.
- Enhanced parallel-IR optimizations and targeting of various runtimes/architectures.
- Flang evaluation, testing, and Legion integration, plus other programming-model enhancements.
- ByFl: Instrumentation-based performance counters using LLVM.

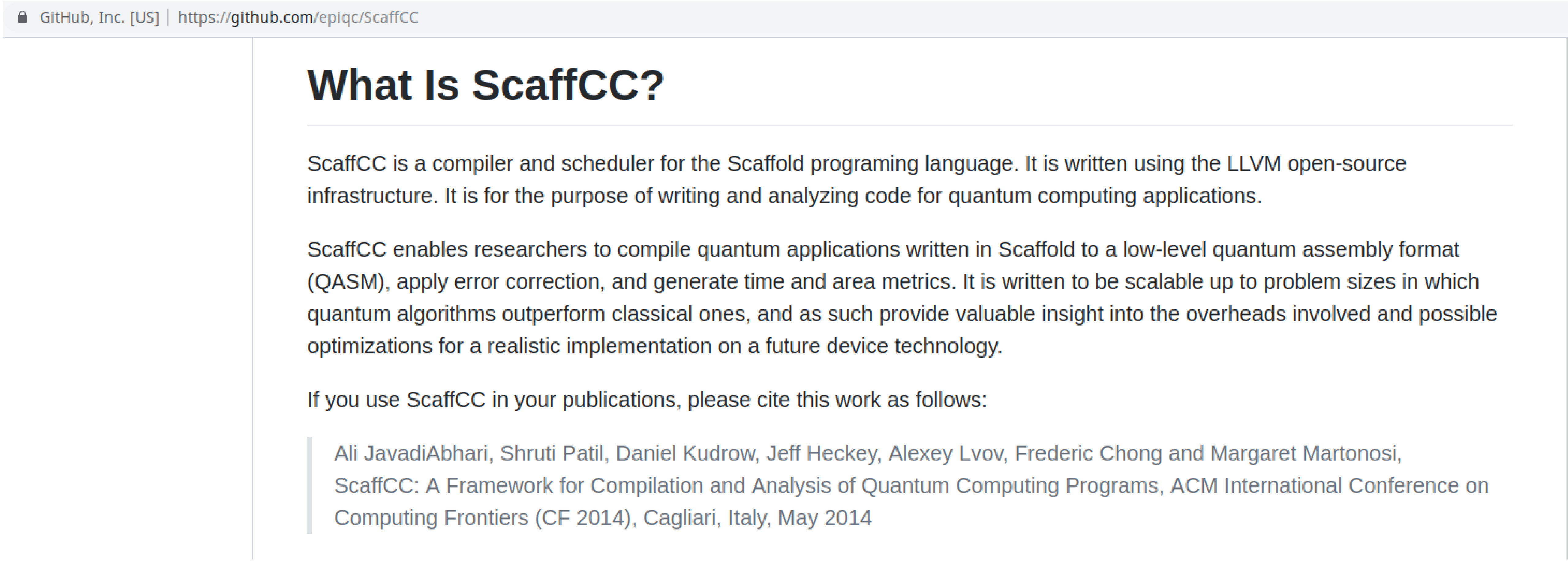
Flang: LLVM Fortran Frontend (WBS 2.3.5.06)

- Working with NVIDIA (PGI), ARM, and others to develop an open-source, production-quality LLVM Fortran frontend.
  - Can target parallel IR to support OpenMP (including OpenMP offloading) and OpenACC.

Note: The proxy-apps project (WBS 2.2.6.01) is also enhancing LLVM's test suite.

EXASCALE COMPUTING PROJECT

And Even in Quantum Computing



GitHub, Inc. [US] | https://github.com/epiqc/ScaffCC

What Is ScaffCC?

ScaffCC is a compiler and scheduler for the Scaffold programing language. It is written using the LLVM open-source infrastructure. It is for the purpose of writing and analyzing code for quantum computing applications.

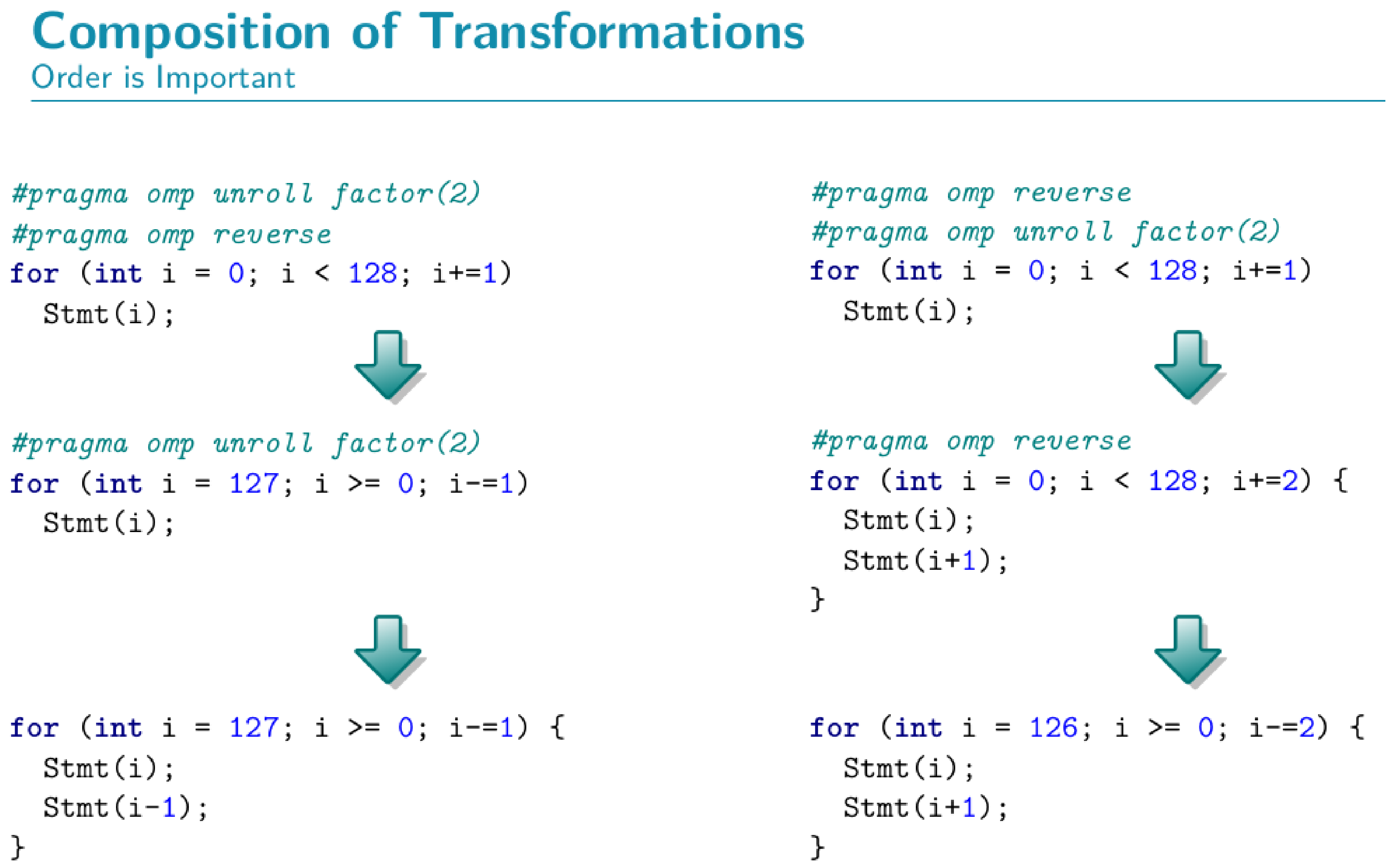
ScaffCC enables researchers to compile quantum applications written in Scaffold to a low-level quantum assembly format (QASM), apply error correction, and generate time and area metrics. It is written to be scalable up to problem sizes in which quantum algorithms outperform classical ones, and as such provide valuable insight into the overheads involved and possible optimizations for a realistic implementation on a future device technology.

If you use ScaffCC in your publications, please cite this work as follows:

> Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic Chong and Margaret Martonosi, ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs, ACM International Conference on Computing Frontiers (CF 2014), Cagliari, Italy, May 2014

This is work by Fred Chong at UChicago (and collaborators).

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)



Composition of Transformations

Order is Important

```
#pragma omp unroll factor(2)
#pragma omp reverse
for (int i = 0; i < 128; i+=1)
  Stmt(i);
```

```
#pragma omp unroll factor(2)
for (int i = 127; i >= 0; i-=1)
  Stmt(i);
```

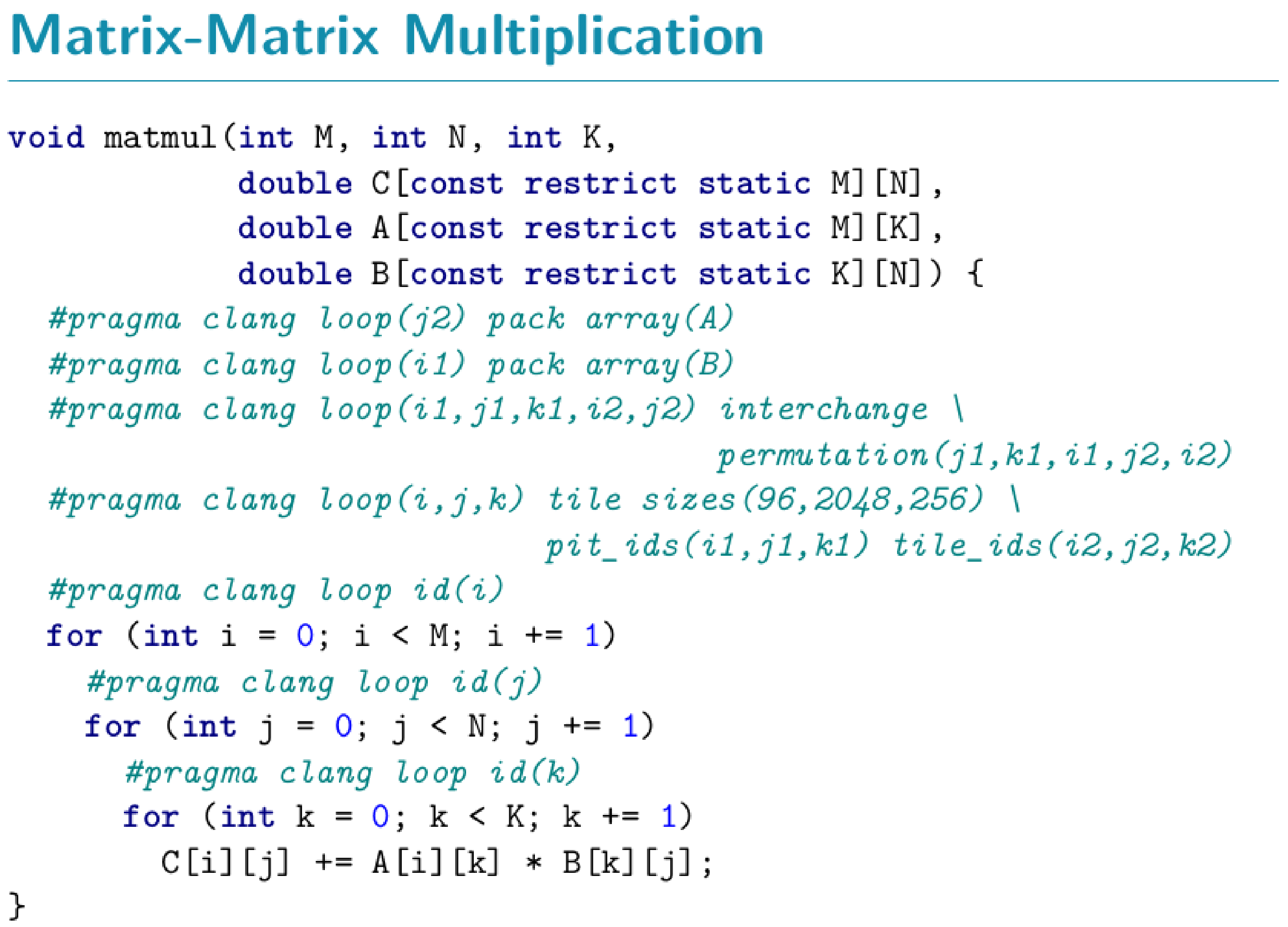
```
for (int i = 127; i >= 0; i-=1) {
  Stmt(i);
  Stmt(i-1);
}
```

```
#pragma omp reverse
#pragma omp unroll factor(2)
for (int i = 0; i < 128; i+=1)
  Stmt(i);
```

```
#pragma omp reverse
for (int i = 0; i < 128; i+=2) {
  Stmt(i);
  Stmt(i+1);
}
```

```
for (int i = 126; i >= 0; i-=2) {
  Stmt(i);
  Stmt(i+1);
}
```

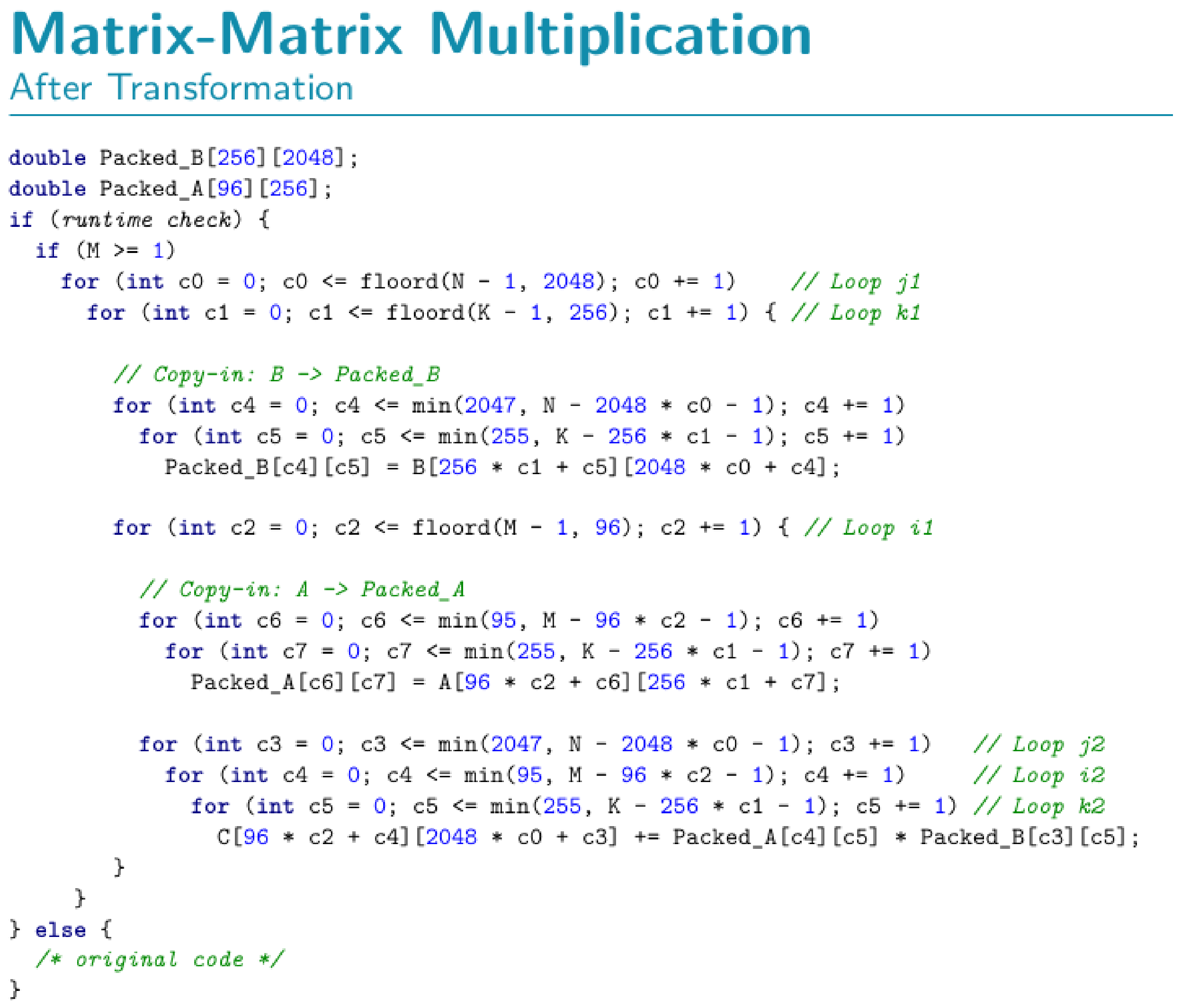
Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)



Matrix-Matrix Multiplication

```
void matmul(int M, int N, int K,
            double C[const restrict static M][N],
            double A[const restrict static M][K],
            double B[const restrict static K][N]) {
  #pragma clang loop(j2) pack array(A)
  #pragma clang loop(i1) pack array(B)
  #pragma clang loop(i1,j1,k1,i2,j2) interchange \
                                   permutation(j1,k1,i1,j2,i2)
  #pragma clang loop(i,j,k) tile sizes(96,2048,256) \
                          pit_ids(i1,j1,k1) tile_ids(i2,j2,k2)
  #pragma clang loop id(i)
  for (int i = 0; i < M; i += 1)
    #pragma clang loop id(j)
    for (int j = 0; j < N; j += 1)
      #pragma clang loop id(k)
      for (int k = 0; k < K; k += 1)
        C[i][j] += A[i][k] * B[k][j];
}
```

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)



Matrix-Matrix Multiplication

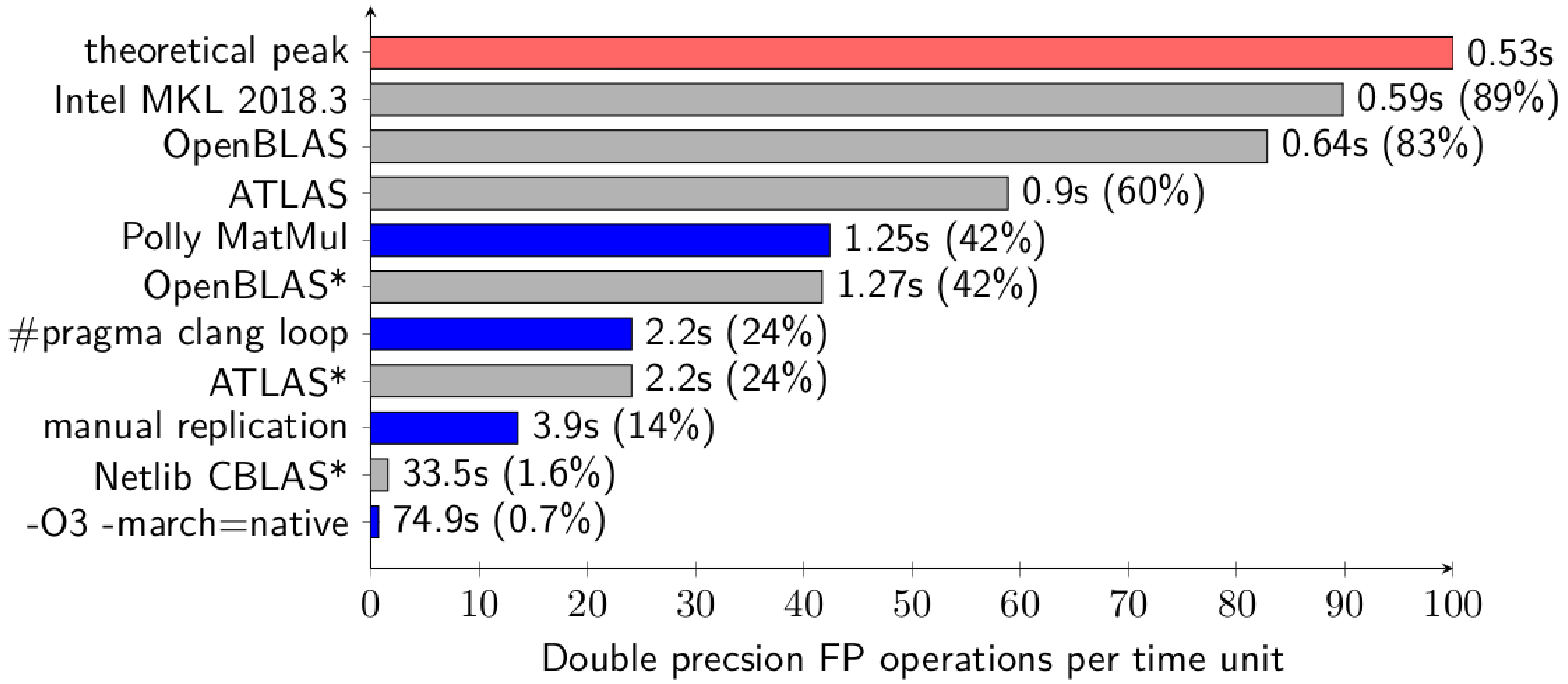
After Transformation

```
double Packed_B[256][2048];
double Packed_A[96][256];
if (runtime check) {
  if (M >= 1)
    for (int c0 = 0; c0 <= floord(N - 1, 2048); c0 += 1)     // Loop j1
      for (int c1 = 0; c1 <= floord(K - 1, 256); c1 += 1) { // Loop k1

        // Copy-in: B -> Packed_B
        for (int c4 = 0; c4 <= min(2047, N - 2048 * c0 - 1); c4 += 1)
          for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1)
            Packed_B[c4][c5] = B[256 * c1 + c5][2048 * c0 + c4];

        for (int c2 = 0; c2 <= floord(M - 1, 96); c2 += 1) { // Loop i1

          // Copy-in: A -> Packed_A
          for (int c6 = 0; c6 <= min(95, M - 96 * c2 - 1); c6 += 1)
            for (int c7 = 0; c7 <= min(255, K - 256 * c1 - 1); c7 += 1)
              Packed_A[c6][c7] = A[96 * c2 + c6][256 * c1 + c7];

          for (int c3 = 0; c3 <= min(2047, N - 2048 * c0 - 1); c3 += 1)    // Loop j2
            for (int c4 = 0; c4 <= min(95, M - 96 * c2 - 1); c4 += 1)      // Loop i2
              for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1) // Loop k2
                C[96 * c2 + c4][2048 * c0 + c3] += Packed_A[c4][c5] * Packed_B[c3][c5];
        }
      }
} else {
  /* original code */
}
```

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)

Matrix-Matrix Multiplication

Execution Speed



* Pre-compiled from Ubuntu repository

What To Do With OpenACC Code?

Clacc: OpenACC Support for Clang and LLVM

Who

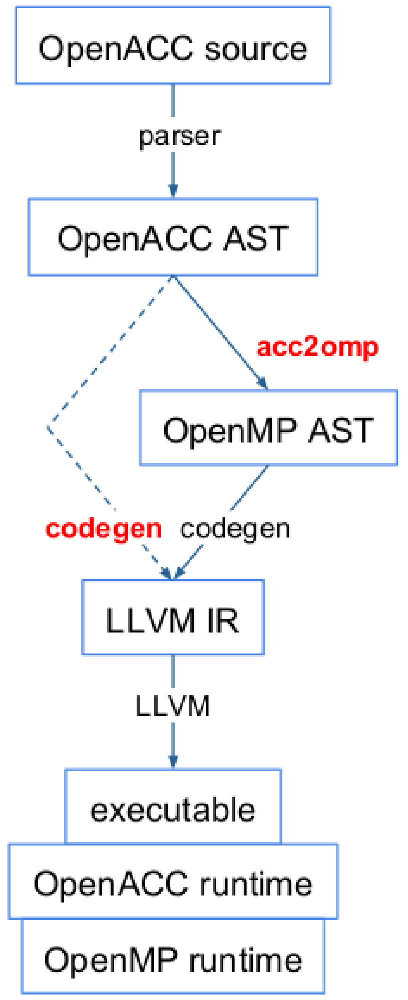
- Joel E. Denny (ORNL)
- Seyong Lee (ORNL)
- Jeffrey S. Vetter (ORNL)

Where

- Clacc: Translating OpenACC to OpenMP in Clang, Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Dallas, TX, USA, (2018).
- https://ft.ornl.gov/research/clacc
- Clacc Poster (Wed at ECP AHM)

What

- Develop production-quality, standard-conforming traditional OpenACC compiler and runtime support by extending Clang and LLVM
- Enable research and development of source-level OpenACC tools
    - Design compiler to leverage Clang/LLVM ecosystem extensibility
    - E.g., Pretty printers, analyzers, lint tools, and debugger and editor extensions
- As matures, contribute OpenACC support to upstream Clang and LLVM
- Throughout development
    - Actively contribute upstream all mutually beneficial Clang and LLVM improvements
    - Actively contribute to the OpenACC specification

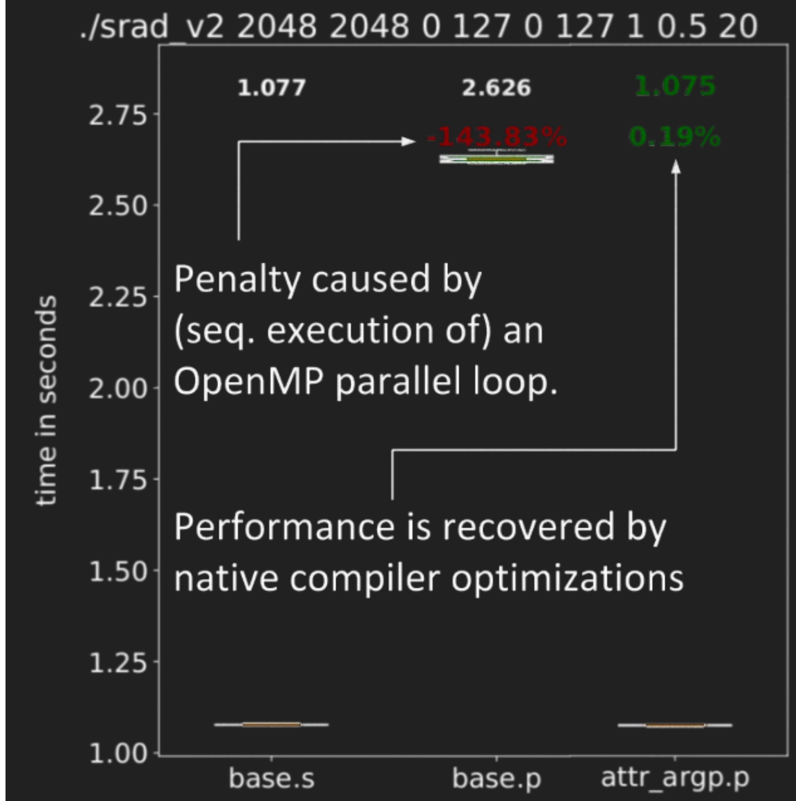


4

EXASCALE COMPUTING PROJECT

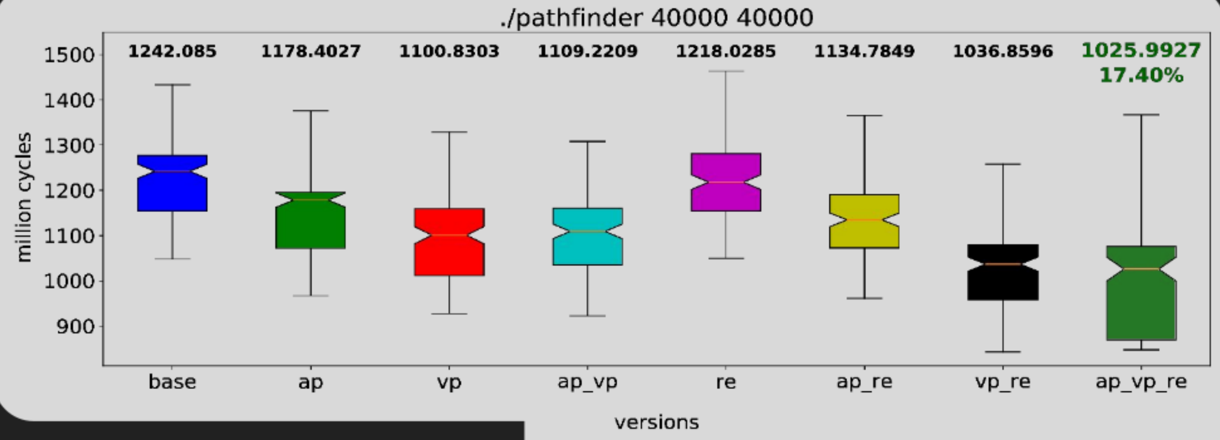
Optimization of Parallel Programs (OpenMP and Similar) (POC: Johannes Doerfert, ANL)

Optimizing Parallel Programs with LLVM



Penalty caused by (seq. execution of) an OpenMP parallel loop.

Performance is recovered by native compiler optimizations



Reuse "scalar" code

- constant propagation
- argument promotion
- attribute deduction
- ...

New "parallel" optimizations

- barrier elimination
- parallel region expansion
- parallelism aware code motion
- ...

See our IWOMP'18 & LCPC'18 papers, as well as the LLVMDev'18 talk/video!

EXASCALE COMPUTING PROJECT

Acknowledgments

Thanks to ALCF, ANL, ECP, DOE, and the LLVM community!

ALCF is supported by DOE/SC under contract DE-AC02-06CH11357.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.