Even Better C++ Performance and Productivity

Enhancing Clang to Support Just-in-Time Compilation of Templates

Hal Finkel
Leadership Computing Facility
Argonne National Laboratory
hfinkel@anl.gov

(https://www.publicdomainpictures.net/en/view-image.php?image=176106&picture=fast-sport-car)

Why JIT?

- Because you can’t compile ahead of time (e.g., client-side Javascript)

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <button id="hellobutton">Hello</button>
    <script>
        document.getElementById('hellobutton').onclick = function() {
            alert('Hello world!');                     // Show a dialog
            var myTextNode = document.createTextNode('Some new words.');
            document.body.appendChild(myTextNode);    // Append "Some new words" to the page
        };
    </script>
  </body>
</html>
```

(https://en.wikipedia.org/wiki/JavaScript)

Why JIT?

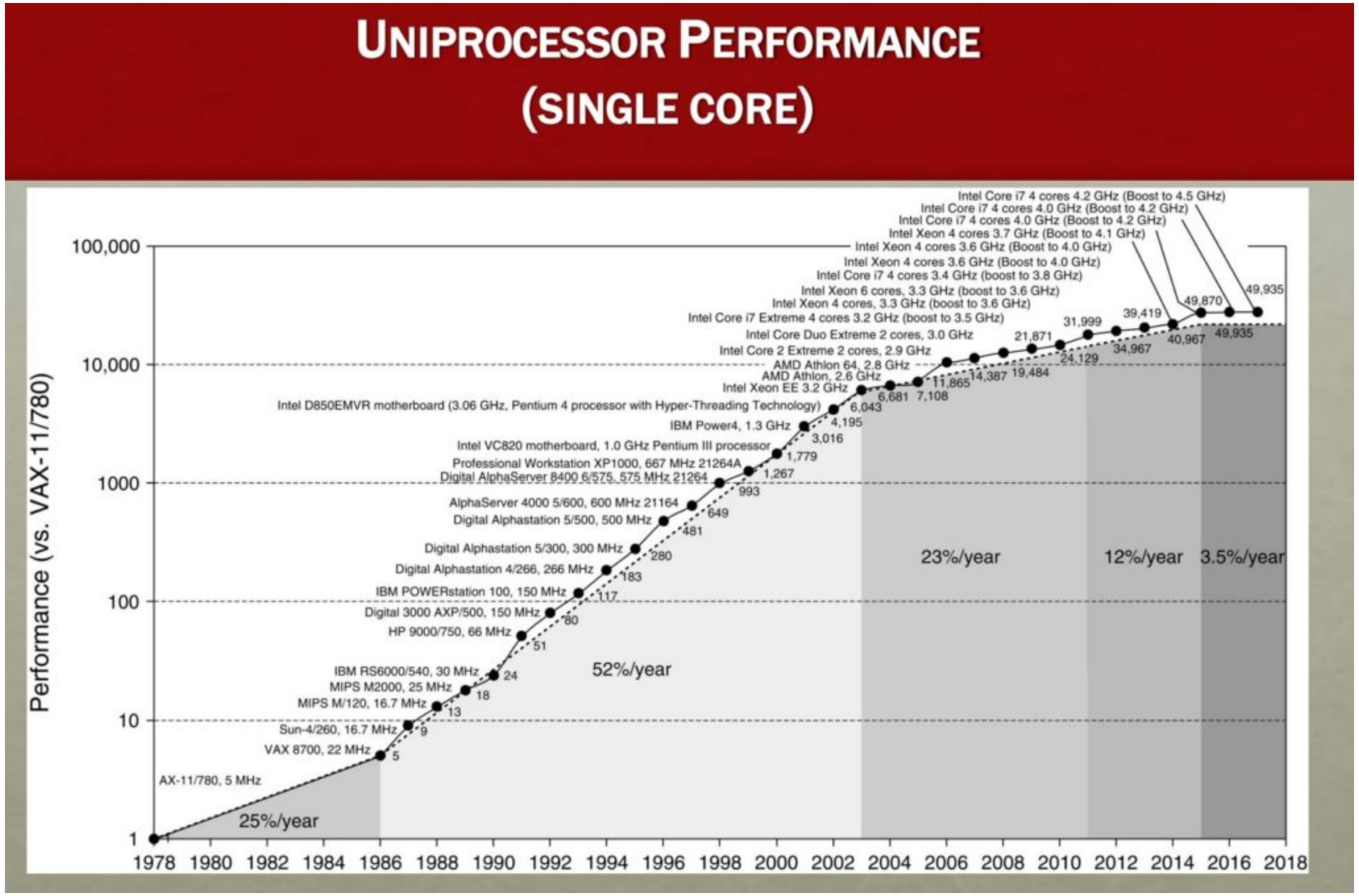
- To minimize time spent compiling ahead of time (e.g., to improve programmer productivity)

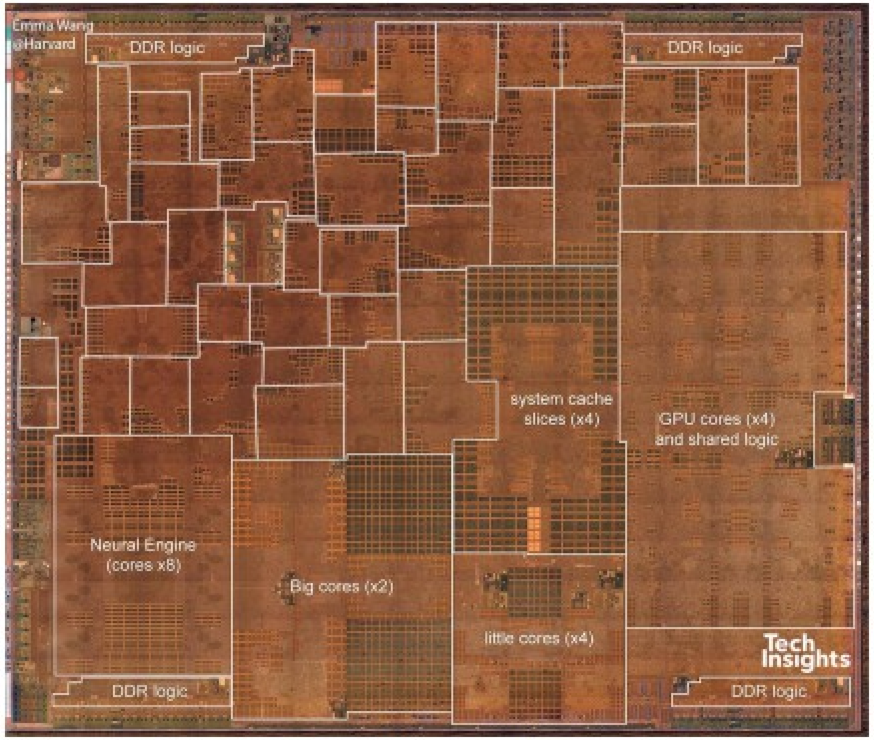
(https://www.pdclipart.org/displayimage.php?album=search&cat=0&pos=3)

Why JIT?

- To adapt/specialize the code during execution:
  - For performance
  - For non-performance-related reasons (e.g., adaptive sandboxing)

Why JIT? – Specialization and Adapting to Heterogeneous Hardware



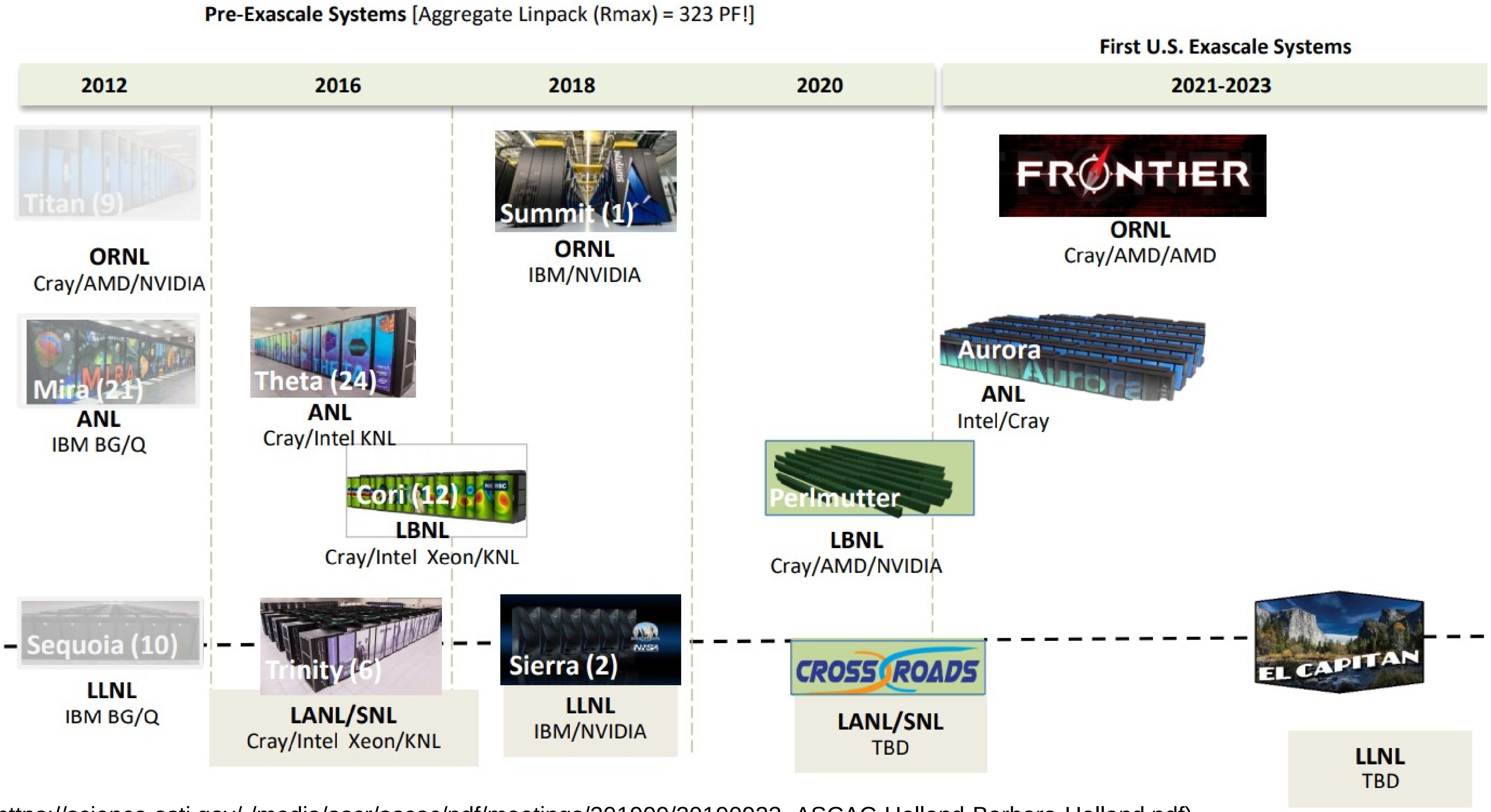


(c) **2019 Apple A12**
7 nm TSMC 83 mm^2
40+ accelerators

Figure 3: Three Apple iPhone SoCs with accelerators highlighted.[1]

(https://arxiv.org/pdf/1907.02064.pdf)

Why JIT? – Specialization and Adapting to Heterogeneous Hardware



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

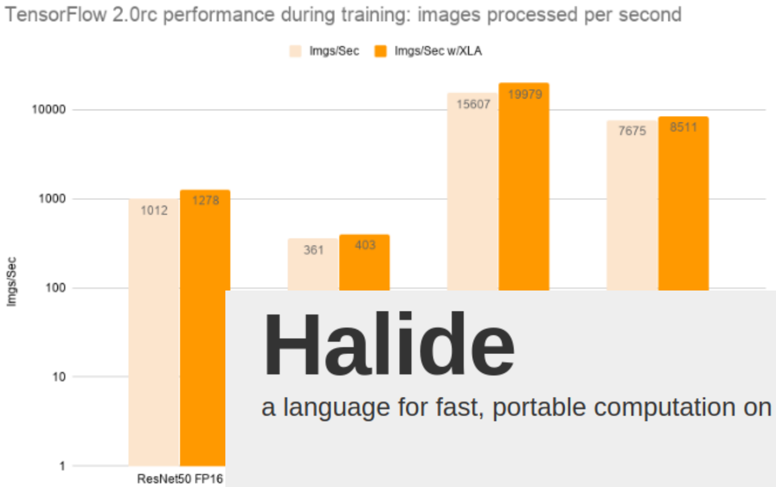
EXASCALE COMPUTING PROJECT

In C++, JITs Are All Around Us…

XLA: Optimizing Compiler for TensorFlow

XLA (Accelerated Linear Algebra) is a domain-specific compiler for linear algebra that accelerates TensorFlow models with potentially no source code changes.

The results are improvements in speed and memory usage: most internal benchmarks run ~1.15x faster after XLA is enabled. The dataset below is evaluated on a single NVidia V100 GPU:



PCRE Performance Project

About

The aim of PCRE-sljit project is speeding up the pattern matching speed of Perl Compatible Regular Expressions library (ftp download). The task is achieved by using sljit, a just-in-time (JIT) compilation library to translate machine code from the internal byte-code representation generated by *pcre_compile()*. PCRE-sljit …milar matching speed to DFA based engines (like re2) on common patterns but still keep PERL …ility (see here).

…k has been released as part of PCRE 8.20 and above. The JIT was improved a lot in 8.32, and a new so…**tive interface** was introduced. See the results.

ArrayFire

Performance of ArrayFire JIT Code Generation

ODED / FEBRUARY 18, 2015 /
ARRAYFIRE, BENCHMARKS, CASE STUDIES, INFRASTRUCTURE / 3 COMMENTS

The ArrayFire library offers JIT (Just In Time) compiling for standard arithmetic operations. This includes trigonometric functions, comparisons, and element-wise operations.

ClamAV blog

Friday, November 4, 2011

Bytecode signatures for polymorphic malware

About one year ago Alain presented the LLVM-based ClamAV bytecode. We've realised that, besides that initial introduction, we've never shown any real life use case, nor did we ever demonstrate the incredible power and flexibility of the ClamAV bytecode engine. I'll try to fix that today.

Halide

a language for fast, portable computation on images and tensors

Overview

Halide is a programming language designed to make it easier to write high-performance ima… processing code on modern machines. Halide currently targets:

- CPU architectures: X86, ARM, MIPS, Hexagon, PowerPC
- Operating systems: Linux, Windows, macOS, Android, iOS, Qualcomm QuRT
- GPU Compute APIs: CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal, Microsoft Direct X 12

Rather than being a standalone programming language, Halide is embedded in C++. This means you write C++ code that builds an in-memory representation of a Halide pipeline using Halide's C++ API. You can then compile this representation to an object file, or JIT-compile it and run it in the same process.

```
// 4. Perform runtime source compilation, and obtain kernel entry point.
cl_program program = clCreateProgramWithSource( context,
                                                1,
                                                &source,
                                                NULL, NULL );

clBuildProgram( program, 1, &device, NULL, NULL, NULL );

cl_kernel kernel = clCreateKernel( program, "memset", NULL );
```

(OpenCL)

Blend2D BETA
2D Vector Graphics Engine

High Performance 2D

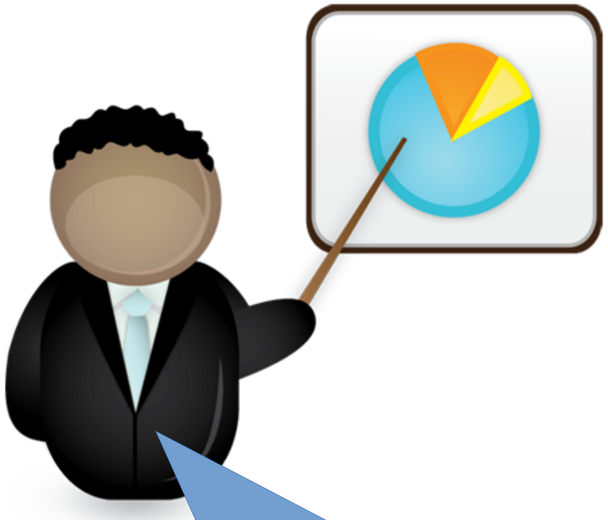
Blend2D is a high performance 2D vector graphics engine written in C++ and released under the Zlib license. It has a built-in JIT compiler that generates optimized pipelines at runtime. Additionally, the engine features a new rasterizer that has been written from scratch. It delivers superior performance while quality is comparable to rasterizers used by AGG and FreeType. The performance has been optimized by using an innovative approach to index data that is built during rasterization and scanned during composition. The rasterizer is robust and excels in rendering complex vector art and text.

EXASCALE COMPUTING PROJECT

In C++, JITs Are All Around Us...

But how many people know how to make one of these? And how portable are they?

We are good C++ programmers…
There are many of us!



I know how to make a
high-performance JIT…
I'm part of a smaller community.

In C++, JITs Are All Around Us...

Does writing a JIT today mean directly generating assembly instructions? Probably not. There are a number of frameworks supporting common architectures:

```
using NativeJIT::FunctionBuffer;

int main()
{
    // Create allocator and buffe
    ExecutionBuffer codeAllocator
    Allocator allocator(8192);
    FunctionBuffer code(codeAlloc

    // Create the factory for exp
    // Our area expression will t
    Function<float, float> expres

    // Multiply input parameter by itself to get radius squared.
    auto & rsquared = expression.Mul(expression.GetP1(), expression.G

    // Multiply by PI.
    const float  PI = 3.14159265358979f;
    auto & area = expression.Mul(rsquared, expression.Immediate(PI));

    // Compile expression into a function.
    auto function = expression.Compile(area);
```

https://github.com/BitFunnel/NativeJIT

```
// Create the true_block.
BasicBlock *RetBB = BasicBlock::Create(Context, "return", FibF);
// Create an exit block.
BasicBlock* RecurseBB = BasicBlock::Create(Context, "recurse", FibF);

// Create the "if (arg <= 2) goto exitbb"
Value *CondInst = new ICmpInst(*BB, ICmpInst::ICMP_SLE, ArgX, Two, "cond");
BranchInst::Create(RetBB, RecurseBB, CondInst, BB);

// Create: ret int 1
ReturnInst::Create(Context, One, RetBB);

// create fib(x-1)
Value *Sub = BinaryOperator::CreateSub(ArgX, One, "arg", RecurseBB);
CallInst *CallFibX1 = CallInst::Create(FibF, Sub, "fibx1", RecurseBB);
```

(LLVM)

C++ | **COAT**

```
condition ){
then_branch
```

```
coat::if_then(coat::Function&, condition, [&]{
    then_branch
});
```

```
condition ){
then_branch
se{
```

```
coat::if_then_else(coat::Function&, condition, [&]{
    then_branch
}, [&]{
```

```
ction&, condition, [&]{
```

```
ion&, [&]{
```

```
(1024);
ta.end(), 0);

Jit in this case
ted function
using func_t = uint64_t (*)(uint64_t *data, uint64_t size);
// context object representing the generated function
coat::Function<coat::runtimeasmjit,func_t> fn(&asmrt);
// start of the EDSL code describing the code of the generated function
{
    // get function arguments as "meta-variables"
    auto [data,size] = fn.getArguments("data", "size");

    // "meta-variable" for sum
    coat::Value sum(fn, uint64_t(0), "sum");
    // "meta-variable" for past-the-end pointer
    auto end = data + size;
    // loop over all elements
    coat::for_each(fn, data, end, [&](auto &element){
        // add each element to the sum
        sum += element;
    });
    // specify return value
    coat::ret(fn, sum);
}
// finalize code generation and get function pointer to the generated function
func_t foo = fn.finalize(&asmrt);
```

But you will write code that writes the code, one operation and control structure at a time.

https://tetzank.github.io/posts/coat-edsl-for-codegen/
(A wrapper for LLVM)

ClangJIT - A JIT for C++

Some basic requirements…

- As-natural-as-possible integration into the language.

- JIT compilation should not access source files (or other ancillary files) during program execution.

(https://www.pdclipart.org/displayimage.php?album=search&cat=0&pos=0)

- JIT compilation should be as incremental as possible: don't repeat work unnecessarily.

(https://www.pdclipart.org/displayimage.php?album=search&cat=0&pos=38)

ClangJIT - A JIT for C++

https://github.com/hfinkel/llvm-project-cxxjit/wiki

hfinkel / **llvm-project-cxxjit**

Unwatch 12 | Star 103 | Fork 9

Code | Issues 6 | Pull requests 1 | Projects 0 | Wiki | Security | Insights | Settings

Home

Edit | New Page

Hal Finkel edited this page now · 20 revisions

Welcome to the llvm-project-cxxjit wiki! This is a fork of LLVM with a Clang enhanced with just-in-time (JIT) compilation functionality.

For more information on the implementation and some evaluation, see: https://arxiv.org/abs/1904.08555

Getting Started

To install ClangJIT, clone https://github.com/hfinkel/llvm-project-cxxjit and build as you would Clang/LLVM for your system.

If you're building on a Linux system, then a basic CMake configuration such as the following will likely work:

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DBUILD_SHARED_LIBS=ON -DLLVM_USE_SPLIT_DWARF
```

and then run `make` (using `-j<number of cores` to get a parallel build) and then `make install`. Before you install, you might wish to run LLVM's and Clang's regression tests (which include regression tests associated with the JIT functionality). `make -j<number of cores> check-llvm check-clang` should do that. Once installed, you can then use the `/path/to/somewhere/bin/clang++` with the `-fjit` option as described below.

Pages 1

Find a Page…

Home

+ Add a custom sidebar

Clone this wiki locally

https://github.com/hfinke

ClangJIT - A JIT for C++

ClangJIT provides an underlying code-specialization capability driven by templates (our existing feature for programming-controlled code specialization). It allows both values and types to be provided as runtime template arguments to function templates with the [[clang::jit]] attribute:

```
#include <iostream>
#include <cstdlib>

template <int x>
[[clang::jit]] void run() {
  std::cout << "Hello, World, I was compiled at runtime, x = " << x << "\n";
}

int main(int argc, char *argv[]) {
  int a = std::atoi(argv[1]);
  run<a>();
}
```

ClangJIT - A JIT for C++

Types as strings (integration with RTTI would also make sense, but this allows types to be composed from configuration files, etc.):

```cpp
#include <iostream>

struct F {
  int i;
  double d;
};

template <typename T, int S>
struct G {
  T arr[S];
};

template <typename T>
[[clang::jit]] void run() {
  std::cout << "I was compiled at runtime, sizeof(T) = " << sizeof(T) << "\n";
}

int main(int argc, char *argv[]) {
  std::string t(argv[1]);
  run<t>();
}
```


EXASCALE COMPUTING PROJECT

ClangJIT - A JIT for C++

```
$ clang++ -O3 -fjit -o /tmp/jit-t /tmp/jit-t.cpp
$ /tmp/jit-t '::F'
I was compiled at runtime, sizeof(T) = 16
$ /tmp/jit-t 'F'
I was compiled at runtime, sizeof(T) = 16
$ /tmp/jit-t 'float'
I was compiled at runtime, sizeof(T) = 4
$ /tmp/jit-t 'double'
I was compiled at runtime, sizeof(T) = 8
$ /tmp/jit-t 'size_t'
I was compiled at runtime, sizeof(T) = 8
$ /tmp/jit-t 'std::size_t'
I was compiled at runtime, sizeof(T) = 8
$ /tmp/jit-t 'G<F, 5>'
I was compiled at runtime, sizeof(T) = 80
```

ClangJIT - A JIT for C++

Semantic properties of the [[clang::jit]] attribute:

- Instantiations of this function template will not be constructed at compile time, but rather, calling a specialization of the template, or taking the address of a specialization of the template, will trigger the instantiation and compilation of the template during program execution.
- Non-constant expressions may be provided for the non-type template parameters, and these values will be used during program execution to construct the type of the requested instantiation. For const array references, the data in the array will be treated as an initializer of a constexpr variable.
- Type arguments to the template can be provided as strings. If the argument is implicitly convertible to a const char *, then that conversion is performed, and the result is used to identify the requested type. Otherwise, if an object is provided, and that object has a member function named c_str(), and the result of that function can be converted to a const char *, then the call and conversion (if necessary) are performed in order to get a string used to identify the type. The string is parsed and analyzed to identify the type in the declaration context of the parent to the function triggering the instantiation. Whether types defined after the point in the source code that triggers the instantiation are available is not specified.

ClangJIT - A JIT for C++

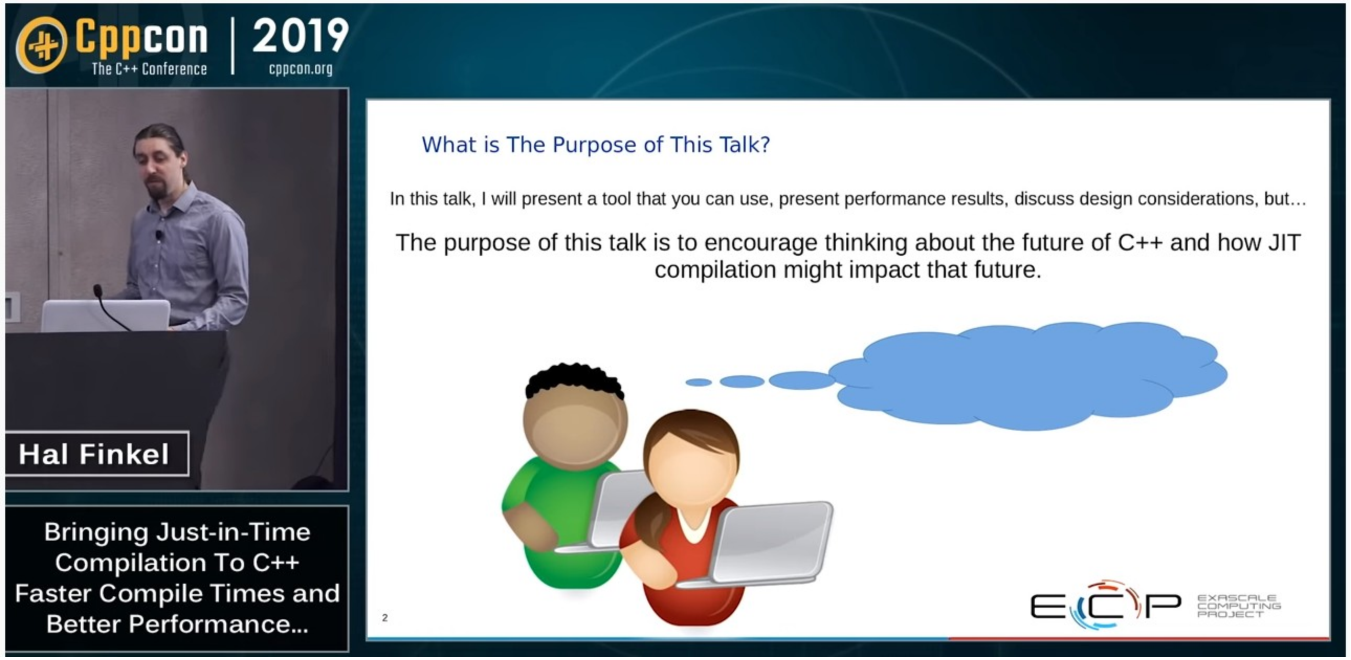
Some restrictions on the use of function templates with the [[clang::jit]] attribute:

- Because the body of the template is not instantiated at compile time, decltype(auto) and any other type-deduction mechanisms depending on the body of the function are not available.
- Because the template specializations are not compiled until during program execution, they're not available at compile time for use as non-type template arguments, etc.

EXASCALE COMPUTING PROJECT

ClangJIT - A JIT for C++

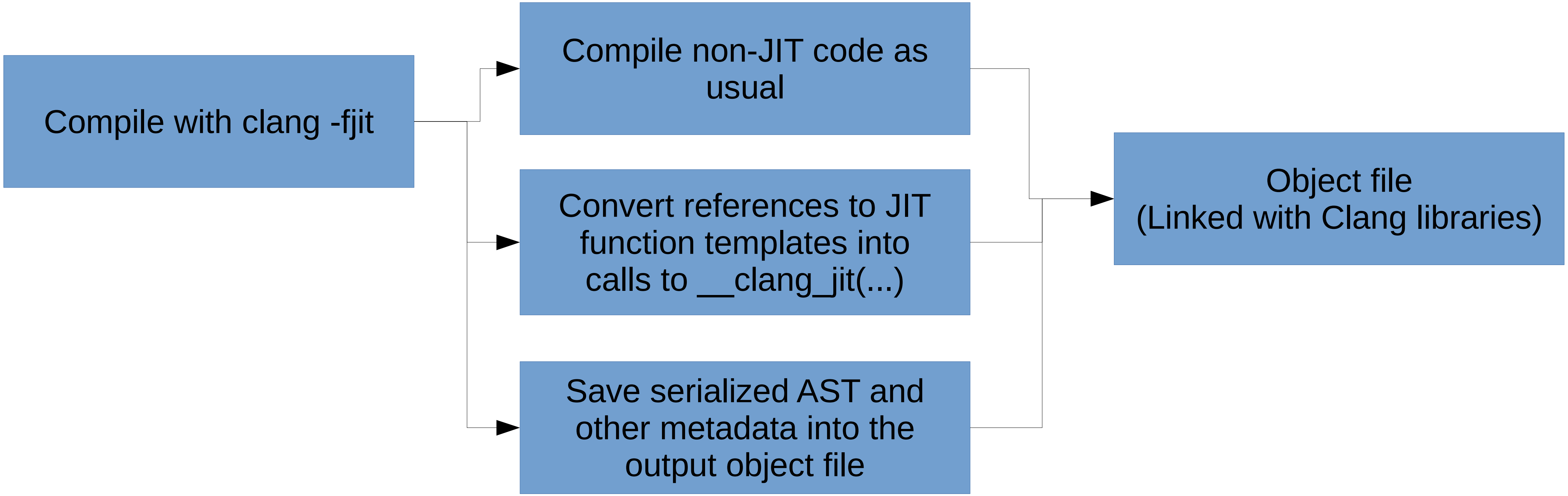
If you'd like to learn more about the potential impact on C++ itself and future design directions, see the talk I gave at CppCon 2019: https://www.youtube.com/watch?v=6dv9vdGIaWs



And the committee proposal: http://wg21.link/p1609

ClangJIT - A JIT for C++

What happens when you compile code with -fjit...



EXASCALE COMPUTING PROJECT

ClangJIT - A JIT for C++

A JIT-enabled ”fat” object file

Serialized AST

- Preprocessor state and compressed source files
- Binary encoding of the AST

Compilation Command-Line Arguments

- Used to restore code-generation options.

Optimized LLVM IR

- Used to allow inlining of pre-compiled functions into JIT-compiled functions.

Local Symbol Addresses

- Used to allow the JIT to look up non-exported symbols in the translation unit.

Serialized AST for Device (First Architecture)

Compilation Command-Line Arguments (First Architecture)

Compilation Optimized LLVM IR (First Architecture)

Serialized AST for Device (Second Architecture)

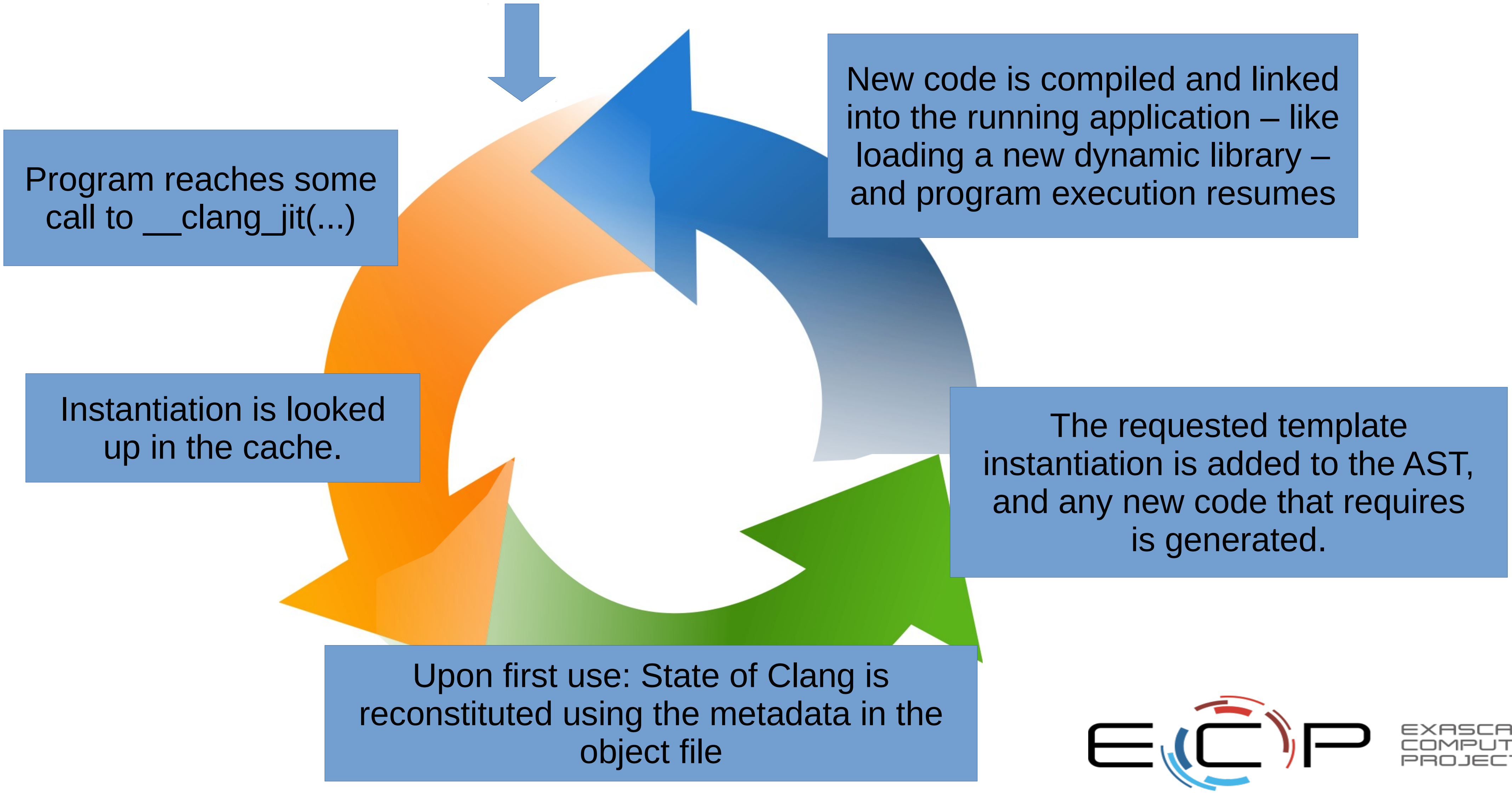
Compilation Command-Line Arguments (Second Architecture)

Compilation Optimized LLVM IR (Second Architecture)

All Targets ◁ ▷ **CUDA Support**

ClangJIT - A JIT for C++

What happens when you run code compiled with -fjit...



ClangJIT - A JIT for C++

```
template <int x>
[[clang::jit]] void foo() { }

void bar(int a) {
  foo<a>();
  foo<a+1>();
}
```

```
|-FunctionTemplateDecl 0x162fdc8 </tmp/f.cpp:1:1, line:2:29> col:21 foo
| |-NonTypeTemplateParmDecl 0x162fc68 <line:1:11, col:15> col:15 'int' depth 0 index 0 x
| |-FunctionDecl 0x162fd28 <line:2:16, col:29> col:21 foo 'void ()'
| | |-CompoundStmt 0x162fea8 <col:27, col:29>
| | `-JITFuncAttr 0x162fe20 <col:3, col:10>
| |-FunctionDecl 0x16301b8 <col:16, col:29> col:21 used foo 'void ()'
| | |-TemplateArgument expr
| | | `-DeclRefExpr 0x1630040 <line:5:7> 'int' lvalue ParmVar 0x162fed0 'a' 'int'
| | |-JITFuncAttr 0x16302b0 <line:2:3>
| | `-JITFuncInstantiationAttr 0x16302f0 <<invalid sloc>> Implicit 0
| `-FunctionDecl 0x1630520 <col:16, col:29> col:21 used foo 'void ()'
|   |-TemplateArgument expr
|   | `-BinaryOperator 0x1630428 <line:6:7, col:9> 'int' '+'
|   |   |-ImplicitCastExpr 0x1630410 <col:7> 'int' <LValueToRValue>
|   |   | `-DeclRefExpr 0x16303d0 <col:7> 'int' lvalue ParmVar 0x162fed0 'a' 'int'
|   |   `-IntegerLiteral 0x16303f0 <col:9> 'int' 1
|   |-JITFuncAttr 0x1630618 <line:2:3>
|   `-JITFuncInstantiationAttr 0x1630658 <<invalid sloc>> Implicit 1
`-FunctionDecl 0x162ff98 <line:4:1, line:7:1> line:4:6 bar 'void (int)'
  |-ParmVarDecl 0x162fed0 <col:10, col:14> col:14 used a 'int'
  `-CompoundStmt 0x1630708 <col:17, line:7:1>
    |-CallExpr 0x16303b0 <line:5:3, col:10> 'void'
    | `-ImplicitCastExpr 0x1630398 <col:3, col:8> 'void (*)()' <FunctionToPointerDecay>
    |   `-DeclRefExpr 0x1630300 <col:3, col:8> 'void ()' lvalue Function 0x16301b8 'foo' 'void ()' (FunctionTemplate 0x162fdc8 'foo')
    `-CallExpr 0x16306e8 <line:6:3, col:12> 'void'
      `-ImplicitCastExpr 0x16306d0 <col:3, col:10> 'void (*)()' <FunctionToPointerDecay>
        `-DeclRefExpr 0x1630668 <col:3, col:10> 'void ()' lvalue Function 0x1630520 'foo' 'void ()' (FunctionTemplate 0x162fdc8 'foo')
```

The template body is skipped during at instantiation.

Each instantiation gets a unique number – used to match __clang_jit calls to an AST location.

EXASCALE COMPUTING PROJECT

ClangJIT - A JIT for C++

Create template arguments, call Sema:SubstDecl and Sema::InstantiateFunctionDefinition. Then call CodeGenModule::getMangledName.

Iterate until convergence:
- Emit all deferred definitions
- Iterate over all definitions in the IR module, for those not available, call GetDeclForMangledName and then HandleInterestingDecl.

Call HandleTranslationUnit

Mark essentially all symbols with ExternalLinkage (no Comdat), renaming as necessary. Link in the previously-compiled IR.

Compile and add module to the process using the JIT.

Add new IR to the previously-compiled IR, marking all definitions as AvailableExternally

ClangJIT - A JIT for C++

```
void bar() { }

template <int i>
[[clang::jit]] void foo() { bar(); }

…

foo<1>();
foo<2>();
```

Initial running module:

```
define available_externally void @_Z3barv() {
  ret void
}
```

ClangJIT - A JIT for C++

```
void bar() { }

template <int i>
[[clang::jit]] void foo() { bar(); }

…

foo<1>();
foo<2>();
```

Running module:

```
define available_externally void @_Z3barv() {
  ret void
}
```

Link

New module:

```
define void @_Z3fooILi1EEvv() {
  call void @_Z3barv()
  ret void
}
```

ClangJIT - A JIT for C++

```
void bar() { }

template <int i>
[[clang::jit]] void foo() { bar(); }

...

foo<1>();
foo<2>();
```

Running module:

```
define available_externally void @_Z3barv() {
  ret void
}

define available_externally void @_Z3fooILi1EEvv() {
  call void @_Z3barv()
  ret void
}
```

ClangJIT - A JIT for C++

```
void bar() { }

template <int i>
[[clang::jit]] void foo() { bar(); }

…

foo<1>();
foo<2>();
```

Running module:

```
define available_externally void @_Z3barv() {
  ret void
}

define available_externally void @_Z3fooILi1EEvv() {
  call void @_Z3barv()
  ret void
}
```

Link

New module:

```
define void @_Z3fooILi2EEvv() {
  call void @_Z3barv()
  ret void
}
```

An Eigen Microbenchmark

```
#include <Eigen/Core>

using namespace std;
using namespace Eigen;

template <typename T>
void test_aot(int size, int repeat) {
  Matrix<T,Dynamic,Dynamic> I = Matrix<T,Dynamic,Dynamic>::Ones(size, size);
  Matrix<T,Dynamic,Dynamic> m(size, size);
  for(int i = 0; i < size; i++)
  for(int j = 0; j < size; j++) {
    m(i,j) = (i+size*j);
  }

  for (int r = 0; r < repeat; ++r) {
    m = Matrix<T,Dynamic,Dynamic>::Ones(size, size) + T(0.00005) * (m + (m*m));
  }
}

void test_aot(const std::string &type, int size, int repeat) {
  if (type == "float")
    test_aot<float>(size, repeat);
  else if (type == "double")
    test_aot<double>(size, repeat);
  else if (type == "long double")
    test_aot<long double>(size, repeat);
  else
    cout << type << "not supported for AoT\n";
}
```

Let's think about a simple benchmark…

- Iterate, for a matrix m: $m_{n+1} = I + 0.00005 * (m_n + m_n*m_n)$
- Here, a version traditionally supporting a runtime matrix size:

An Eigen Microbenchmark

Here, a version using JIT to support a runtime matrix size via runtime specialization:

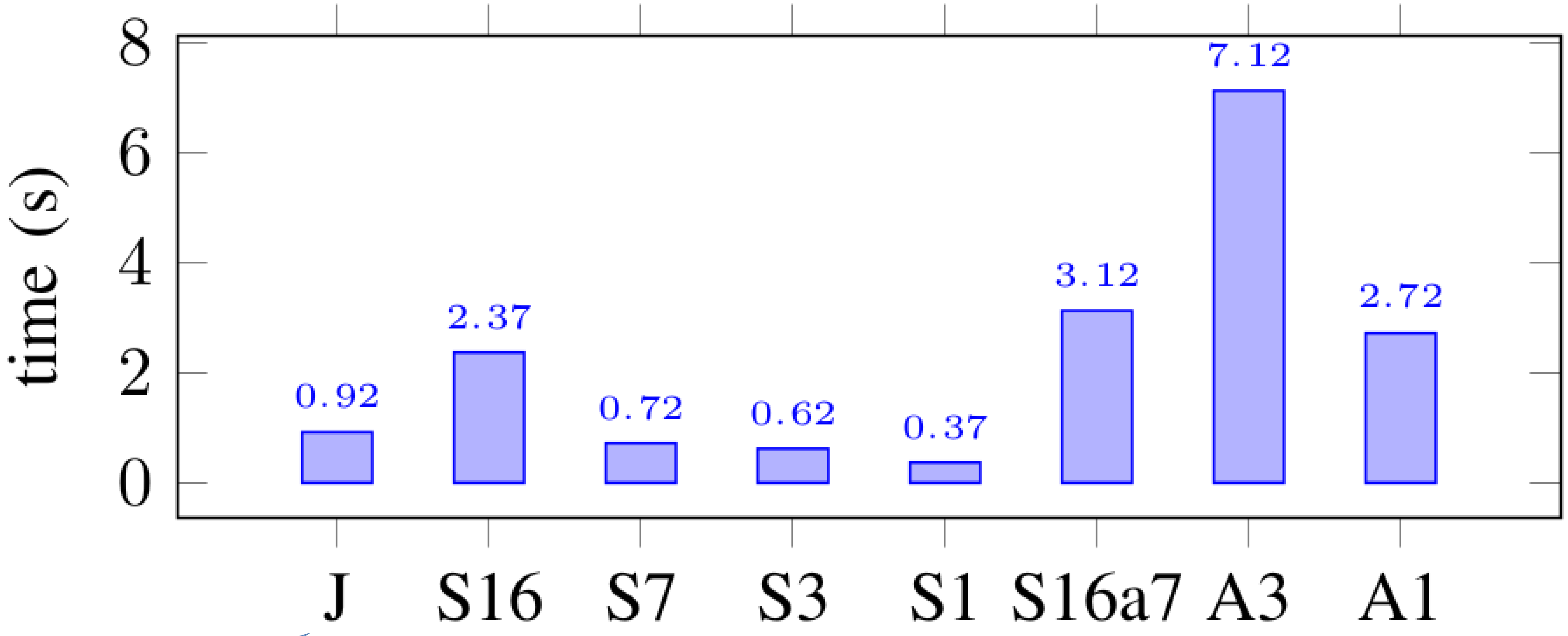
```
template <typename T, int size>
[[clang::jit]] void test_jit_sz(int repeat) {
  Matrix<T, size, size> I = Matrix<T, size, size>::Ones();
  Matrix<T, size, size> m;
  for(int i = 0; i < size; i++)
  for(int j = 0; j < size; j++) {
    m(i,j) = (i+size*j);
  }

  for (int r = 0; r < repeat; ++r) {
    m = Matrix<T, size, size>::Ones() + T(0.00005) * (m + (m*m));
  }
}

void test_jit(const std::string &type, int size, int repeat) {
  return test_jit_sz<type, size>(repeat);
}
```

An Eigen Microbenchmark

First, let’s consider (AoT) compile time (time over baseline):



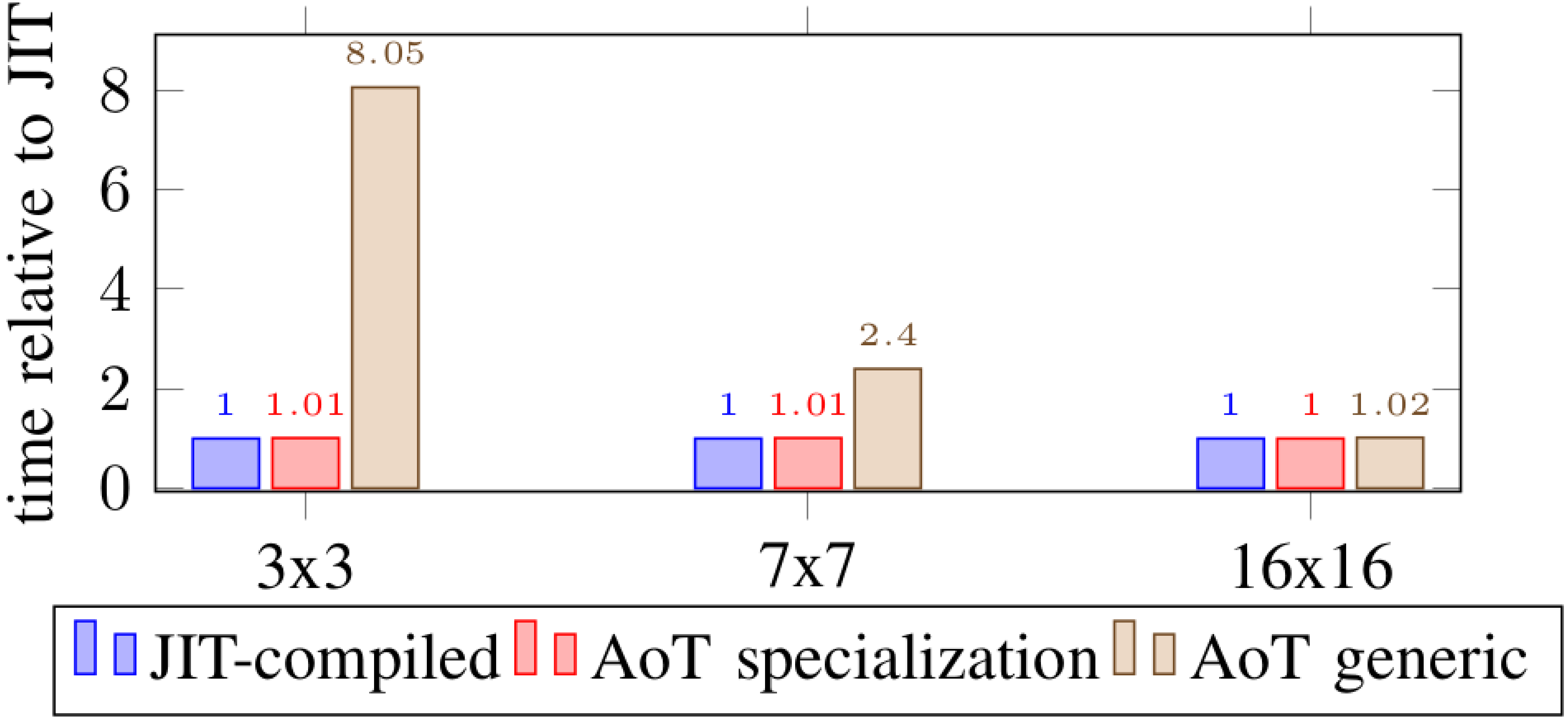
The JIT version.

Time to compile a version with one specific (AoT) specialization

The AoT version (with one or all three float types)

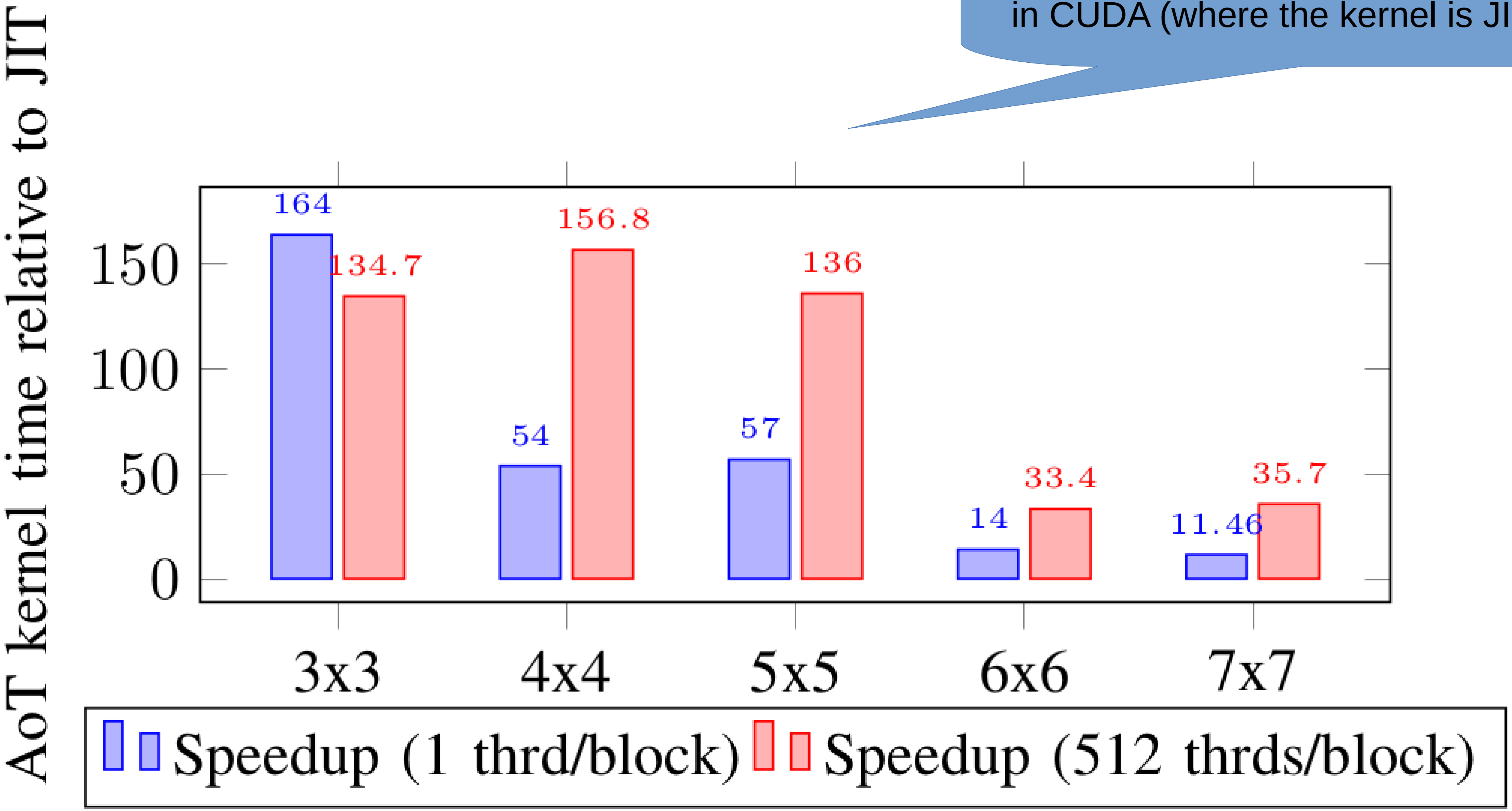
An Eigen Microbenchmark

Now, let's look at runtime performance (neglecting runtime-compilation overhead):



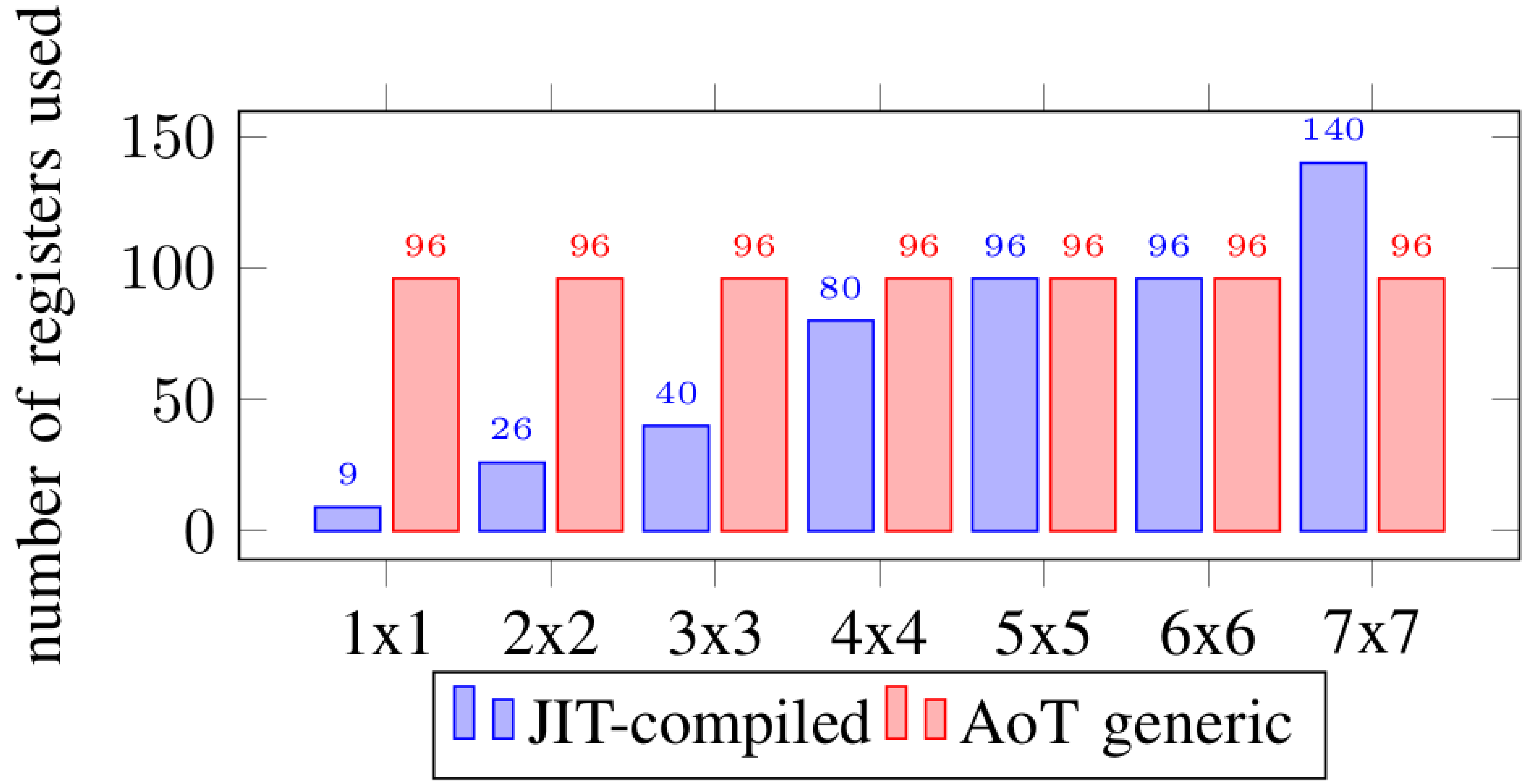
An Eigen Microbenchmark

Essentially the same benchmark, but this time in CUDA (where the kernel is JIT specialized)



An Eigen Microbenchmark

For CUDA, one important aspect of specialization is the reduction of register pressure:



Can This Fix All C++ Compile-Time Issues?

I use C++. I can start testing my code just minutes after writing it...

I use programming language X.
I can start testing my code as soon as I can press “enter.”

[[clang::jit]] will not, by itself, solve all C++ compile-time problems, however the underlying facility can be used directly to solve some problems, such as...

Can This Fix All C++ Compile-Time Issues?

This kind of manual-dispatch code
is very expensive to compile.
Using [[clang::jit]] can get rid of this in a
straightforward way, providing
a faster and more-flexible solution.

Listing 13: The manual explicit-instantiati ... d-dispatch code in Laghos's rMassMultAdd.cpp that Clang ... makes obsolete.

```
const unsigned int id = (DIM<<16)|((NUM_DOFS_1D-1)<<8)|(NUM_QUAD_1D>>1);
static std::unordered_map<unsigned int, fMassMultAdd> call =
{
  {0x20001,&rMassMultAdd2D<1,2>},     {0x20101,&rMassMultAdd2D<2,2>},
  {0x20102,&rMassMultAdd2D<2,4>},     {0x20202,&rMassMultAdd2D<3,4>},
  {0x20203,&rMassMultAdd2D<3,6>},     {0x20303,&rMassMultAdd2D<4,6>},
  {0x20304,&rMassMultAdd2D<4,8>},     {0x20404,&rMassMultAdd2D<5,8>},
  ... // There are approximately 32 lines, in total, like those above.
};

assert(call[id]); // This solution is not as flexible as using ClangJIT.
call[id](numElements,dofToQuad,dofToQuadD,quadToDof,quadToDofD,op,x,y);
```

Can This Fix All C++ Compile-Time Issues?

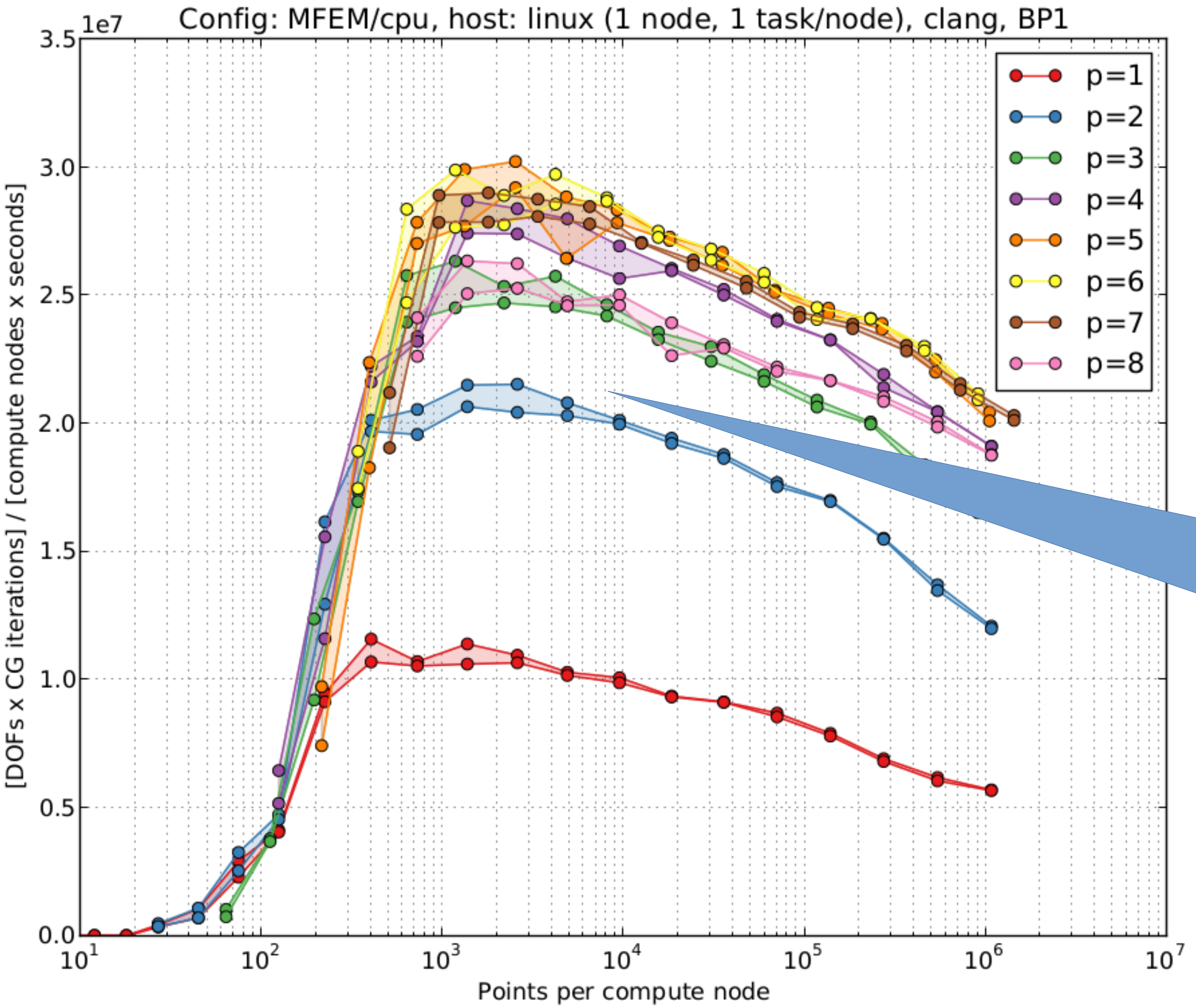
Listing 10: An excerpt of the rMassMultAdd2D template in Laghos.

```
template<const int NUM_DOFS_1D, const int NUM_QUAD_1D>
void rMassMultAdd2D(
  const int numElements,
  const double* restrict dofToQuad,
  const double* restrict dofToQuadD,
  const double* restrict quadToDof,
  const double* restrict quadToDofD,
  const double* restrict oper,
  const double* restrict solIn,
  double* restrict solOut) {
  forall(e,numElements, {
    double sol_xy[NUM_QUAD_1D][NUM_QUAD_1D];
    // sol_xy[*][*] = 0;
    for (int dy = 0; dy < NUM_DOFS_1D; ++dy) {
      double sol_x[NUM_QUAD_1D];
      // sol_x[*] = 0;
      for (int dx = 0; dx < NUM_DOFS_1D; ++dx) {
        const double s = solIn[ijkN(dx,dy,e,NUM_DOFS_1D)];
        for (int qx = 0; qx < NUM_QUAD_1D; ++qx) {
          sol_x[qx] += dofToQuad[ijN(qx,dx,NUM_QUAD_1D)]* s;
        }
      }
      for (int qy = 0; qy < NUM_QUAD_1D; ++qy) {
        const double d2q = dofToQuad[ijN(qy,dy,NUM_QUAD_1D)];
        for (int qx = 0; qx < NUM_QUAD_1D; ++qx) {
          sol_xy[qy][qx] += d2q * sol_x[qx];
        }
      }
    }
    // a second loop nest similar to that above...
  }); }
```

(In case you're curious what that kernel looks like...)

Can This Fix All C++ Compile-Time Issues?

We integrated this into a large application, and benchmarked it for different polynomial-order choices…



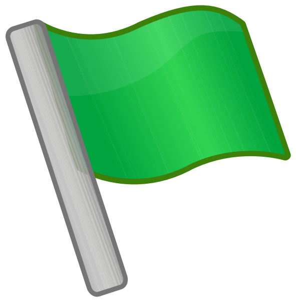
For each polynomial order, the JIT version was slightly faster (likely because ClangJIT's cache lookup, based on DenseMap, is faster than the lookup in the original implementation)

EXASCALE COMPUTING PROJECT

Some Notes on Costs

In the ClangJIT prototype, on an Intel Haswell processor, for the simplest lookup involving a single template argument (all numbers approximate):

Cache lookup (already compiled)	350 cycles (140 ns)
Resolving the instantiation request to the previously-compiled (same type with different spelling)	160 thousand cycles (65 μs)
Compiling new instantiations	At the very least, tens of millions of cycles (a few milliseconds)



Some Other Concerns

- Because the instantiation of some templates can affect the instantiation of other templates (e.g., because friend injection can affect later overload resolution), as currently proposed, the implementation of the JIT-compilation functionality cannot be "stateless." This seems likely to make it harder to automatically discard unneeded specializations.
- ABI: If an application is compiled with all of the necessary metadata embedded within it to compile the JIT-attributed templates, does that metadata format, and the underlying interface to the JIT-compilation engine that uses it, become part of the ABI that the system must support in order to run the application? The answer to this question seems likely to be yes, although maybe this just provides another failure mode…
- JIT compilation can fail because, in addition to compiler bugs, the compilation engine might lack some necessary resources, or the code might otherwise trigger some implementation limit. In addition, compilation might fail because an invalid type was provided or the provided type or value triggered some semantic error (including triggering a static_assert).
- How does this interact with code signing? Can we have a fallback interpreter for cases/environments where JIT is not possible?
- C++ serialized ASTs can be large, and C++ compilation can consume a lot of memory (in addition to being slow).

Where Might We Go From Here?

hfinkel / llvm-project-cxxjit

<> Code | Issues 6 | Pull requests 1 | Projects 0 | Wiki

Home

Hal Finkel edited this page now · 20 revisions

Welcome to the llvm-project-cxxjit wiki! This is a fork of LLVM with a Clan
time (JIT) compilation functionality.

For more information on the implementation and some evaluation, see:
https://arxiv.org/abs/1904.08555

Getting Started

To install ClangJIT, clone https://github.com/hfinkel/llvm-project-cxxjit and
Clang/LLVM for your system.

If you're building on a Linux system, then a basic CMake configuration su
likely work:

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DBUILD_SHARED_LIBS=ON -DLLVM_USE_SPLIT_DWARF
```

and then run `make` (using `-j<number of cores` to get a parallel build) and then `make install`. Before you install, you might wish to run LLVM's and Clang's regression tests (which include regression tests associated with the JIT functionality). `make -j<number of cores> check-llvm check-clang` should do that. Once installed, you can then use the `/path/to/somewhere/bin/clang++` with the `-fjit` option as described below.

ROOT
Data Analysis Framework

Download | Documentation | News | Support | About | Development | Contribute

Home

Cling

What is Cling

Cling is an interactive C++ interpreter, built on the top of LLVM and Clan
interpreters are that it has command line prompt and uses just-in-time (JIT
developers (e.g. Mono in their project called CSharpRepl) of such kind
compilers.

One of Cling's main goals is to provide contemporary, high-performance a
ROOT project - CINT. The backward-compatibility with CINT is major p

The LLDB Debugger

Welcome to the LLDB version 8 documentation!

LLDB is a next generation, high-performance debugger. It is built as a set of reusable components which highl

LLDB is the default debugger in Xcode on macOS and supports debugging C, Objective-C and C++ on the de

All of the code in the LLDB project is available under the "Apache 2.0 License with LLVM exceptions".

Why a New Debugger?

In order to achieve our goals we decided to start with a fresh architecture that would support modern multi-thre
in support for functionality and extensions. Additionally we want the debugger capabilities to be available to o

Compiler Integration Benefits

LLDB currently converts debug information into clang types so that it can leverage the clang compiler infrastr
runtimes in expressions without having to reimplement any of this functionality. It also leverages the compiler
extracting instruction details, and much more.

The major benefits include:

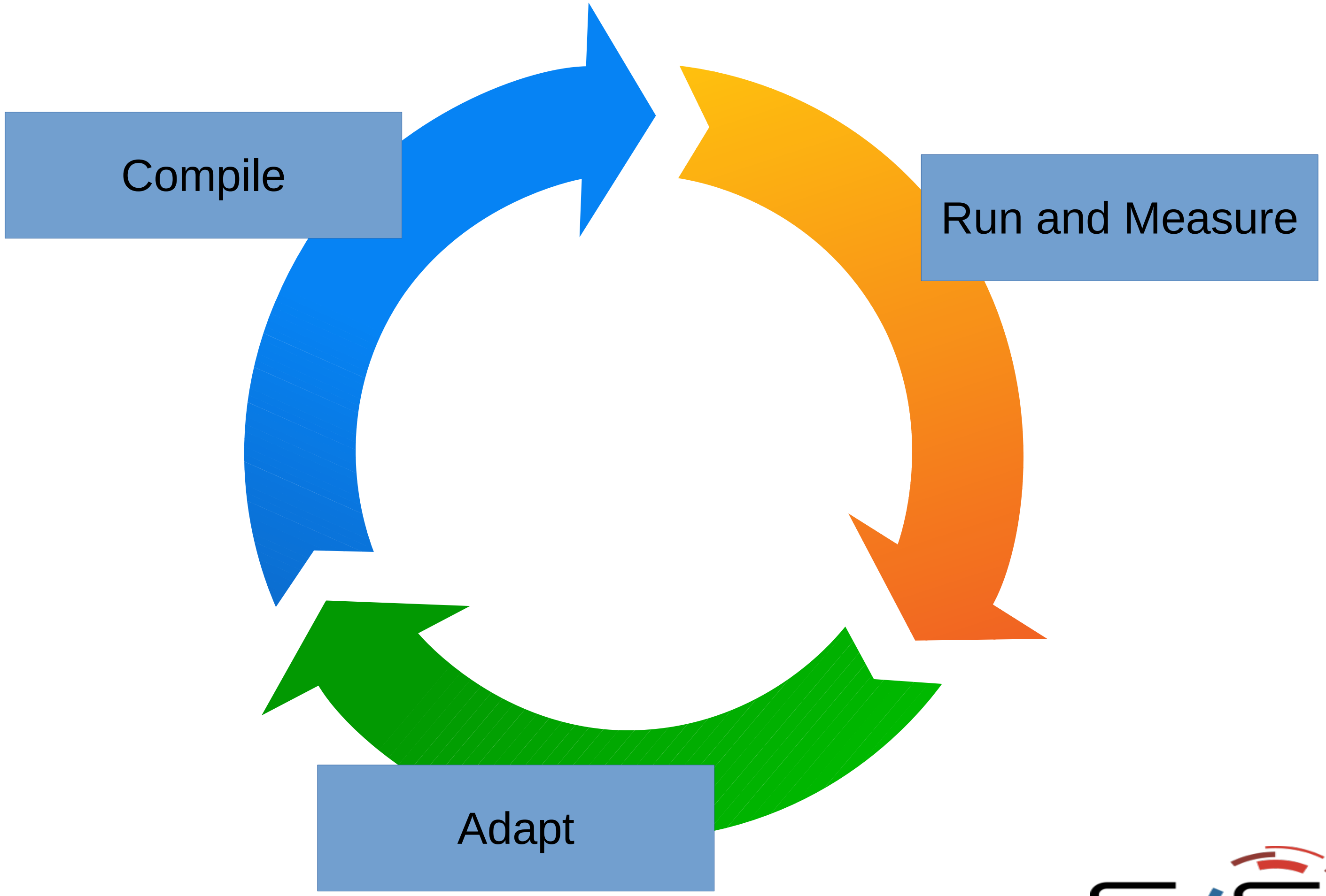
- Up to date language support for C, C++, Objective-C
- Multi-line expressions that can declare local variables and types
- Utilize the JIT for expressions when supported
- Evaluate expression Intermediate Representation (IR) when JIT can't be used

A common infrastructure for C++ JIT compilation?
(A roundtable today @ noon)

EXASCALE COMPUTING PROJECT

What To Build On Top? - Autotuning

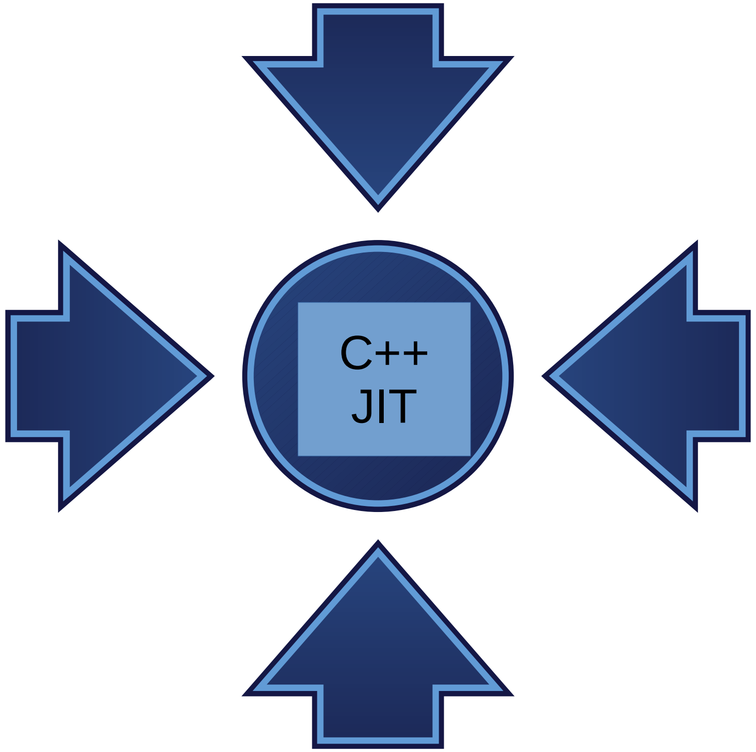
Adapting to hardware, especially heterogeneous hardware, with high-performance specializations may require autotuning:



Conclusion

Hardware Trends + Performance Requirements
(Need for efficiency, heterogeneity, and more)

Modern JIT-compilation technology



Evolution of C++
(e.g., constexpr programming)

Needs for increased programmer productivity

Acknowledgments

- David Poliakoff (SNL), Jean-Sylvain Camier (LLNL) and David F. Richards (LLNL), my co-authors on the associated academic work
- The many members of the C++ standards committee who provided feedback
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357
- Work at Lawrence Livermore National Laboratory was performed under contract DE-AC52-07NA27344

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.