Two Insane Compiler Tricks That Will Blow Your Mind

How to Get Better Performance *and* Higher Productivity

David Richards & David Poliakoff

February 28, 2019

LLNL-PRES-758109

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Acknowledgements

- Tom Scogland, Jean-Sylvain Camier (LLNL JIT)
- Ramesh Pankajakshan, Bjorn Sjogreen (LLNL SW4)
- Rob Rieben, Tzanio Kolev (LLNL Blast/Laghos)
- Peter Robinson & the ALE3D Team
- Brian Ryujin & the Ares Team
- Adam Kunen & the Ardra Team
- Si Hammond, Christian Trott (Sandia Kokkos)

Acknowledgements

- Tom Scogland, Jean-Sylvain Camier (LLNL JIT)
- Ramesh Pankajakshan, Bjorn Sjogreen (LLNL SW4)
- Rob Rieben, Tzanio Kolev (LLNL Blast/Laghos)
- Peter Robinson & the ALE3D Team
- Brian Ryujin & the Ares Team
- Adam Kunen & the Ardra Team
- Si Hammond, Christian Trott (Sandia Kokkos)

The best tool for finding errors in a million line code is not a developer

- Compilers can easily spot bugs which are opaque to users
  - "=" and "==" look the same to humans, but are unrelated to a compiler
- Many standard static analysis tools are available
  - Klockwork, Coverity, Clang-tidy, Fortify, cppcheck, Lint, etc.
- Standard tools help if we are writing bad code for the **language**
- What if I'm writing good C++ but bad Kokkos?
  - What if I'm writing good RAJA but bad Ardra?
  - What if what was good Ardra yesterday isn't good Ardra today?

Static analysis can be customized to the style and idiom of a specific code to find errors quickly and make developers more productive

Customized static analysis easily finds idiom-specific performance problems

- To make codes play nicely with UM, access only the innermost portions of data structures in kernels. Valid C++. Bad RAJA

```
RAJA::forall<RAJA::seq_exec>(0,10,[=](int i){
    my_field[i] = AllPhysics->Hydrodynamics->Temperature->Data[i];
});
```

Traversing the data hierarchy in the GPU section can cause large data transfers to GPU

A simple alias avoids the problem

```
auto data = AllPhysics->Hydrodynamics->Temperature->Data;
RAJA::forall<RAJA::seq_exec>(0,10,[=](int i){
    my_field[i] = data[i];
});
```

This issue is easy to spot in three lines of code, but very difficult to enforce in three million over thirty years of development

Clang query provides a flexible, maintainable method to create customized static analysis

- Clang query is a scripting language which describes patterns in an AST, for which Clang will then report matches
- Pro:
  - Tested and updated with clang API changes
  - Uses scripts instead of shared libraries.
    - Less vulnerable to API changes.
    - Easier to distribute
- Con:
  - Some expressive power is lost
  - Can't "unpack" a lambda and see inside
  - But you can frequently find work-arounds

Clang query script code to find access through hierarchy of structures

```
let foralls callExpr(
  callee(
    functionDecl(matchesName("for.*all"))
  )
)
```

```
match memberExpr(
  hasAncestor(
    lambdaExpr(
      hasAncestor(foralls)
    )
  )
)
```

This looks complicated, but experience has shown that with a few examples and modest training, developers can start to write their own clang query scripts

Clang query can find many performance bugs and other anti-patterns

- RAJA Kernels
  - Uses of outer structs
  - Uses of indirection arrays
  - Kernels with no Reducers not taking advantage of reducer-free policies
  - CHAI ManagedArrays used outside of Kernels
    - Uses of raw arrays inside Kernels
- Kokkos Kernels
  - Uses of STL classes
  - Non-const index arguments in lambdas
  - Nested parallelism errors
    - Kokkos::single types in inappropriate enclosing construct
    - Writes to variables in an outer scope from inside a parallel_for

Clang query can find many performance bugs and other anti-patterns

- RAJA Kernels
  - Uses of outer structs
  - Uses of indirection arrays
  - Kernels with no Reducers not taking advantage of reducer-free policies
  - CHAI ManagedArrays used outside of Kernels
    - Uses of raw arrays inside Kernels
- Kokkos Kernels
  - Uses of STL classes
  - Non-const index arguments in lambdas
  - Nested parallelism errors
    - Kokkos::single types in inappropriate enclosing construct
    - Writes to variables in an outer scope from inside a parallel_for

Application programmers love clang query

- Brian Ryujin, Ares: "*[The] tool greatly simplified this process to the point that we could split the code up between the team and finish the entire code in 3 days. This would have been impossible to do without the tool. I would estimate that it saved over 100 hours of effort and a fair amount of sanity. I think it goes without saying that we were very happy with the tool.*"

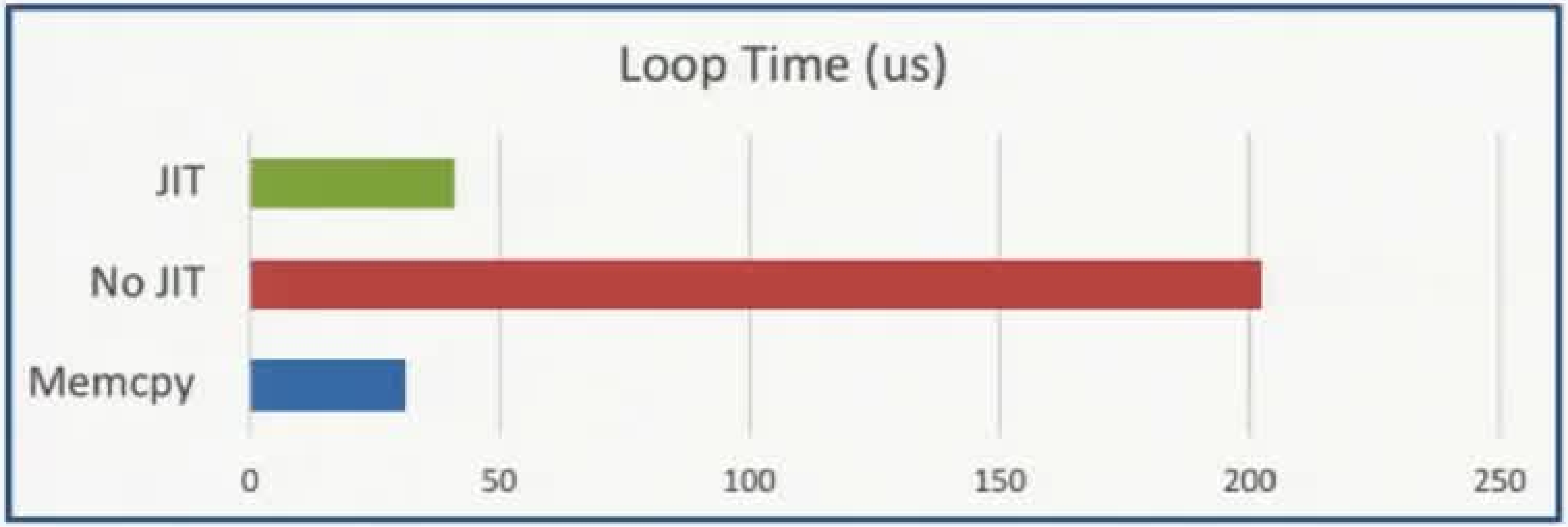
- Adam Kunen, Ardra: "*I am really interested in using this tool, as it will help us discover porting mistakes as we continue to transition our code to RAJA+CHAI. I am particularly excited at how easily it integrated into an existing CMake build system, and how powerful of a tool it is. This tool is really high-impact and low-cost. We are not currently using it, but over the next 6 months intend to collaborate with David more extensively on this tool.*"

- Tzanio Kolev, MFEM, CEED: "*I want to add this to my code!*"

Static analysis based on clang query is on its way to production!!!

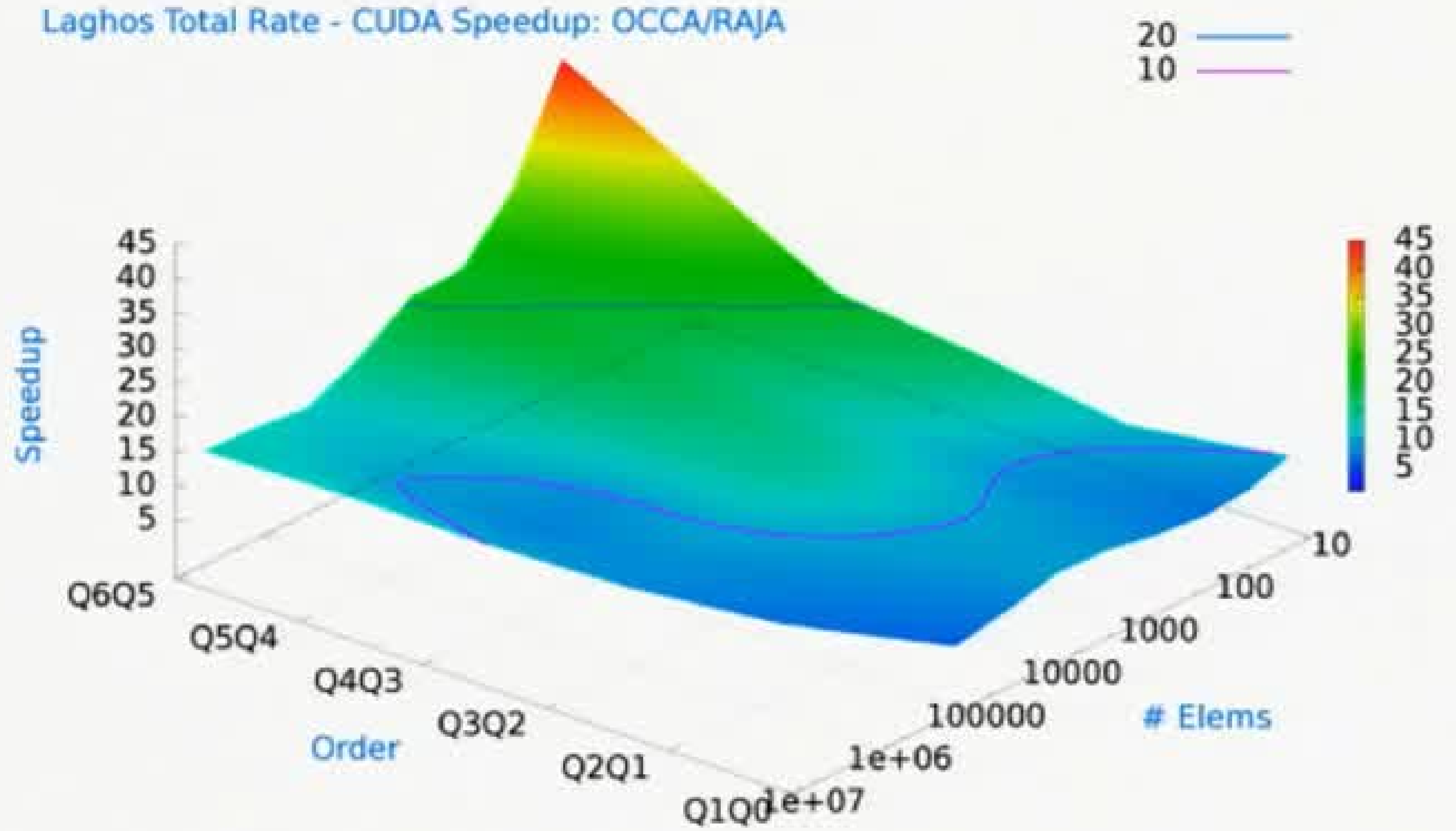
Early experience with GPUs and Cardioid revealed a need for JIT compilation

- Cardioid relies heavily on polynomial evaluation
- Performance is *greatly* improved when polynomial orders and coefficients (model parameters) are known at compile time
- Scientists prefer to define model parameters at run time



JIT compilation is the only practical way to get High performance and the preferred usage model

OCCA implementation of Laghos set the GPU performance bar very high!



OCCA performance advantage derives from
- hand tuned kernels
- shared memory utilization
- JIT compilation

RAJA needs a JIT capability to match OCCA performance

Loops in high-order finite element codes are excellent targets for run-time optimization

- Laghos & Blast: Loop bounds defined by input deck

```
for(int el =0; el<numElements;el++){
  double e_xy[NUM_QUAD_1D*NUM_QUAD_1D];
  for (int dx = 0; dx < NUM_DOFS_1D; ++dx) {
    const double r_e = e[ijkN(dx,dy,el,NUM_DOFS_1D)];
    for (int qx = 0; qx < NUM_QUAD_1D; ++qx) {
      myField += L2DofToQuad[ijN(qx,dx,NUM_QUAD_1D)] * r_e;
      /** More tensor math */
    }
  }
};
```

- NUM_QUAD_1D and NUM_DOFS_1D are in the (4,32) range

We created a prototype JIT compiler for RAJA to explore possible optimizations

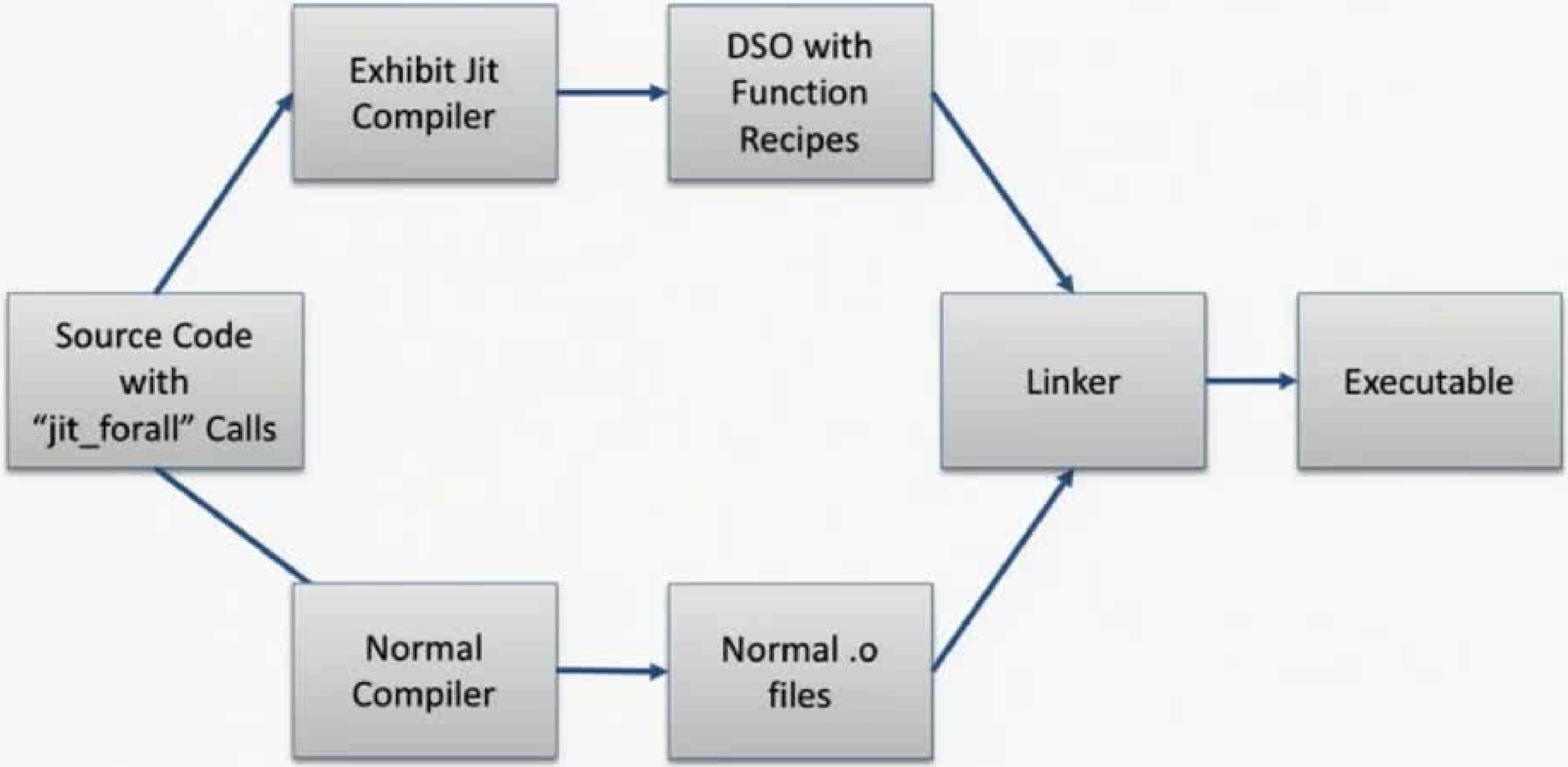
Changed from RAJA::forall<RAJA::seq_exec>

```
jit_forall_cpu(0,numElements,[&](int el) {
  double e_xy[NUM_QUAD_1D*NUM_QUAD_1D];
  for (int dx = 0; dx < NUM_DOFS_1D; ++dx) {
    const double r_e = e[ijkN(dx,dy,el,NUM_DOFS_1D)];
    for (int qx = 0; qx < NUM_QUAD_1D; ++qx) {
      myField += L2DofToQuad[ijN(qx,dx,NUM_QUAD_1D)] * r_e;
      /** More tensor math */
    }
  }
},
parameters(myField,L2DofToQuad, /** ... */),
replace_scalar(NUM_QUAD_1D),
replace_scalar(NUM_DOFS_1D)
);
```

Tells JIT compiler what optimizations to use

JIT-able functions are practically unchanged from original RAJA version

Two compilers are required to produce a JIT enabled binary



JIT run-time overhead is nearly zero

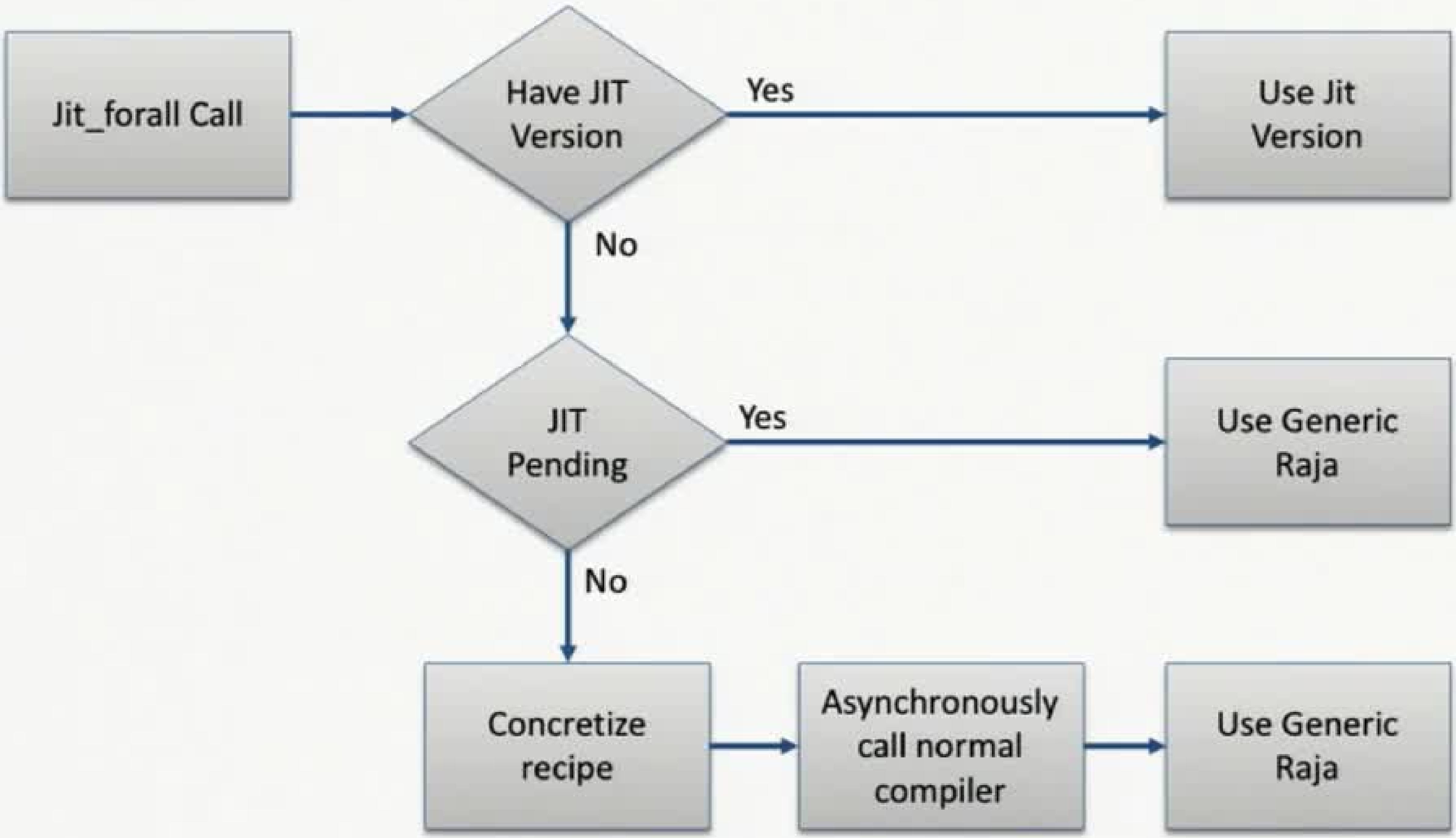


Exhibit compiler creates a function recipe

```
char* compile_assistinitialization(const char** replacements, const char* specialization_arg){
  std::string* program =new std::string(
      R"META(extern "C" void initialization_)META"+
      std::string(specialization_arg)+ // This is how we manage multiple variants of the same recipe
      R"META((int compiler_generated_start_index_name,
       int compiler_generated_end_index_name,float * h_array,int debug_do_not_merge=0){
         for(int i = compiler_generated_start_index_name;i<compiler_generated_end_index_name;++i){)META"+R"INNER(
           h_array[i] = )INNER"+std::string(replacements[0])+R"INNER( * i;
         })INNER"
        "\n}"
  );
  return (char*)program->c_str();
}
```

A concretized version is generated from the recipe

```
extern "C" void initialization_0(
    int compiler_generated_start_index_name,
    int compiler_generated_end_index_name,
    float * h_array,
    int debug_do_not_merge=0
  ){
  for(int i = compiler_generated_start_index_name;
        i<compiler_generated_end_index_name;
        ++i
    ){
    h_array[i] = 8 * i;
    }
}
```

Our first trials with JIT were very encouraging

```
jit_kernel_gpu<2>(code_location, 0, array_size,[=] __device__(int i){
  for(int k = 0; k < scalar; k++){
    d_array[i] += scalar * scalar * scalar;
  }
}, parameters(d_array), make_replacement(scalar));
```

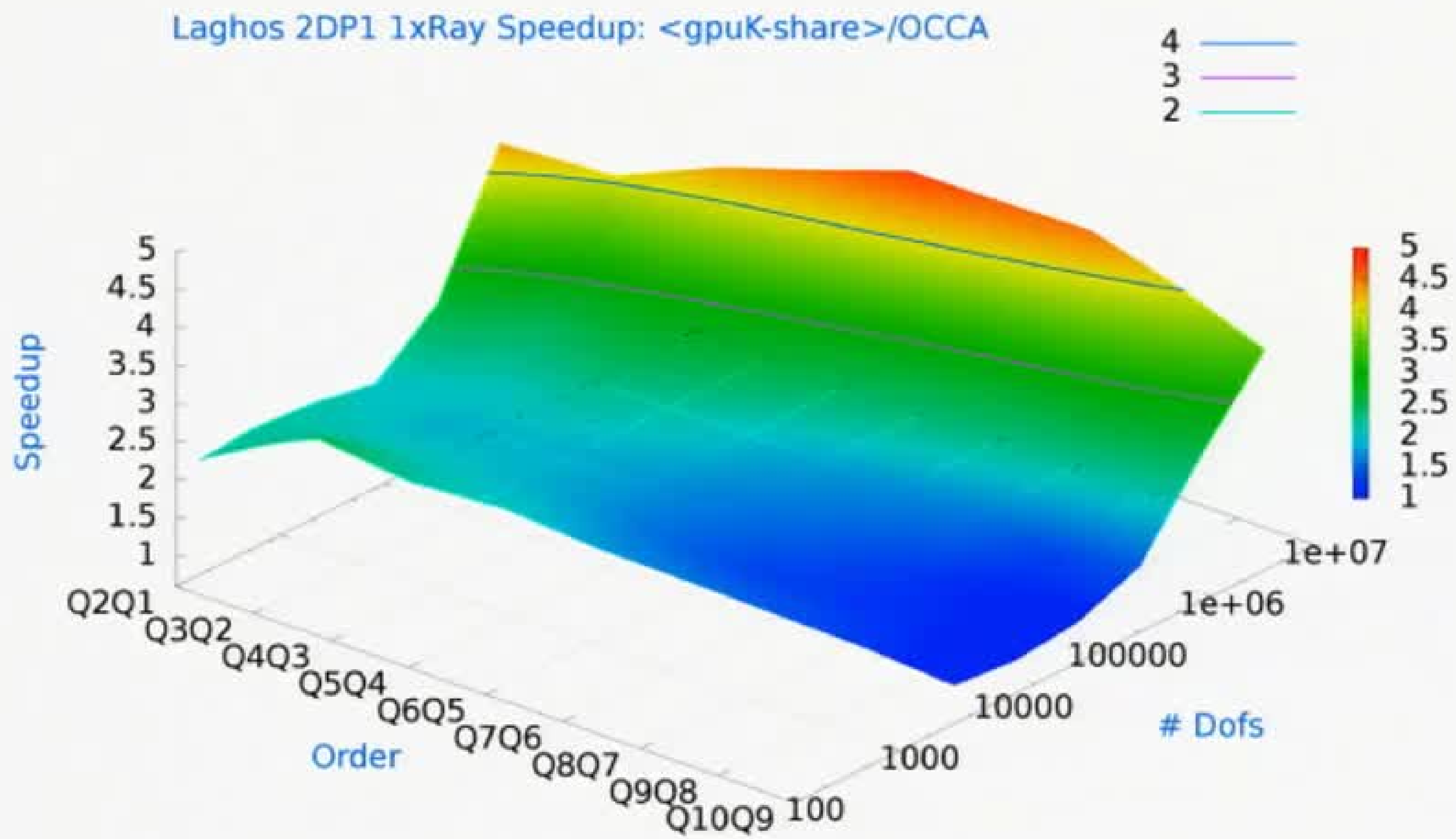
scalar	Without Jit (s)	With Jit (s)
512	131.322	9.888
128	39.403	9.882
2	11.504	9.860

JIT improves performance

- GPU Polynomial evaluation (n=120)
  - No JIT: 175 seconds
  - JIT: 20 seconds
- Laghos on GPU – parity with templates
  - Template: 13.91 seconds
  - JIT: 13.83 seconds
- MFEM on CPU:

ForceMult2D	
(Thermal, Kinematic Orders)	Speedup From Jit
(2,3)	20%
(3,4)	35%

JIT is the final piece of the puzzle to allow RAJA to match OCCA performance



JIT improves performance

- GPU Polynomial evaluation (n=120)
  - No JIT: 175 seconds
  - JIT: 20 seconds
- Laghos on GPU – parity with templates
  - Template: 13.91 seconds
  - JIT: 13.83 seconds
- MFEM on CPU:

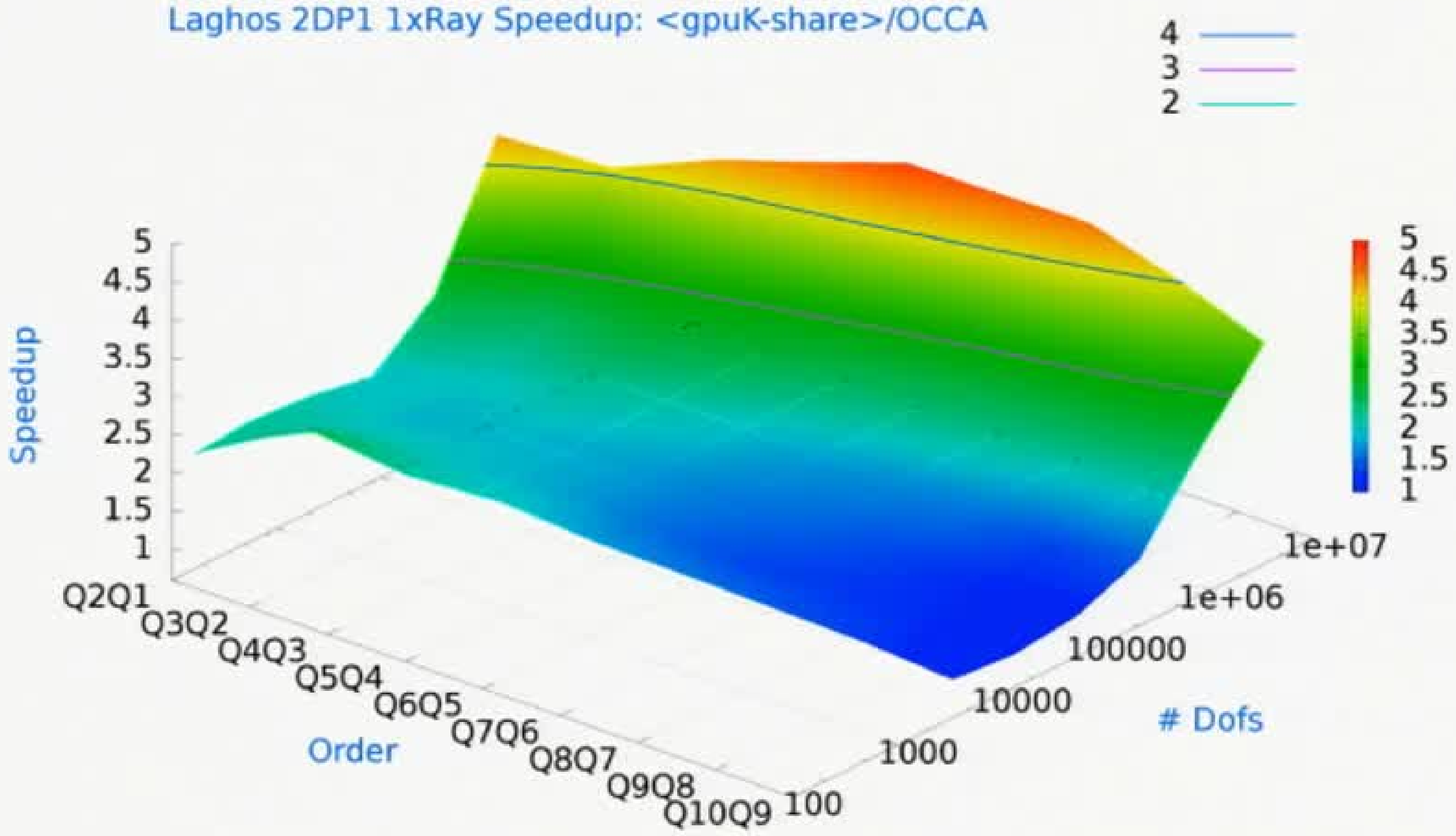
ForceMult2D	
(Thermal, Kinematic Orders)	Speedup From Jit
(2,3)	20%
(3,4)	35%

JIT improves performance

- GPU Polynomial evaluation (n=120)
  - No JIT: 175 seconds
  - JIT: 20 seconds
- Laghos on GPU – parity with templates
  - Template: 13.91 seconds
  - JIT: 13.83 seconds
- MFEM on CPU:

ForceMult2D	
(Thermal, Kinematic Orders)	Speedup From Jit
(2,3)	20%
(3,4)	35%

JIT is the final piece of the puzzle to allow RAJA to match OCCA performance



If templates can provide the necessary specializations, why do we need JIT?

- Template solution involves instantiation of thirty commonly used thermal and kinematic orders
- Binary size (per object file) (approximately 40 object files in MFEM)
  - Without JIT: 209K
  - With JIT: 14K
- Compile time
  - Without Jit: 72 seconds
  - With Jit: 11 seconds

JIT substantially lowers compile time and produces smaller binaries

JIT isn't just for RAJA

- To test interoperability with the other labs, we decided to JIT Kokkos regions and pass them to Kitsune.
- We tested two benchmarks:
  - GUPS
  - Stream
- Neither was predicted to see performance benefits from JIT, and neither did. However, we introduced no overhead.
- We are looking into KokkosKernels for possible JIT optimization candidates

Our prototype RAJA JIT compiler was easily adapted to handle Kokkos

We plan to bring customized static analysis and JIT into production with ASC codes

- Customized static analysis has a demonstrated ability to
  - Find problems in code
  - Improve developer productivity

- Our JIT prototype has shown sufficiently promising results to justify work on a production quality implementation
  - MARBL lead Rob Rieben has asked for a RAJA JIT capability

- Contacts:
  - David Richards (richards12@llnl.gov)
  - David Poliakoff (poliakoff1@llnl.gov)

Lawrence Livermore National Laboratory