Proteus: Programmable JIT compilation for C/C++

Tutorial @ CGO26

Feb. 1st 2026

Zane Fink, Konstantinos Parasyris, Giorgis Georgakoudis
Center for Advanced Scientific Computing (CASC), LLNL

Prepared by LLNL under Contract DE-AC52-07NA27344.

Proteus: programmable C/C++ for JIT compilation and optimization

Mneme: advanced scalable autotuning using Proteus

Olympus-HPC

- We'll show how to install and use **Proteus**, dive into its internals, and highlight compelling performance results 🚀
- We'll introduce **Mneme**, walk through its installation and usage, and explore examples of extreme autotuning in action 🔧⚡
- Questions are welcome throughout: jump in anytime! 💬🙌

The Proteus Project

- https://github.com/Olympus-HPC/proteus
- Project goal

Research and develop programmable JIT compilation and optimization to maximize the performance of HPC codes

easy-to-integrate state-of-the-art

easy-to-use

high-performance scalable

Giorgis Georgakoudis (PI)

David Beckingsale

Konstantinos Parasyris

John Bowen

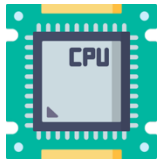
Zane Fink

Tal Ben Nun

Thomas Stitt

We develop Proteus open-source

- Support

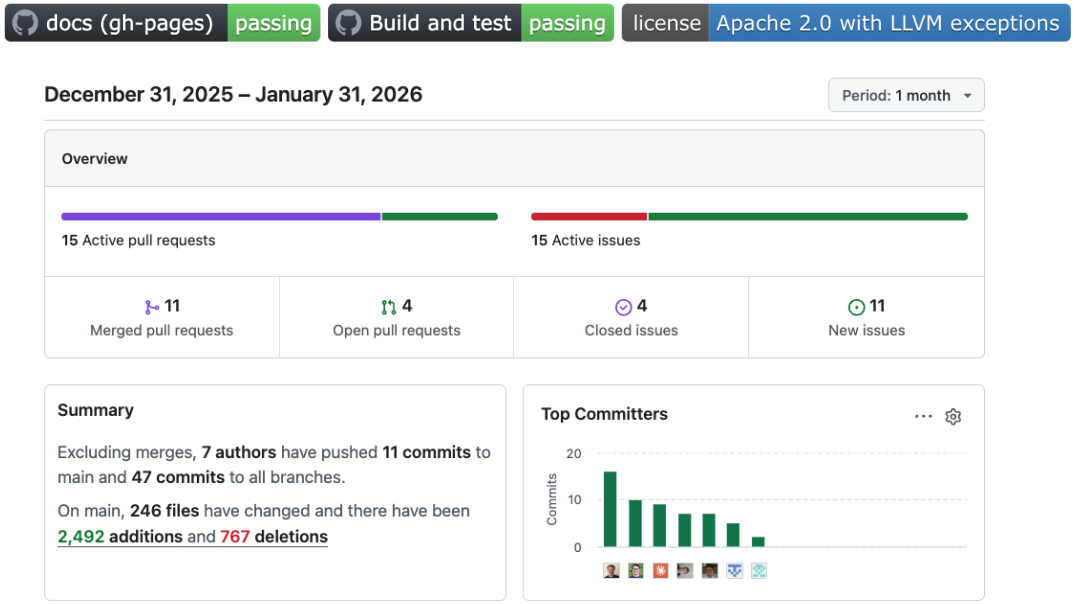


  - RDC/non-RDC compilation
  - Device libraries
    - (both Proteus and non-Proteus compiled)

- Continuous integration testing
  - GitHub and GPU-enabled GitLab CI
  - LLVM 18/19/20
  - CUDA 12.2
  - ROCm 6.2.1, 6.3.1, 6.4.1, 7.1.0

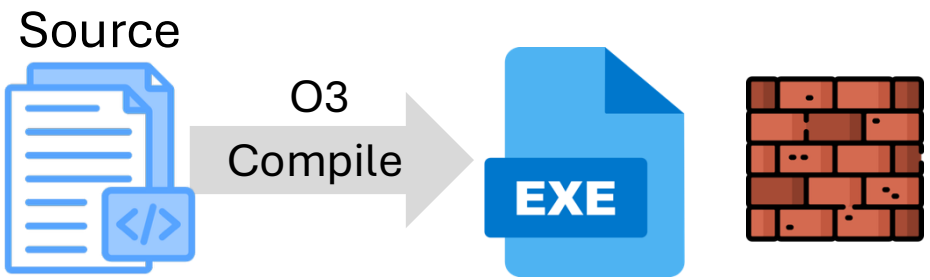
- Documentation
  - https://olympus-hpc.github.io/proteus/

- Actively Developed
  - https://github.com/Olympus-HPC/proteus



What we do now to get high performance from our HPC software has limitations

- Static ahead-of-time (**AOT**) optimizing compilation



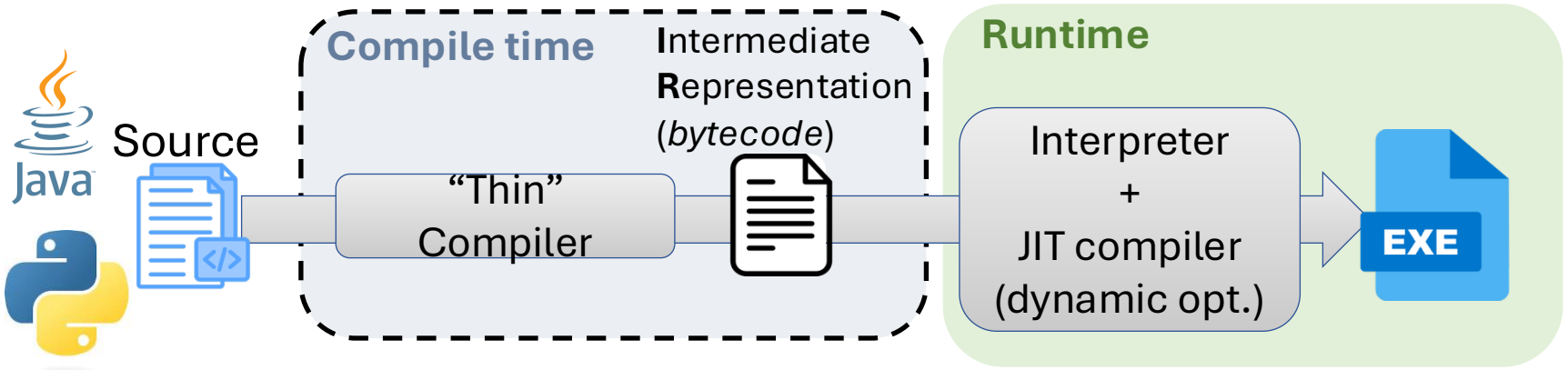
- Write ugly, compile-time value specializations

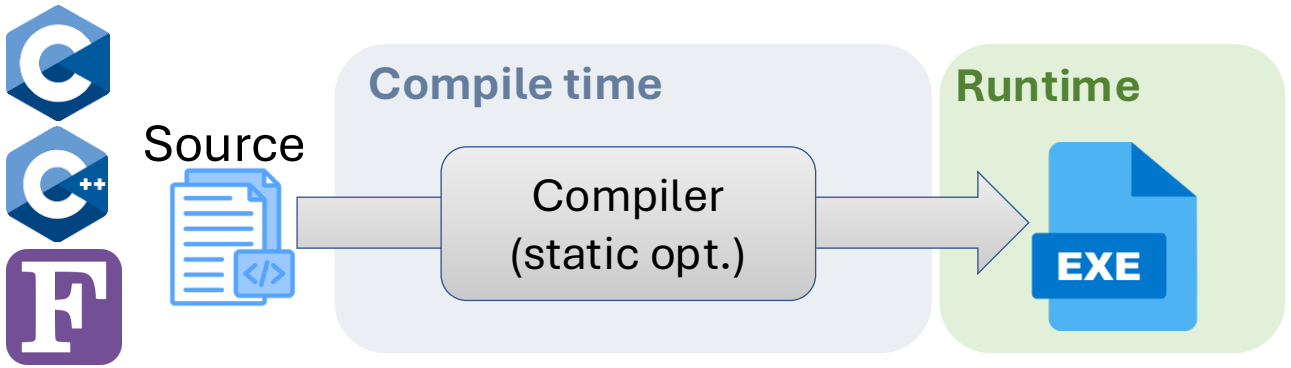
master Laghos / laghos_solver.cpp

Code Blame 1415 lines (1326 loc) · 49 KB Raw

```
1198    void QKernel(const int NE, const int NQ,
1307       const int id = (dim << 4) | Q1D;
1308       typedef void (*fQKernel)(const int NE, const int NQ,
1309                                const bool use_viscosity,
1310                                const bool use_vorticity,
1311                                const double h0, const double h1order,
1312                                const double cfl, const double infinity,
1313                                const ParGridFunction &gamma_gf,
1314                                const Array<double> &weights,
1315                                const Vector &Jacobians, const Vector &rho0DetJ0w,
1316                                const Vector &e_quads, const Vector &grad_v_ext,
1317                                const DenseTensor &Jac0inv,
1318                                Vector &dt_est, DenseTensor &stressJinvT);
1319       static std::unordered_map<int, fQKernel> qupdate =
1320       {
1321          // 2D.
1322          {0x24,&QKernel<2,4>}, {0x26,&QKernel<2,6>},
1323          {0x28,&QKernel<2,8>}, {0x2A,&QKernel<2,10>},
1324          // 3D.
1325          {0x34,&QKernel<3,4>}, {0x36,&QKernel<3,6>}, {0x38,&QKernel<3,8>}
1326       };
1327       if (!qupdate[id])
1328       {
1329          mfem::out << "Unknown kernel 0x" << std::hex << id << std::endl;
1330          MFEM_ABORT("Unknown kernel");
1331       }
1332       qupdate[id](NE, NQ, use_viscosity, use_vorticity, qdata.h0, h1order,
```

The Just-in-Time (JIT) compilation landscape: it's hard for statically compiled languages



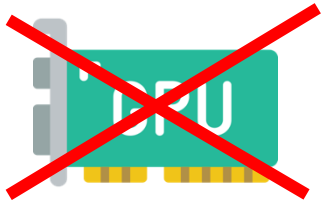


- Challenges for JIT in C/C++/Fortran
  - Introspection is hard for statically compiled lang.
  - Hard to beat static compilation optimization
  - Overhead

Prior work is inspirational, but obsolete, non-portable, slow

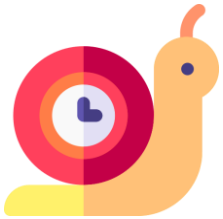
EasyJIT

C++ functors



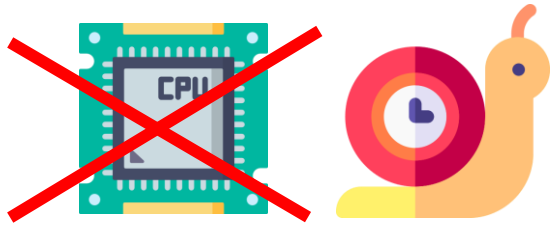
ClangJIT

C++ templates

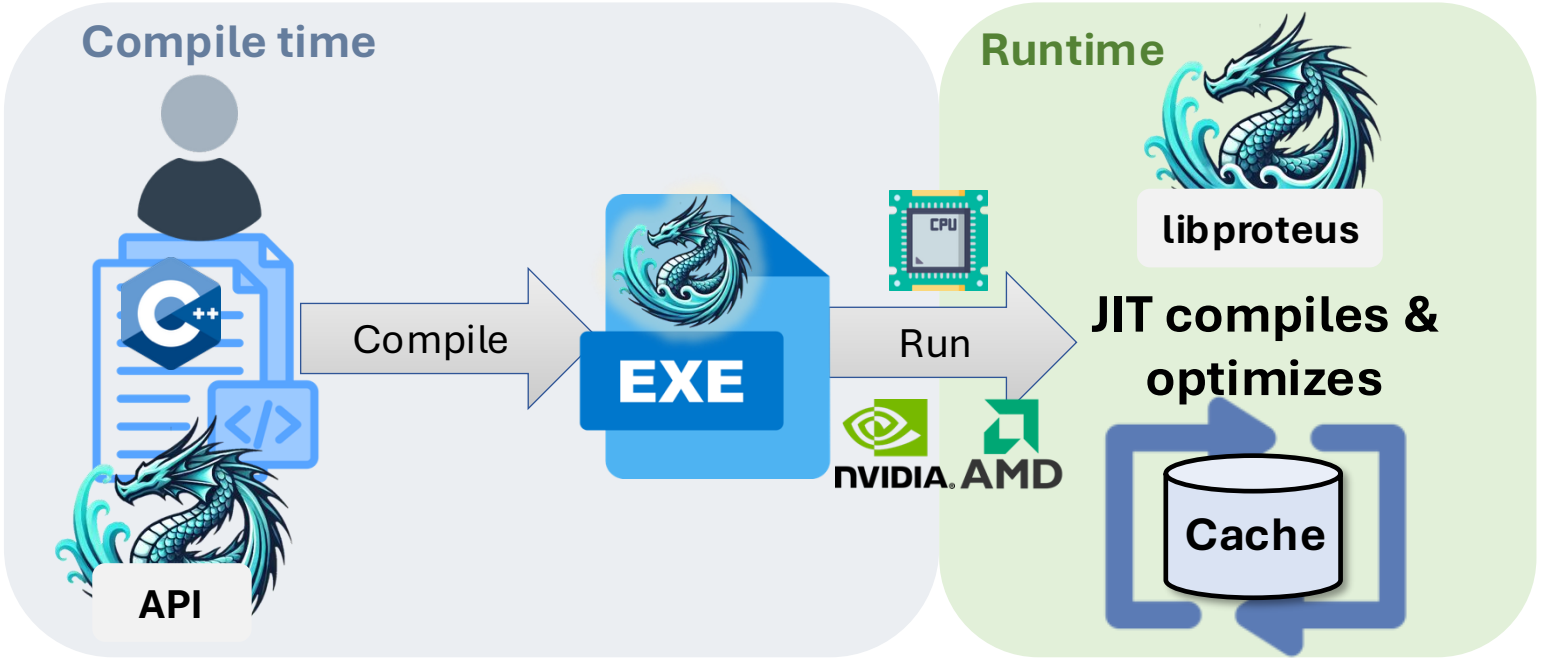


CUDA/HIP RTC (CUDA Jitify)

CUDA/HIP only



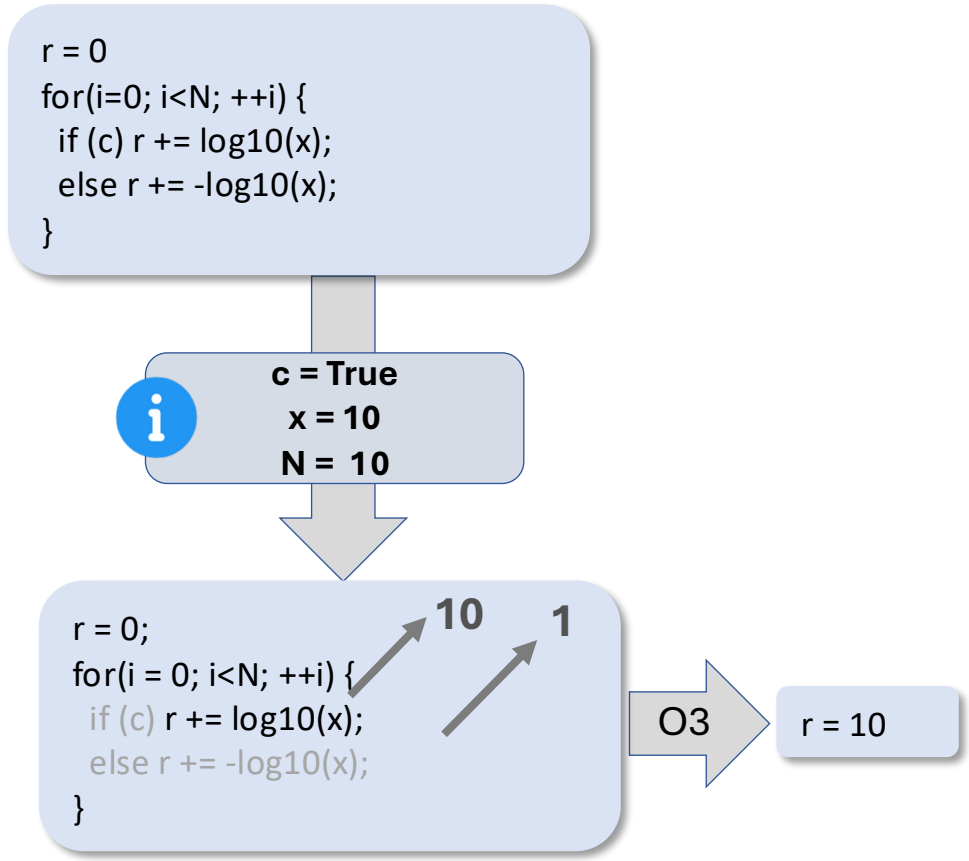
Workflow: developer uses the Proteus API, compiles, runs with JIT compilation and optimization enabled



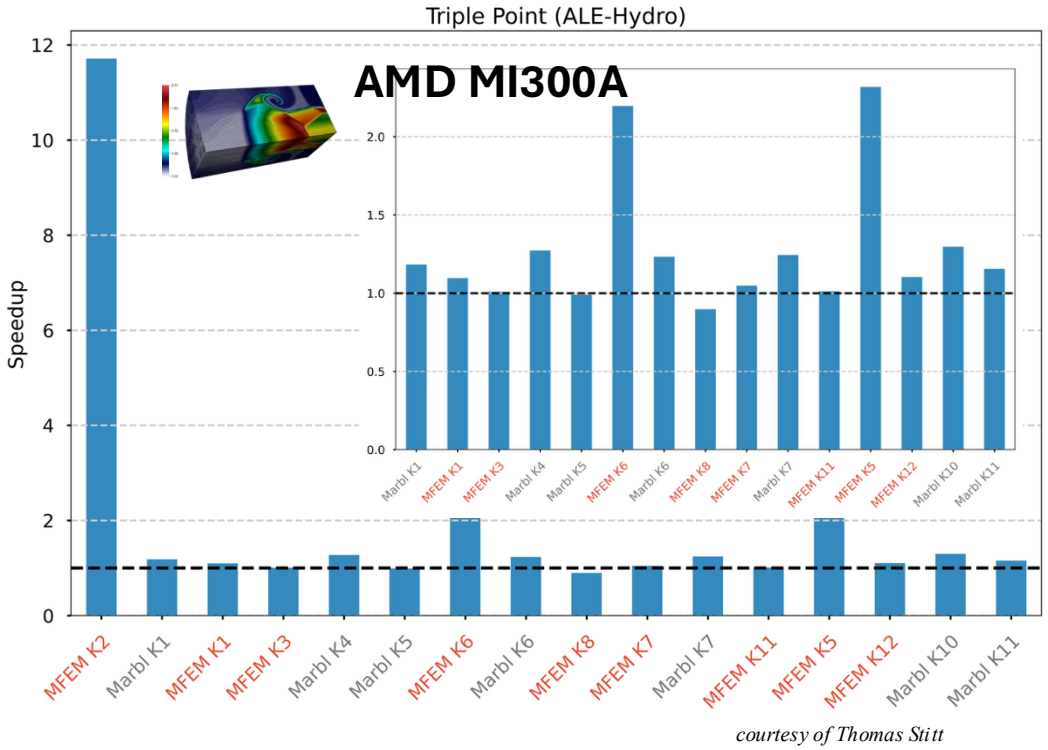
- APIs
  - Code annotations (Annotation)
  - C++ Frontend (PJ-CPP)
  - Embedded Domain Specific Language (PJ-DSL)
- Optimization at runtime
  - Runtime constexpr
  - Template instantiations
  - GPU launch parameters
  - Customize compiler pipeline per kernel

Proteus' central optimization is runtime constant folding

- **Runtime Constant Folding**
  - Scalars, arrays, objects
    - API dependent
  - GPU launch parameters: number of threads, blocks
  - Eliminate computation, unroll loops, inlining, algebraic transformations, …

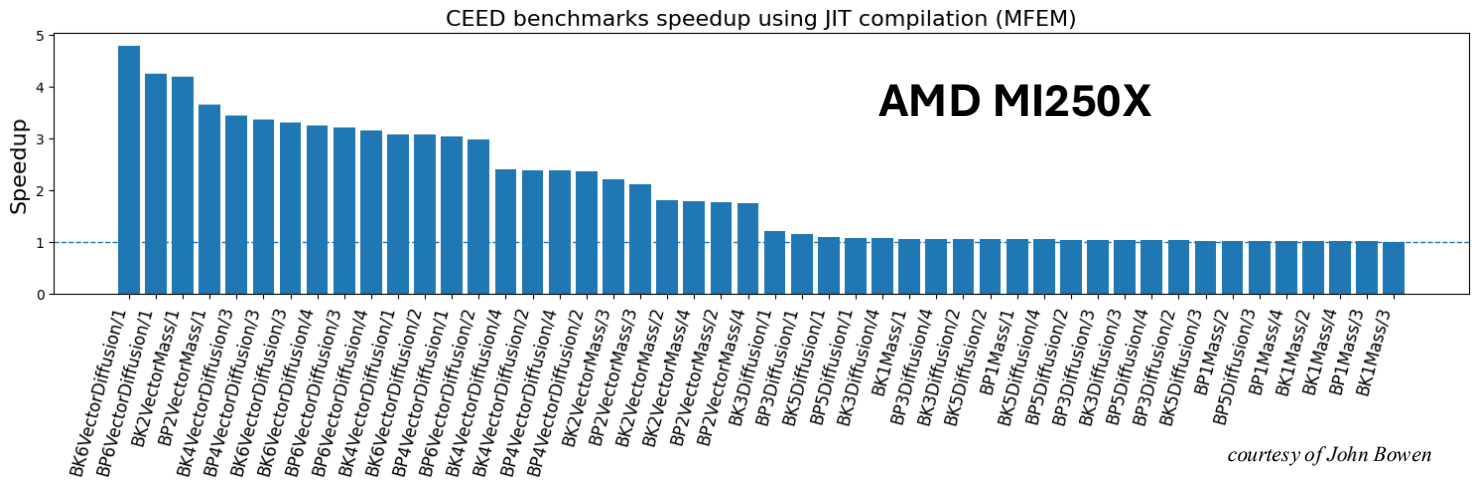


Proteus JIT optimization significantly reduces execution time with minimal overhead



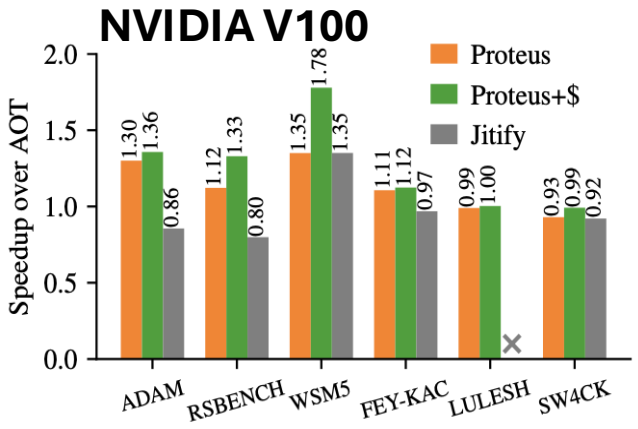
courtesy of Thomas Stitt

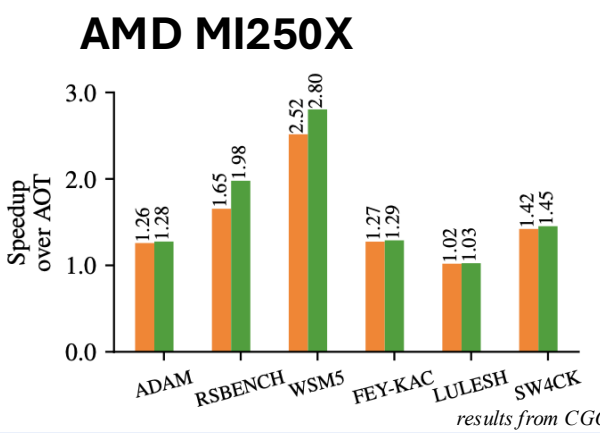
- 16 of the top 20 kernels are in Marbl or MFEM
- These 16 kernels represent 61.5% of the kernel runtime
- Total runtime is 35.2% faster with JIT specialization



courtesy of John Bowen

Heterogeneous Computing Benchmarks (HeCBench)





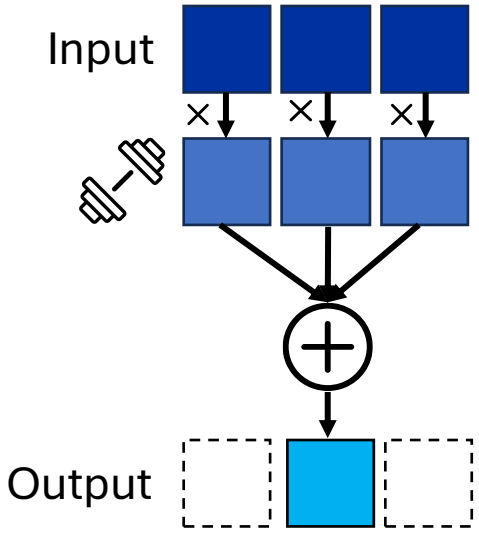
results from CGO25 paper

Overview

1. **Proteus JIT APIs**
    1. Annotation
    2. PJ-CPP
    3. PJ-DSL
2. Building Proteus
3. Proteus JIT Internals
4. Proteus JIT Performance

The Code Annotation API is easy to apply to existing code

JIT Opportunities:

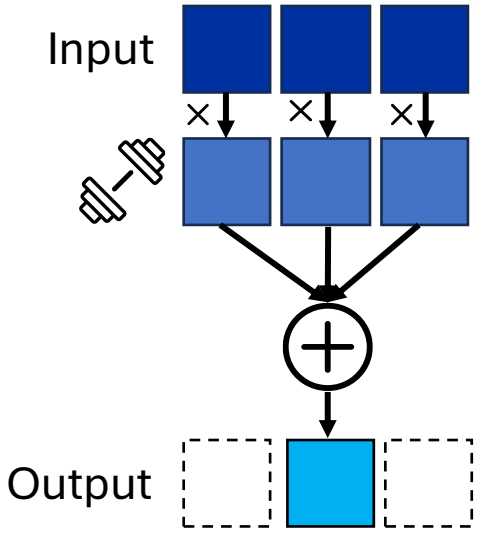



```
void stencil1d(float* out, float* in, size_t N,
               int radius, float* weights)
{
    extern __shared__ float tile[];
    int gid = getGlobalThreadIdX();
    int tid = getThreadIdX();

    tile[tid + radius] = in[gid];
    if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
    }
    __syncthreads();

    float sum = 0.0f;
    for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
    out[gid] = sum;
}
```

The Code Annotation API is easy to apply to existing code

JIT Opportunities:

- N, radius are runtime constants

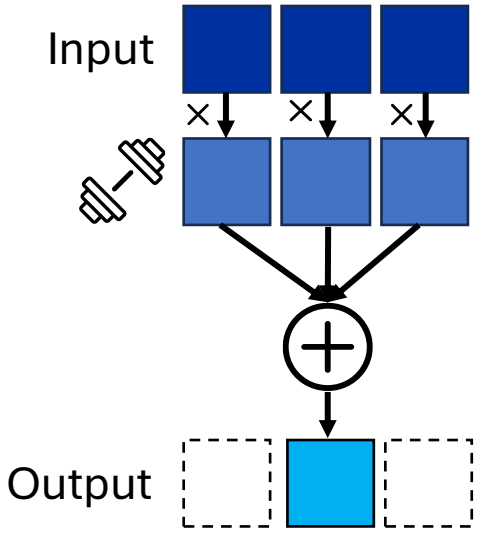



```
void stencil1d(float* out, float* in, size_t N,
               int radius, float* weights)
{
    extern __shared__ float tile[];
    int gid = getGlobalThreadIdX();
    int tid = getThreadIdX();

    tile[tid + radius] = in[gid];
    if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
    }
    __syncthreads();

    float sum = 0.0f;
    for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
    out[gid] = sum;
}
```

The Code Annotation API is easy to apply to existing code

JIT Opportunities:

- N, radius are runtime constants
- *weights* is constant

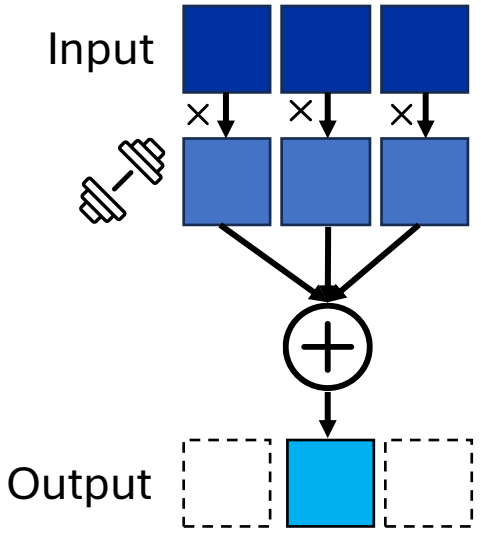



```
void stencil1d(float* out, float* in, size_t N,
               int radius, float* weights)
{
    extern __shared__ float tile[];
    int gid = getGlobalThreadIdX();
    int tid = getThreadIdX();

    tile[tid + radius] = in[gid];
    if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
    }
    __syncthreads();

    float sum = 0.0f;
    for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
    out[gid] = sum;
}
```

The Code Annotation API is easy to apply to existing code

JIT Opportunities:

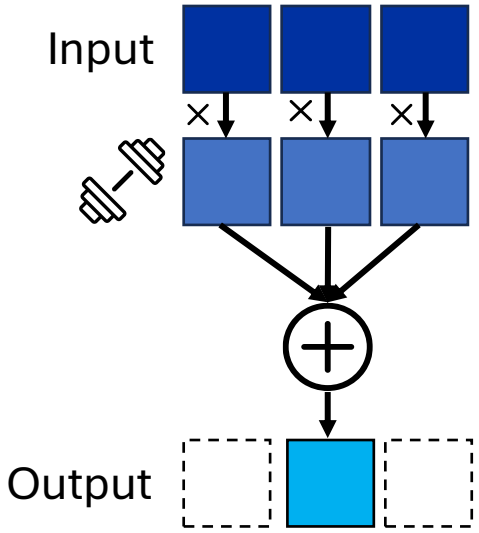
- N, radius are runtime constants
- *weights* is constant
- Tile is dynamic shared memory




```
void stencil1d(float* out, float* in, size_t N,
               int radius, float* weights)
{
    extern __shared__ float tile[];
    int gid = getGlobalThreadIdX();
    int tid = getThreadIdX();

    tile[tid + radius] = in[gid];
    if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
    }
    __syncthreads();

    float sum = 0.0f;
    for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
    out[gid] = sum;
}
```

The Code Annotation API is easy to apply to existing code

__attribute__((annotate("jit", 1, …, N)))



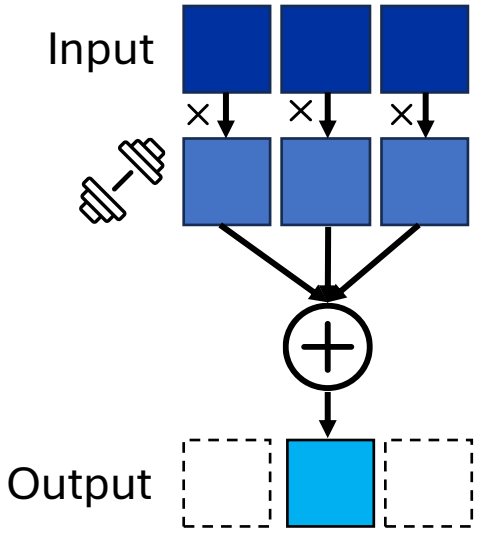
Argument index

```
__attribute__((annotate("jit", 3, 4)))  // N, radius
__global__
void stencil1d(float* out, float* in, size_t N,
               int radius, float* weights)
{
    extern __shared__ float tile[];
    int gid = getGlobalThreadIdX();
    int tid = getThreadIdX();

    tile[tid + radius] = in[gid];
    if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
    }
    __syncthreads();

    float sum = 0.0f;
    for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
    out[gid] = sum;
}
```

The Code Annotation API can JIT-compile constant arrays

__attribute__((annotate(“jit”, 1, …, N)))

proteus::jit_array



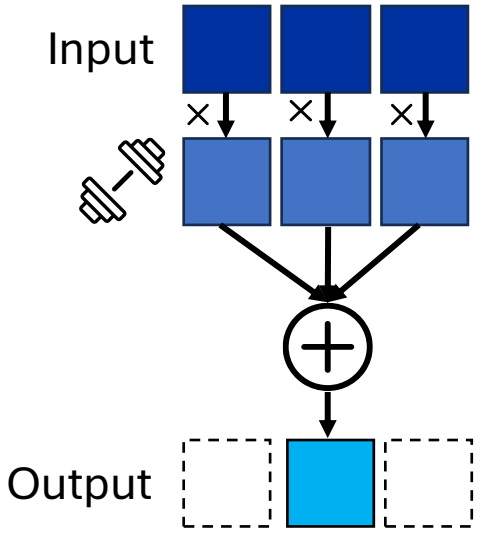

```
__attribute__((annotate("jit", 3, 4))) __global__
void stencil1d(float *out, float *in, size_t N,
               int radius, float *weights) {
  proteus::jit_array(weights, NWeights);
  extern __shared__ float tile[];
  int gid = getGlobalThreadIdX();
  int tid = getThreadIdX();

  tile[tid + radius] = in[gid];
  if (tid < radius) {
    tile[tid] = in[gid - radius];
    tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
  }
  __syncthreads();

  float sum = 0.0f;
  for (int j = -radius; j <= radius; j++)
    sum += tile[tid + radius + j] * weights[radius + j];
  out[gid] = sum;
}
```

The Code Annotation API converts dynamic shared memory to static

__attribute__((annotate(“jit”, 1, …, N)))

proteus::jit_array

proteus::shared_array



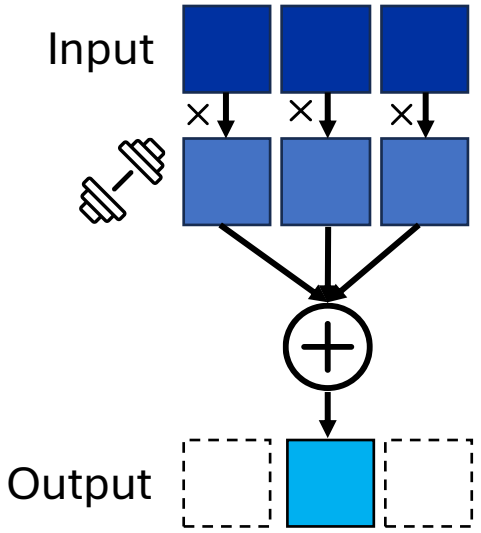

```
__attribute__((annotate("jit", 3, 4))) __global__
void stencil1d(float *out, float *in, size_t N,
               int radius, float *weights, int SMSize) {
  proteus::jit_array(weights, NWeights);
  float *tile = proteus::shared_array<float, 10>(SMSize);
  int gid = getGlobalThreadIdX();
  int tid = getThreadIdX();

  tile[tid + radius] = in[gid];
  if (tid < radius) {
    tile[tid] = in[gid - radius];
    tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
  }
  __syncthreads();

  float sum = 0.0f;
  for (int j = -radius; j <= radius; j++)
    sum += tile[tid + radius + j] * weights[radius + j];
  out[gid] = sum;
}
```

Launch Bound, Dimension Specialization expose additional optimization opportunities

__attribute__((annotate(“jit”, 1, …, N)))

proteus::jit_array

proteus::shared_array




```
__attribute__((annotate("jit", 3, 4))) __global__
void stencil1d(float *out, float *in, size_t N,
               int radius, float *weights, int SMSize) {
  proteus::jit_array(weights, NWeights);
  float *tile = proteus::shared_array<float, 10>(SMSize);
  int gid = range(0, 32768) getGlobalIdx();
  int tid = range(0, 128) getThreadIdx();

  tile[tid + radius] = in[gid];
  if (tid < radius) {
    tile[tid] = in[gid - radius];
    tile[tid + 128 + radius] = in[gid + 128];
  }

  __syncthreads();

  float sum = 0.0f;
  for (int j = -radius; j <= radius; j++)
    sum += tile[tid + radius + j] * weights[radius + j];
  out[gid] = sum;
}
```

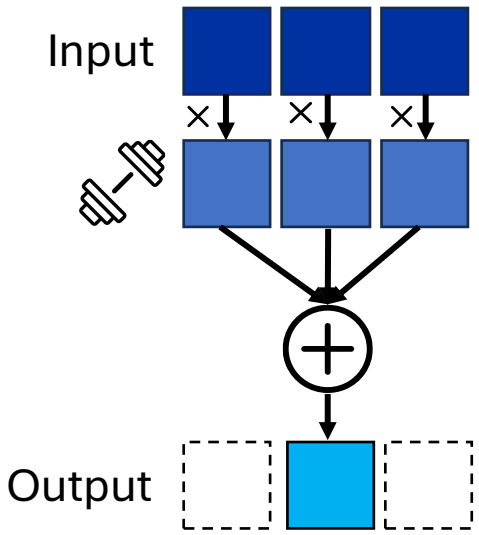
The Annotation API applies to lambda functions

__attribute__((annotate("jit", 1, ..., N)))

proteus::jit_array

proteus::shared_array

proteus::register_lambda




```
auto Kernel =
[=, radius = proteus::jit_variable(radius),
  tileSize = proteus::jit_variable(tileSize)] __device__(float *weights)
    __attribute__((annotate("jit"))) {
      proteus::jit_array(weights, NWeights);
      float *tile = proteus::shared_array<float, MaxTileSize>(tileSize);
      int gid = getGlobalThreadIdX();
      int tid = getThreadIdX();

      tile[tid + radius] = in[gid];
      if (tid < radius) {
        tile[tid] = in[gid - radius];
        tile[tid + blockDim.x + radius] = in[gid + blockDim.x];
      }
      __syncthreads();

      float sum = 0.0f;
      for (int j = -radius; j <= radius; j++)
        sum += tile[tid + radius + j] * weights[radius + j];
      out[gid] = sum;
    };
```

The C++ Frontend is a portable RTC implementation that supports specialization

```
double A = ...
int N = ...

std::string Code = std::format(R"cpp(
extern "C"
__global__
void daxpy(double *X, double *Y)
{{
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int stride = blockDim.x * gridDim.x;
    for(int i=tid; i<{0}; i+=stride)
        X[i] = {1}*X[i] + Y[i];
}})cpp", N, A);

CppJitModule CJM{"cuda", Code};
auto Kernel = CJM.getKernel<void(double *, double *)>("daxpy");
Kernel.launch(
    /* GridDim */ {NumBlocks, 1, 1},
    /* BlockDim */ {ThreadsPerBlock, 1, 1},
    /* ShmemSize */ 0,
    /* Stream */ nullptr,
    X, Y);
```

Code with specialization

Create Jit Module

Get Kernel handle (will compile)

Launch!

Format →

```
extern "C" __global__
void daxpy(double *X, double *Y)
{
  int tid = threadIdx.x + (blockIdx.x * blockDim.x);
  int stride = blockDim.x * gridDim.x;
  for(int i=tid; i < 2048; i+=stride)
    X[i] = 0.5*X[i] + Y[i];
}
```

C++ Frontend enables runtime template instantiation

Templated code

```
std::string Code = R"cpp(
    template<typename T, int N>
    __global__ void axpy(double A, T *X, T *Y) {
        size_t I = blockIdx.x * 256 + threadIdx.x;
        if (I < N)
            Y[I] += X[I] * A;
    }
)cpp";

constexpr int N = 1024;
constexpr int NumThreadsPerBlock = 256;
constexpr int NumBlocks = (N + NumThreadsPerBlock - 1) / NumThreadsPerBlock;
CppJitModule CJM{"hip", Code};

float *X_f, *Y_f;
...
auto AxpyFloat = CJM.instantiate("axpy", "float", std::to_string(N));
AxpyFloat.launch({NumBlocks, 1, 1}, {NumThreadsPerBlock, 1, 1}, 0, nullptr,
                 1.0, X_f, Y_f);

double *X_d, *Y_d;
...
auto AxpyDouble = CJM.instantiate("axpy", "double", std::to_string(N));
AxpyDouble.launch({NumBlocks, 1, 1}, {NumThreadsPerBlock, 1, 1}, 0, nullptr,
                  1.0, X_d, Y_d);
```

Function **Type** **Value**

Get instance handle (will compile)

C++ Frontend is string templating engine agnostic

`std::format()`

{fmt}

mustache

```
std::string Code =R"cpp( __global__
  extern "C" void stencil1d(float *out, float *in, float *weights, int tileSize) {
      extern __shared__ float tile[];
      int tid = threadIdx.x;
      int gid = blockIdx.x * blockDim.x + tid;

      tile[tid + {{ radius }}] = in[gid];
      if (tid < {{ radius }}) {
          tile[tid] = in[gid - {{ radius }}];
          tile[tid + blockDim.x + {{ radius }}] = in[gid + blockDim.x];
      }
      __syncthreads();

      float sum = 0.0f;
      #pragma unroll
      for (int j = -{{ radius }}; j <= {{ radius }}; ++j)
          sum += tile[tid + {{ radius }} + j] * weights[{{ radius }} + j];
      out[gid] = sum;
  }
 )cpp";
```

The C++ frontend supports the full suite of specialization

```
std::string Code = R"cpp(__global__
  extern "C" void stencil1d_cpp(float *out, float *in) {
      const float weights[] = {{ weights }};
      __shared__ float tile[{{ tileSize }}];
      int tid = threadIdx.x;
      int gid = blockIdx.x * {{ blockSize }} + tid;

      tile[tid + {{ radius }}] = in[gid];
      if (tid < {{ radius }}) {
          tile[tid] = in[gid - {{ radius }}];
          tile[tid + {{ blockSize }} + {{ radius }}] = in[gid + {{ blockSize }}];
      }
      __syncthreads();

      float sum = 0.0f;
      #pragma unroll
      for (int j = -{{ radius }}; j <= {{ radius }}; ++j)
          sum += tile[tid + {{ radius }} + j] * weights[{{ radius }} + j];
      out[gid] = sum;
  }
 )cpp";
```

PJ-DSL constructs LLVM IR at runtime

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);









  return std::make_pair(std::move(J), KernelHandle);
}
```

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"
```

Kernels are added to JitModules with their name and signature

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;









  return std::make_pair(std::move(J), KernelHandle);
}
```

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"

define amdgpu_kernel void @daxpy(ptr %0, ptr %1) {
entry:









}
```

PJ-DSL uses paired constructs for control structures

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;

  F.beginFunction();
  {









  }
  F.endFunction();

  return std::make_pair(std::move(J), KernelHandle);
}
```

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"

define amdgpu_kernel void @daxpy(ptr %0, ptr %1) {
entry:


  br label %body
body:
  br label %exit









exit:
  unreachable
}
```

```
F.beginIf(Cond);
F.beginFor(Iter, Init, UB, Inc);
F.beginWhile([&](){return A < B;});
```

Typed Var objects wrap LLVM scalar types

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;

  F.beginFunction();
  {
    auto [X, Y] = F.getArgs();
    Var<const double> A = F.defRuntimeConst(_A);
    Var<const size_t> N = F.defRuntimeConst(_N);

    Var<size_t> I = F.declVar<size_t>("I");









  }


  return std::make_pair(std::move(J), KernelHandle);
}
```

Var<double*>

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"

define amdgpu_kernel void @daxpy(ptr %0, ptr %1) {
entry:
  %A = alloca double
  ; Remaining allocas.
  br label %body
body:
  store double 6.0e+00, ptr %A
  • • •
  br label %exit









exit:
  unreachable
}
```

PJ-DSL provides access to bult-in device functions

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;

  F.beginFunction();
  {
    auto [X, Y] = F.getArgs();
    Var<const double> A = F.defRuntimeConst(_A);
    Var<const size_t> N = F.defRuntimeConst(_N);

    Var<size_t> I = F.declVar<size_t>("I");
    I = F.callBuiltin(getBlockIdX) * F.callBuiltin(getBlockDimX) +
        F.callBuiltin(getThreadIdX);









  }


  return std::make_pair(std::move(J), KernelHandle);
}
```

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"

define amdgpu_kernel void @daxpy(ptr %0, ptr %1) {
entry:
  %A = alloca double
  ; Remaining allocas.
  br label %body
body:
  store double 6.0e+00, ptr %A
  %12 = call i32 @llvm.amdgcn.workitem.id.x()
  store i32 %12, ptr %threadIdx.x, align 4
  • • •
  br label %exit









exit:
  unreachable
}
```

Screenshot of other builtins

Var objects support arithmetic, math operations

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;

  F.beginFunction();
  {
    auto [X, Y] = F.getArgs();
    Var<const double> A = F.defRuntimeConst(_A);
    Var<const size_t> N = F.defRuntimeConst(_N);

    Var<size_t> I = F.declVar<size_t>("I");
    I = F.callBuiltin(getBlockIdX) * F.callBuiltin(getBlockDimX) +
        F.callBuiltin(getThreadIdX);
    auto [J, Inc, Zero] = F.defVars(0, 1, 0);
    F.beginFor(J, Zero, N, Inc);
    { Y[I] = Y[I] + X[I] * A; }
    F.endFor();

    F.ret();
  }
  F.endFunction();

  return std::make_pair(std::move(J), KernelHandle);
}
```

```
; ModuleID = 'JitModule'
source_filename = "JitModule"
target triple = "amdgcn-amd-amdhsa"

define amdgpu_kernel void @daxpy(ptr %0, ptr %1) {
entry:
  %A = alloca double
  ; Remaining allocas.
  br label %body
body:
  store double 6.0e+00, ptr %A
  %12 = call i32 @llvm.amdgcn.workitem.id.x()
  store i32 %12, ptr %threadIdx.x, align 4
  ; Remaining stores. Set up loop condition
  br label %loop.header
loop.header:
  ...
  br label %loop.cond
loop.cond:
  ; Calculate J < N
  br i1 %23, label %loop.body, label %loop.end
loop.body:
  ; Compute Y[I] += X[I] * A;
  br label %loop.inc
loop.inc:
  ; J += Computed grid size
  br label %loop.cond
loop.end:
  br label %body.split
body.split:
  ret void
exit:
  unreachable
}
```

```
F.min(A, B);
F.pow(X, Y);
F.cos(N);
F.sin(M);
...
```

PJ-DSL kernels are portable between GPU vendors

```
auto createJitKernel(double _A, size_t _N) {
  auto J = std::make_unique<JitModule>(TARGET);

  auto KernelHandle = J->addKernel<void(double *, double *)>("daxpy");
  auto &F = KernelHandle.F;

  F.beginFunction();
  {
    auto [X, Y] = F.getArgs();
    Var<const double> A = F.defRuntimeConst(_A);
    Var<const size_t> N = F.defRuntimeConst(_N);

    Var<size_t> I = F.declVar<size_t>("I");
    I = F.callBuiltin(getBlockIdX) * F.callBuiltin(getBlockDimX) +
        F.callBuiltin(getThreadIdX);
    auto [J, Inc, Zero] = F.defVars(0, 1, 0);
    F.beginFor(J, Zero, N, Inc);
    { Y[I] = Y[I] + X[I] * A; }
    F.endFor();

    F.ret();
  }
  F.endFunction();

  return std::make_pair(std::move(J), KernelHandle);
}
```

```
std::string TARGET = "hip";
```

```
J.compile();
```

AMD

```
std::string TARGET = "cuda";
```

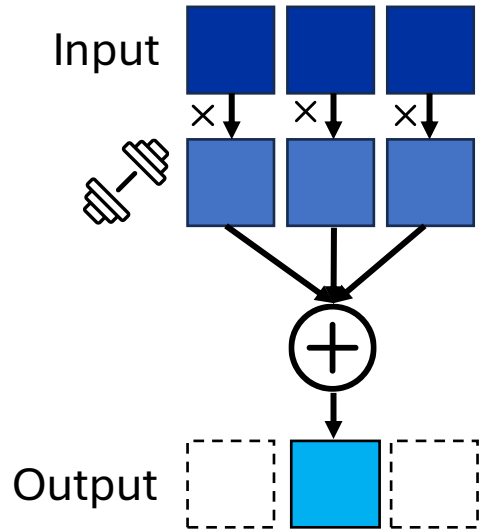
```
J.compile();
```

NVIDIA

PJ-DSL supports constant, variable arrays

```
F.beginFunction(); {
  auto [Out, In] = F.getArgs();
  auto Radius = F.defRuntimeConst<int>(Radius_);
  auto Weights = F.defRuntimeConst<float[]>(Weights_, NumWeights_);
  auto Tile = F.declVar<float[]>(TileSize_, AddressSpace::SHARED);

  auto Tid = F.callBuiltin(getThreadIdX);
  auto Gid = F.callBuiltin(getBlockIdX) * F.callBuiltin(getBlockDimX) + Tid;

  Tile[Tid + Radius] = In[Gid];

  F.beginIf(Tid < Radius); {
    Tile[Tid] = In[Gid - Radius];
    Tile[Tid + F.callBuiltin(getBlockDimX) + Radius] =
        In[Gid + F.callBuiltin(getBlockDimX)];
  } F.endIf();

  F.callBuiltin(syncThreads);

  auto Sum = F.defVar<float>(0.0f);
  auto J = F.declVar<int>("j");

  F.beginFor(J, -Radius, Radius+1, F.defVar(1)); {
    Sum += Tile[Tid + Radius + J] * Weights[Radius + J];
  } F.endFor();

  Out[Gid] = Sum;

  F.ret();
} F.endFunction();
```

```
@Weights =  addrspace(4) constant [5 x float] \
           [float 0.0, float 0.1, float 0.2, float 0.3, float 0.4]
@Tile =  addrspace(3) global [68 x float]
```



PJ-DSL function calls enable modular code

```
F.beginFunction(); {
  auto [Out, In] = F.getArgs();
  auto Radius = F.defRuntimeConst(Radius_);
  auto Weights = F.defRuntimeConst(Weights_, NumWeights_);
  auto Tile = F.declVar<float[]>(TileSize_, AddressSpace::SHARED);

  auto Tid = F.callBuiltin(getThreadIdX);
  auto Gid = F.callBuiltin(getBlockIdX) * F.callBuiltin(getBlockDimX) + Tid;

  Tile[Tid + Radius] = In[Gid];

  F.beginIf(Tid < Radius); {
    Tile[Tid] = In[Gid - Radius];
    Tile[Tid + F.callBuiltin(getBlockDimX) + Radius] =
        In[Gid + F.callBuiltin(getBlockDimX)];
  } F.endIf();

  F.callBuiltin(syncThreads);

  auto BaseIdx = Tid + Radius;
  auto Sum = F.call("innerProduct", Tile, Weights, BaseIdx, Radius);

  Out[Gid] = Sum;

  F.ret();
} F.endFunction();
```

```
auto &Fn = J->addFunction<float(float[], const float[], int, int)>("innerProduct");
Fn.beginFunction(); {
  auto [Tile, Weights, BaseIdx, Radius] = Fn.getArgs();

  auto Sum = Fn.defVar<float>(0.0f);
  auto J = Fn.declVar<int>("j");
  auto One = Fn.defRuntimeConst(1);

  Fn.beginFor(J, -Radius, Radius + One, One); {
    Sum += Tile[BaseIdx + J] * Weights[Radius + J];
  } Fn.endFor();

  Fn.ret(Sum);
} Fn.endFunction();
```

Deferred code generation enables structural optimization in PJ-DSL

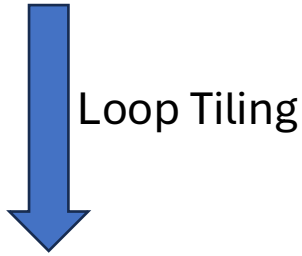
```
auto [C, A, B] = F.getArgs();
F.beginFunction();
{
  auto [I, J, K] = F.declVars<int, int, int>();
  auto [UbnI, UbnJ, UbnK, IncOne, Zero] = F.defRuntimeConsts(N, N, N, 1, 0);

  F.buildLoopNest(F.forLoop(I, Zero, UbnI, IncOne),
                  F.forLoop(J, Zero, UbnJ, IncOne),
                  F.forLoop(K, Zero, UbnK, IncOne,
                            [&]() {
                              auto CIdx = I * N + J;
                              auto AIdx = I * N + K;
                              auto BIdx = K * N + J;
                              C[CIdx] += A[AIdx] * B[BIdx];
                            })
                  )
      .emit();

  F.ret();
}
F.endFunction();
```

```
$ ./matmul 1024
Average runtime: 6403.990 ms
```

Deferred code generation enables structural optimization in PJ-DSL

```
auto [C, A, B] = F.getArgs();
F.beginFunction();
{
  auto [I, J, K] = F.declVars<int, int, int>();
  auto [UbnI, UbnJ, UbnK, IncOne, Zero] = F.defRuntimeConsts(N, N, N, 1, 0);

  F.buildLoopNest(F.forLoop(I, Zero, UbnI, IncOne),
                  F.forLoop(J, Zero, UbnJ, IncOne),
                  F.forLoop(K, Zero, UbnK, IncOne,
                            [&]() {
                              auto CIdx = I * N + J;
                              auto AIdx = I * N + K;
                              auto BIdx = K * N + J;
                              C[CIdx] += A[AIdx] * B[BIdx];
                            })
                  ).tile(TileSize)
      .emit();

  F.ret();
}
F.endFunction();
```

```
$ ./matmul 1024
Average runtime: 6403.990 ms
```




```
$ ./matmul 1024 16
Average runtime: 1003.984 ms
```

PJ-DSL has potential for many more optimizations

JIT Frontends trade off supported specializations, developer ease, and compiler portability

	Specializations						Developer Ease	Compiler Portability
	Value	Array	Object	Launch Bounds	Grid Dim.	Template		
Annot.	✅	✅	✅	✅	✅	❌	👍👍👍	👎
PJ-CPP	✅	✅	✅	✅	✅	✅	👍👍	👍
PJ-DSL	✅	✅	❌	✅	✅	❌	👍	👍

Overview

1. Proteus JIT APIs
    1. Annotation
    2. PJ-CPP
    3. PJ-DSL
2. **Building Proteus**
3. Proteus JIT Internals
4. Proteus JIT Performance

Proteus is simple to build using CMake

```
$ git clone git@github.com:Olympus-HPC/proteus.git
$ mkdir proteus/build && cd proteus/build
$ cmake .. \
  -DLLVM_INSTALL_DIR=${LLVM_INSTALL_DIR} \
  -DCMAKE_C_COMPILER=${LLVM_INSTALL_DIR}/bin/clang \
  -DCMAKE_CXX_COMPILER=${LLVM_INSTALL_DIR}/bin/clang++ \
  -DPROTEUS_ENABLE_HIP=on
```

AMD

```
$ git clone git@github.com:Olympus-HPC/proteus.git
$ mkdir proteus/build && cd proteus/build
$ cmake .. \
  -DLLVM_INSTALL_DIR="$LLVM_INSTALL_DIR" \
  -DPROTEUS_ENABLE_CUDA=on \
  -DCMAKE_CUDA_ARCHITECTURES=90 \
  -DCMAKE_C_COMPILER="$LLVM_INSTALL_DIR/bin/clang" \
  -DCMAKE_CXX_COMPILER="$LLVM_INSTALL_DIR/bin/clang++" \
  -DCMAKE_CUDA_COMPILER="$LLVM_INSTALL_DIR/bin/clang++"
```

NVIDIA

Installing Proteus with Spack is easy as 1, 2, 3!

Step 1: Add the repo

```
$ spack repo add https://github.com/Olympus-HPC/proteus/tree/main/packaging/spack
```

Installing Proteus with Spack is easy as 1, 2, 3!

Step 1: Add the repo

```
$ spack repo add https://github.com/Olympus-HPC/proteus/tree/main/packaging/spack
```

Step 2: Install Proteus!

```
$ spack install proteus@main
```

```
$ spack install proteus@main +hip
```

```
$ spack install proteus@main +cuda
```

Installing Proteus with Spack is easy as 1, 2, ~~3~~!

Step 1: Add the repo

```
$ spack repo add https://github.com/Olympus-HPC/proteus/tree/main/packaging/spack
```

Step 2: Install Proteus!

```
$ spack install proteus@main
```

```
$ spack install proteus@main +hip
```

```
$ spack install proteus@main +cuda
```

How to compile with Proteus enabled

Code Annotations API

- Requires Clang/LLVM
  - AMD: Vanilla
  - NVIDIA: Vanilla
- Add the ProteusPass plugin and link with libproteus and deps

```
clang++ -fpass-plugin=<path>/libProteusPass.so \
  -lproteus ${LLVM_LIBS} ${CLANG_LIBS}
```

- We provide cmake exports

How to compile with Proteus enabled

Code Annotations API

- Requires Clang/LLVM
  - AMD: Vanilla
  - NVIDIA: Vanilla
- Add the ProteusPass plugin and link with libproteus and deps

```
clang++ -fpass-plugin=<path>/libProteusPass.so \
  -lproteus ${LLVM_LIBS} ${CLANG_LIBS}
```

- We provide cmake exports

```
find_package(proteus CONFIG REQUIRED)
add_proteus(<target>)
```

How to compile with Proteus enabled

Code Annotations API

- Requires Clang/LLVM
  - AMD: Vanilla
  - NVIDIA: Vanilla
- Add the ProteusPass plugin and link with libproteus and deps

```
clang++ -fpass-plugin=<path>/libProteusPass.so \
  -lproteus ${LLVM_LIBS} ${CLANG_LIBS}
```

- We provide cmake exports

```
find_package(proteus CONFIG REQUIRED)
add_proteus(<target>)
```

C++ Frontend / DSL API

- Link with libproteus and deps

```
clang++ -lproteus ${LLVM_LIBS} ${CLANG_LIBS}
```

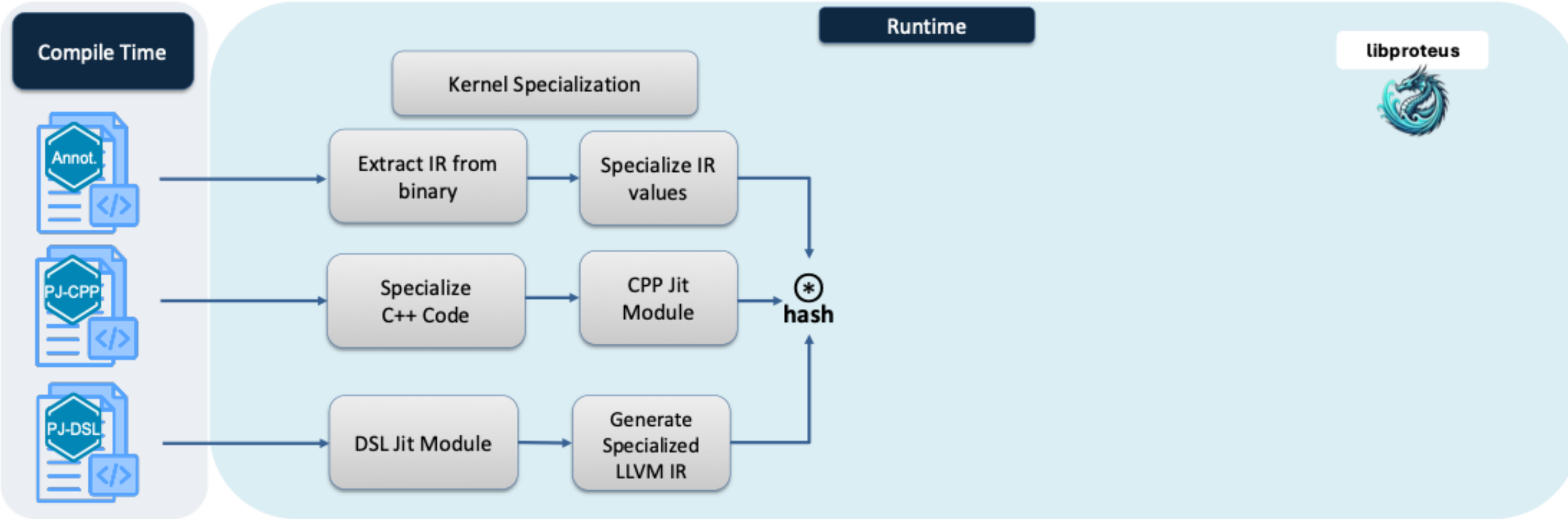
- We provide cmake exports

```
find_package(proteus CONFIG REQUIRED)
target_link_libraries(<target> proteusFrontend)
```

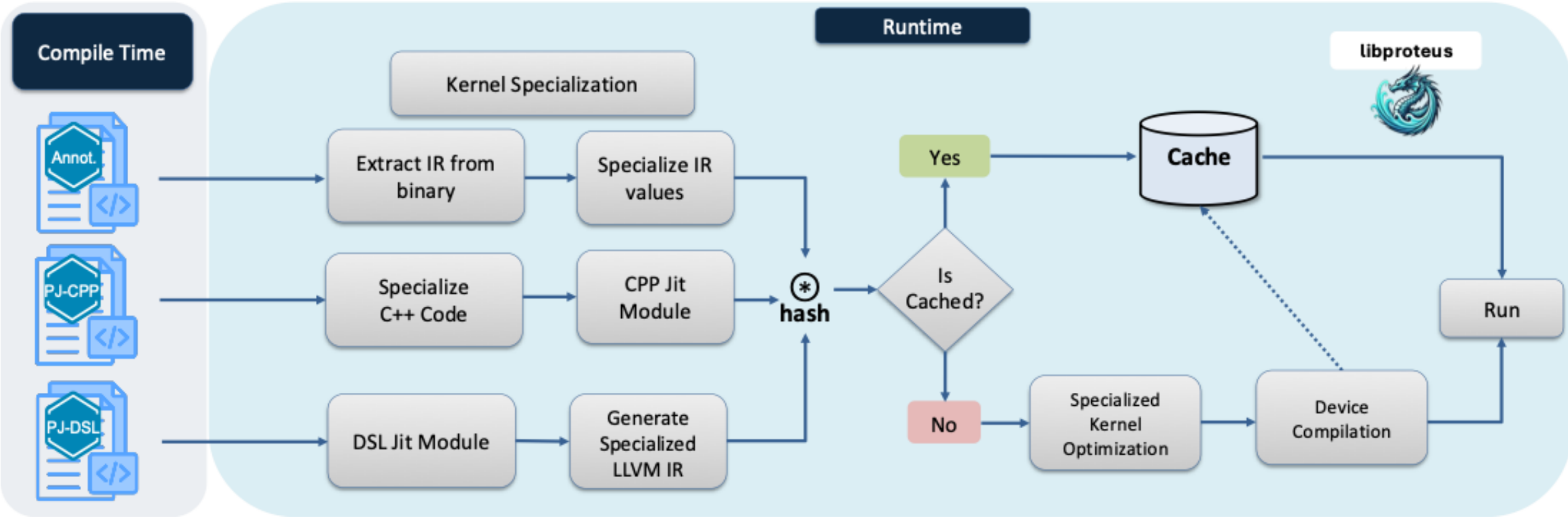
Overview

1. Proteus JIT APIs
    1. Annotation
    2. PJ-CPP
    3. PJ-DSL
2. Building Proteus
3. **Proteus JIT Internals**
4. Proteus JIT Performance

Proteus uses hashing to uniquely identify specializations

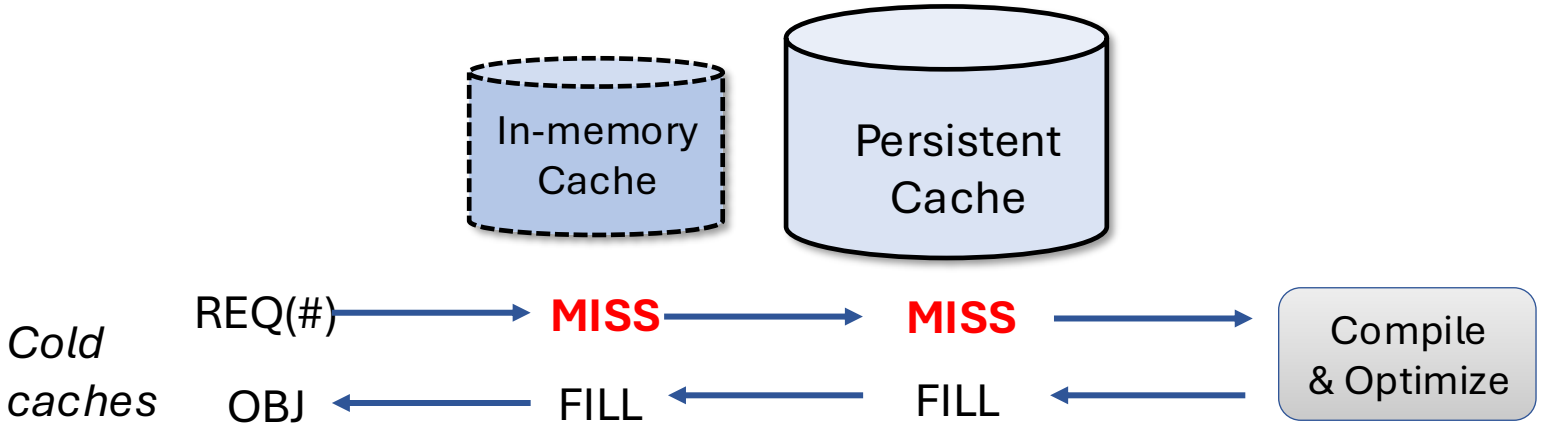


All JIT frontends use shared Proteus infrastructure



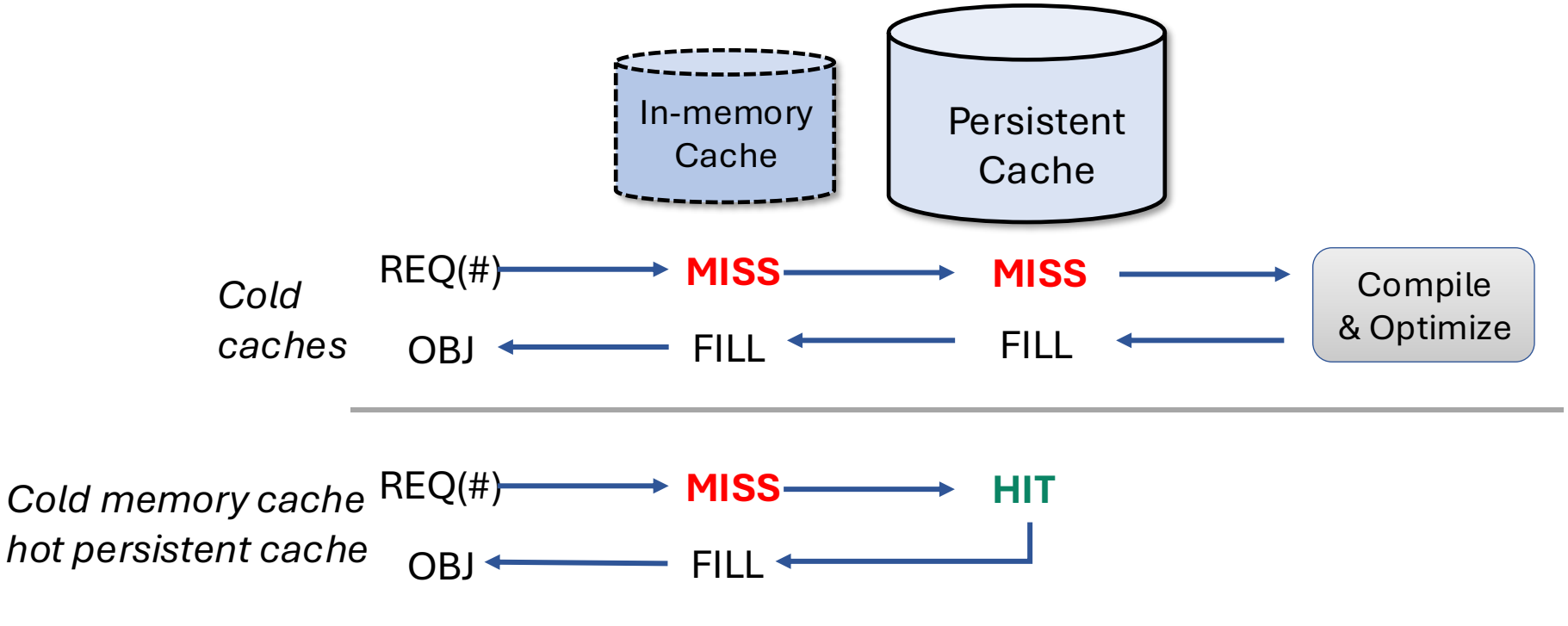
Caching in a two-level hierarchy with a volatile in-memory cache and a persistent file-based cache

- Key is the **hash**
- Stores the compiled object
- Caching is *inclusive*



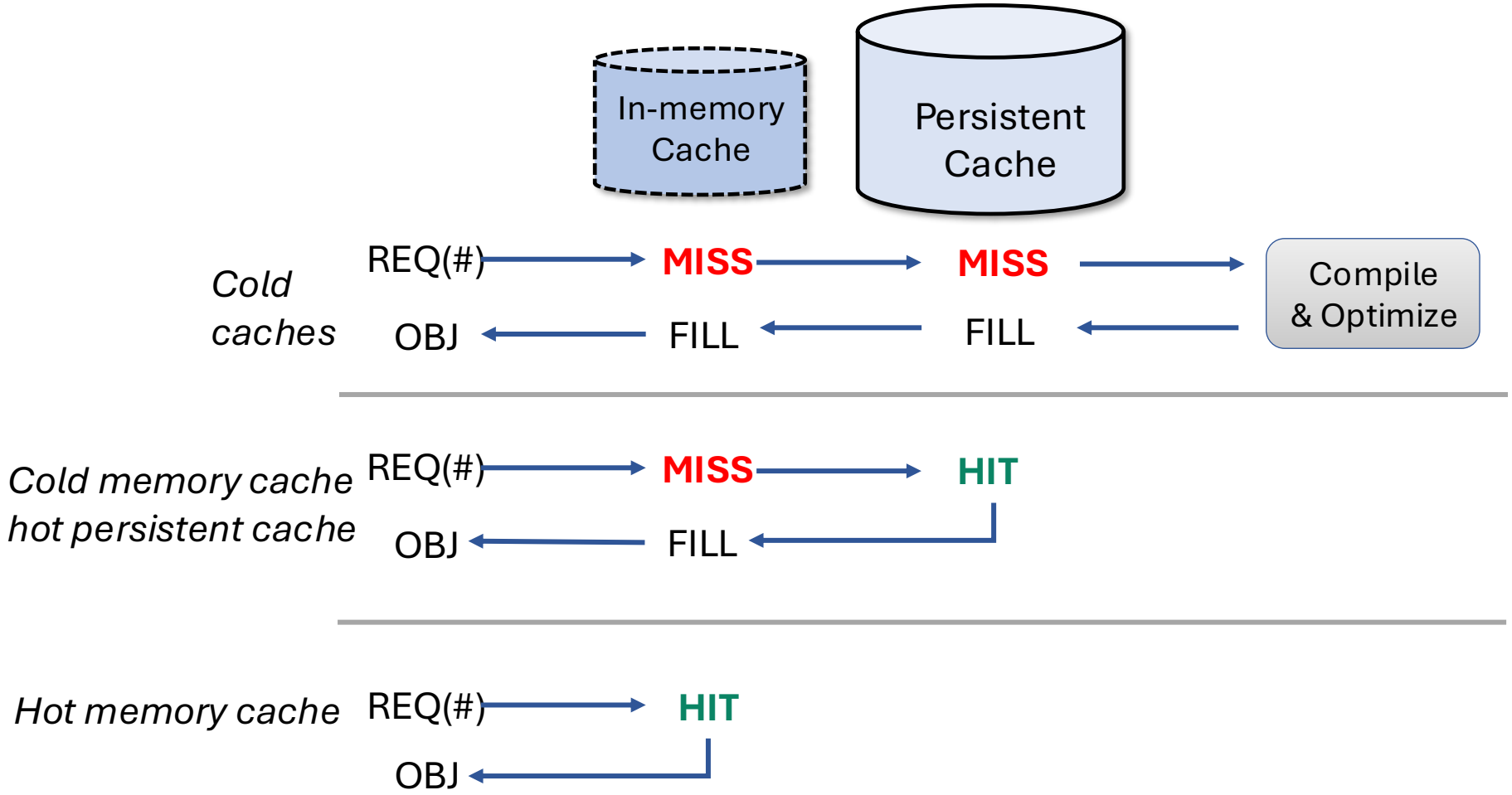
Caching in a two-level hierarchy with a volatile in-memory cache and a persistent file-based cache

- Key is the **hash**
- Stores the compiled object
- Caching is *inclusive*



Caching in a two-level hierarchy with a volatile in-memory cache and a persistent file-based cache

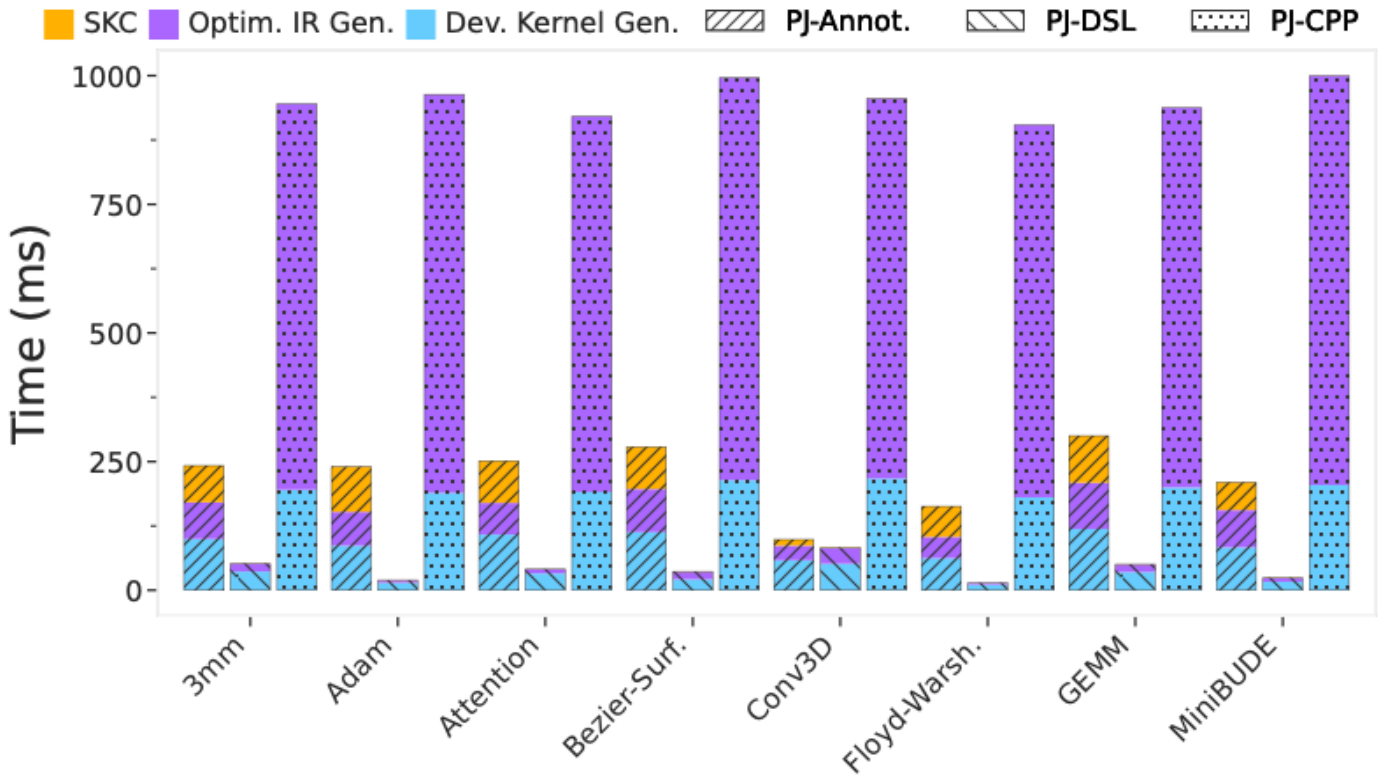
- Key is the **hash**
- Stores the compiled object
- Caching is *inclusive*



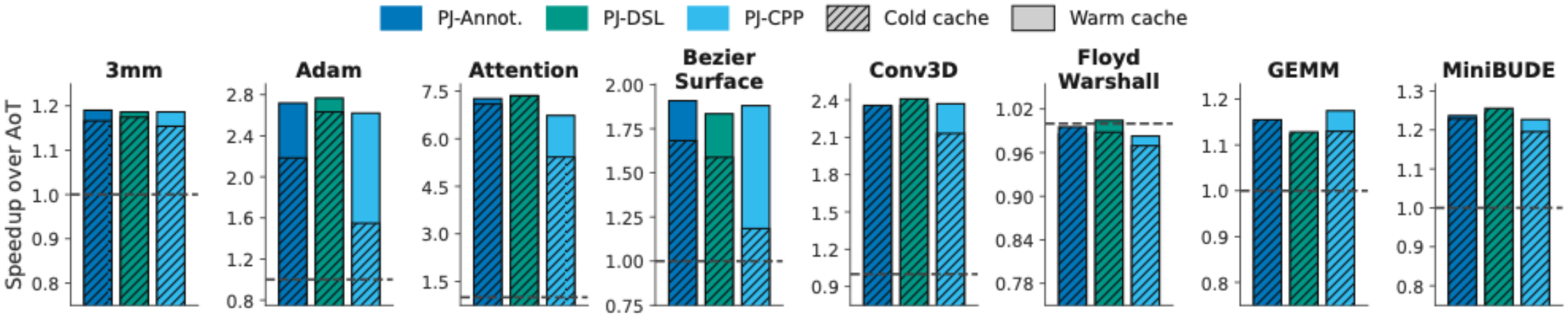
Overview

1. Proteus JIT APIs
    1. Annotation
    2. PJ-CPP
    3. PJ-DSL
2. Building Proteus
3. Proteus JIT Internals
4. **Proteus JIT Performance**

PJ-DSL has lowest JIT overhead, followed by Annotation and PJ-CPP



Compilation time can substantially affect JIT speedup



Summary of Covered Components

What did we cover

- Proteus specialization optimizations
  - Value, array, launch bounds, grid dim, template
- Proteus JIT Frontends
  - Annotation
  - PJ-CPP
  - PJ-DSL
- Proteus Internals
  - Compile, runtime
  - Caching
- JIT Performance
  - JIT stage breakdown
  - Cache impact

Additional capabilities (beyond this tutorial)

- Runtime configuration
- Asynchronous compilation
- Distributed caching modes
- Compiler pipeline customization

A Record-Replay approach for efficient auto-tuning

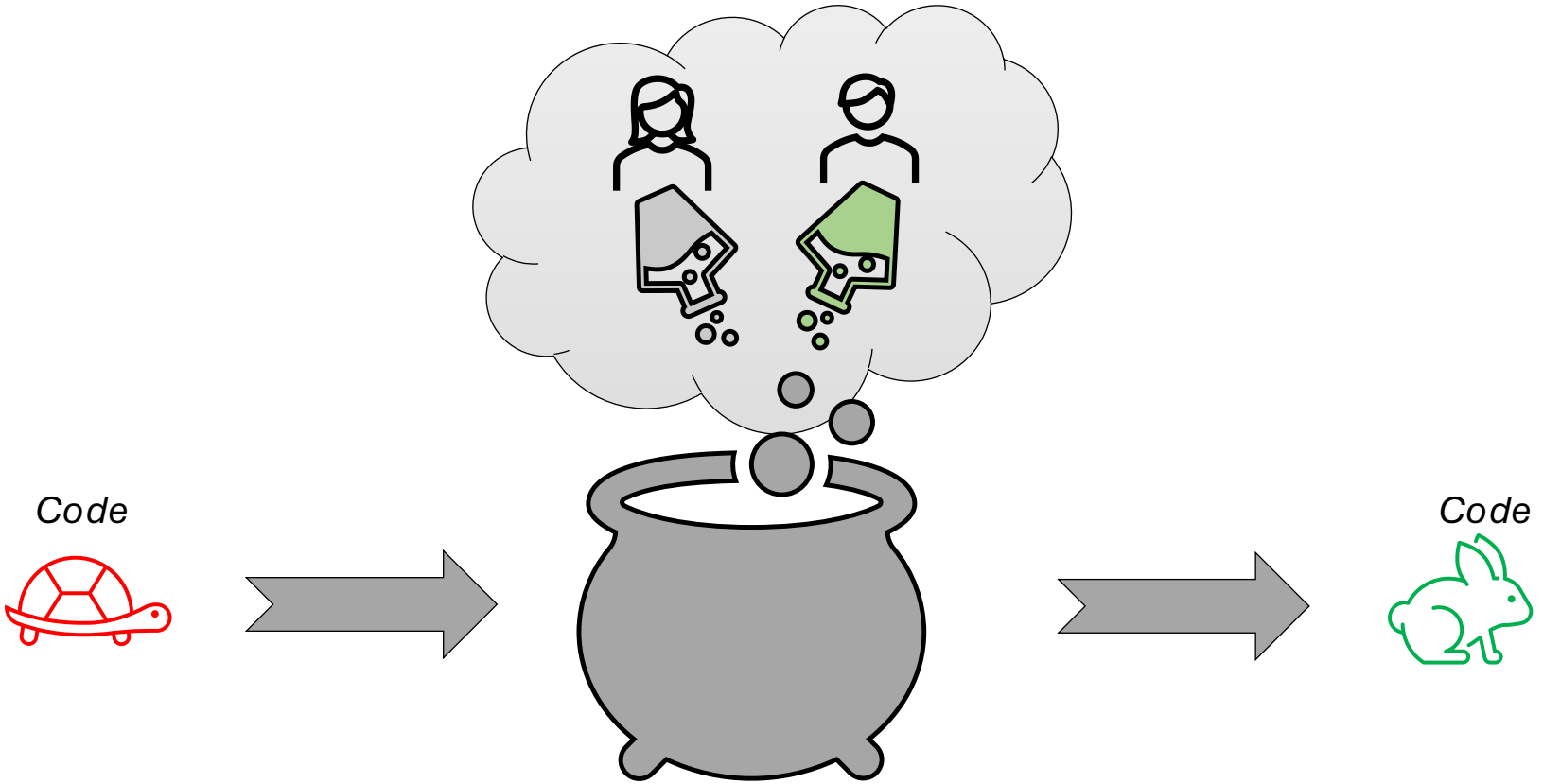
Tuning with Mneme

Overview

1. **Motivation & Problem Statement**
2. Record–Replay Concept
3. Recording & Replay Mechanics
4. Autotuning (Search Spaces & Derivation)
5. Advanced Tuning (Optuna)
6. Deployment with Proteus

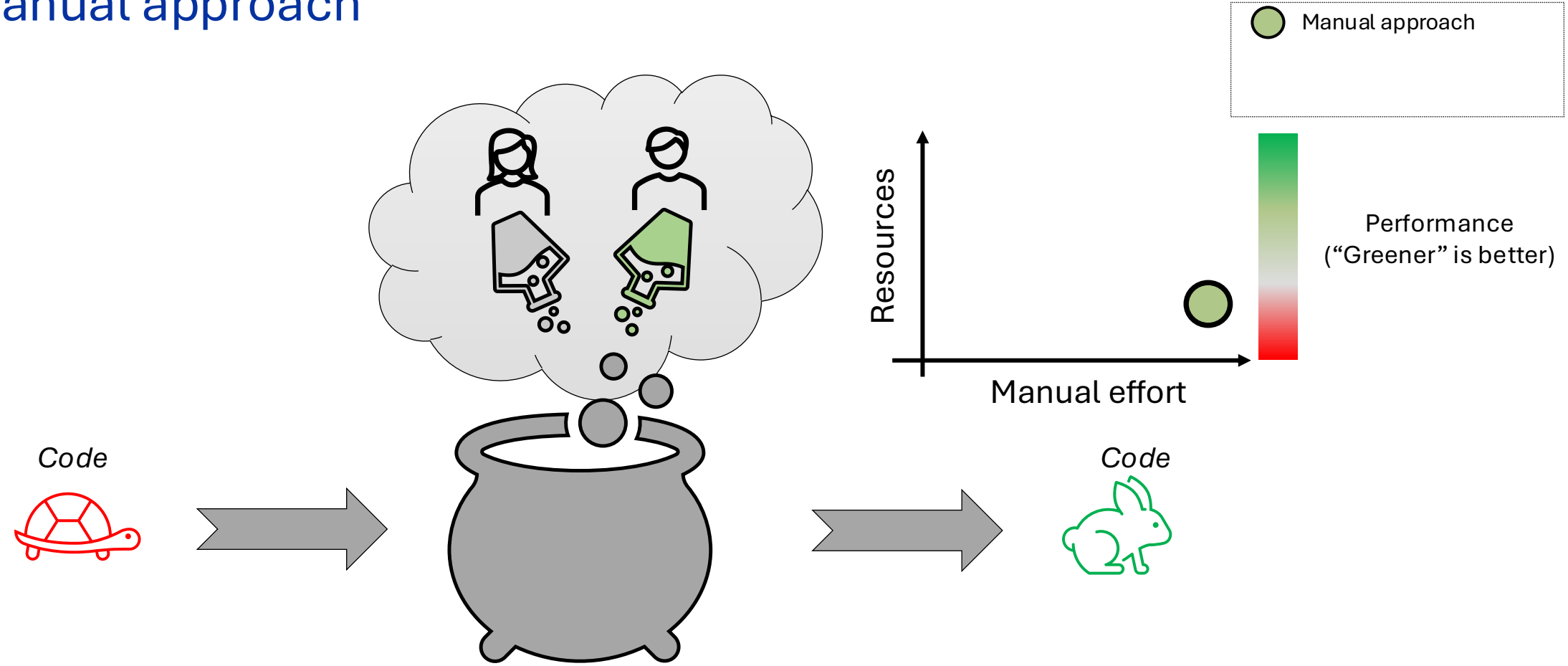
Performance Optimization in a Nutshell

The manual approach



Performance Optimization in a Nutshell

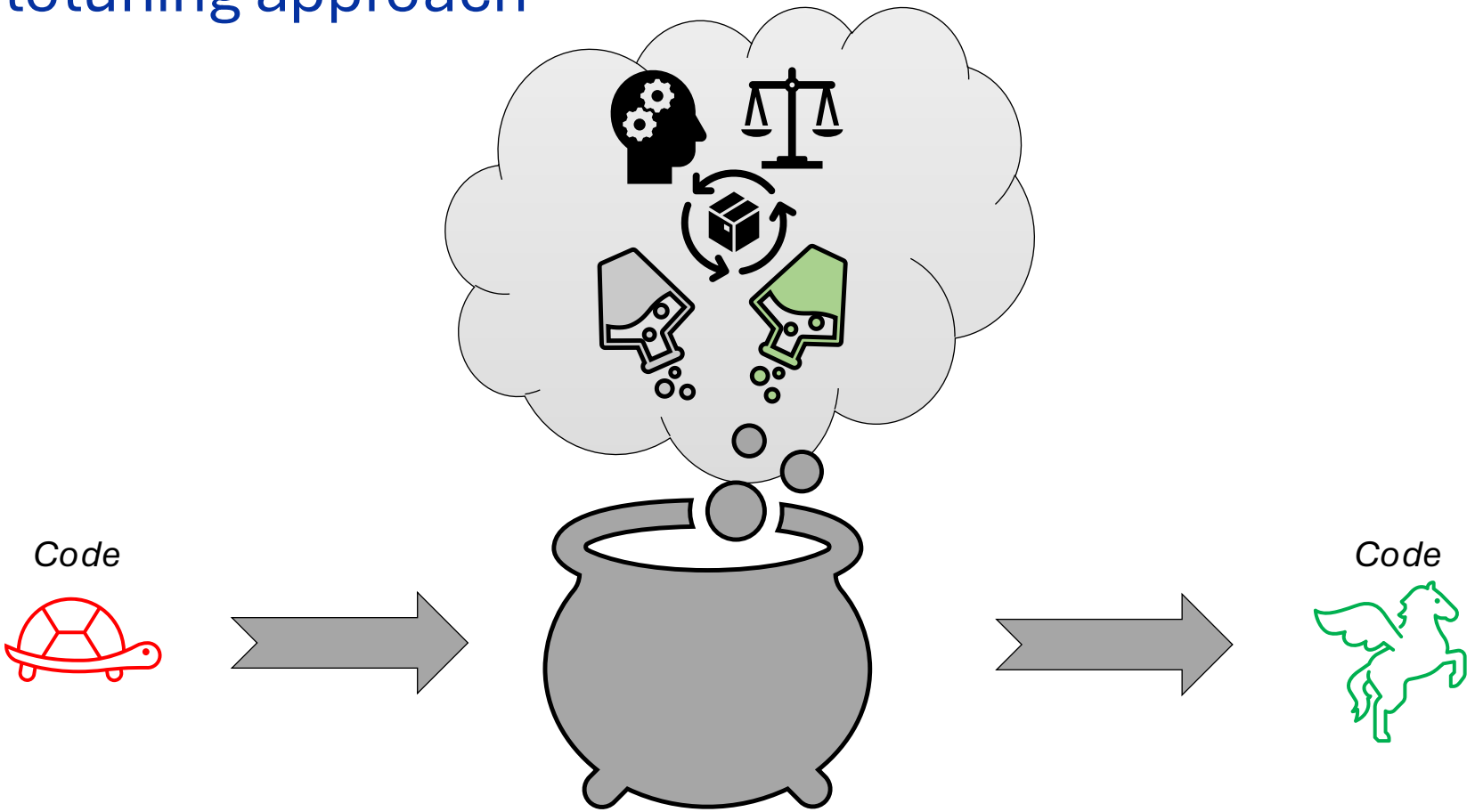
The manual approach



Manually optimizing code requires deep expertise that is hard and expensive to find

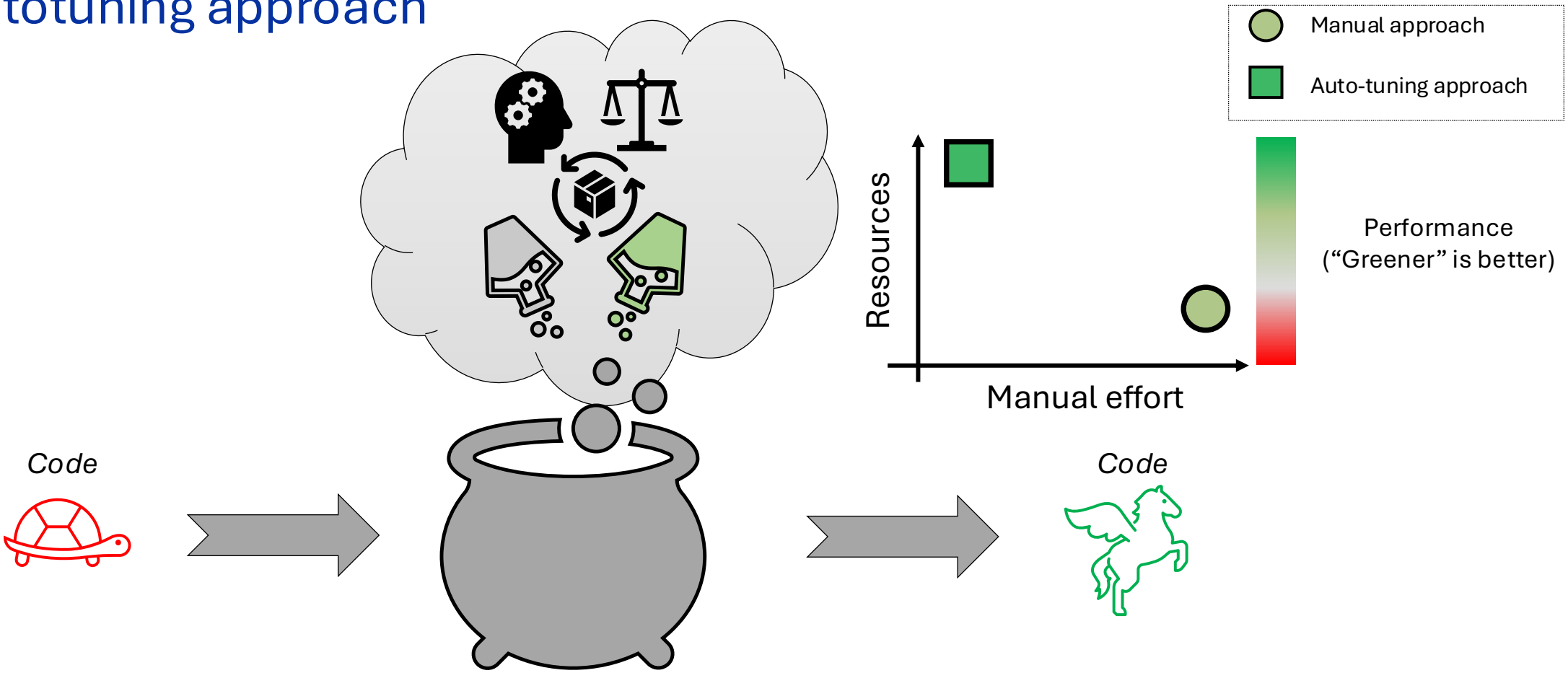
Performance Optimization in a Nutshell

The autotuning approach



Performance Optimization in a Nutshell

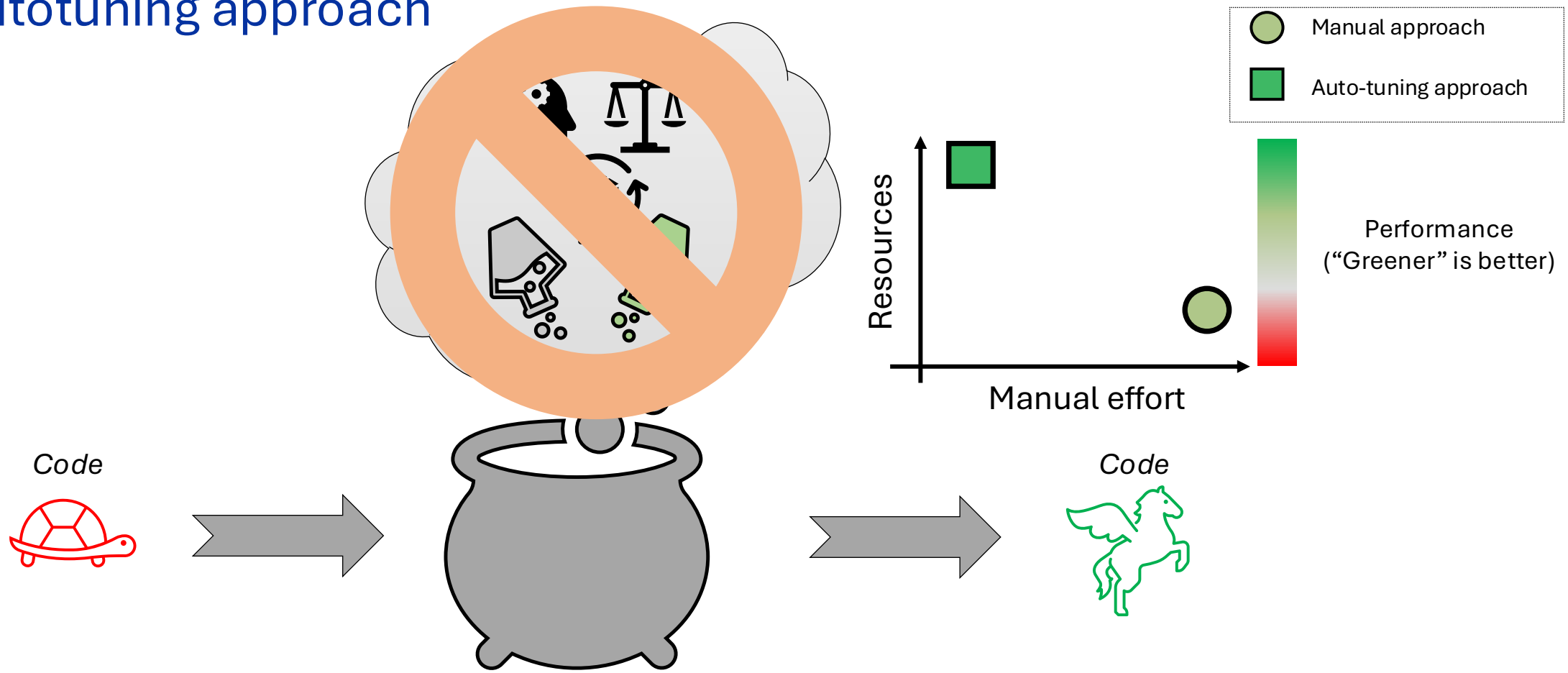
The autotuning approach



There exist many autotuning approaches that optimize codes.

Performance Optimization in a Nutshell

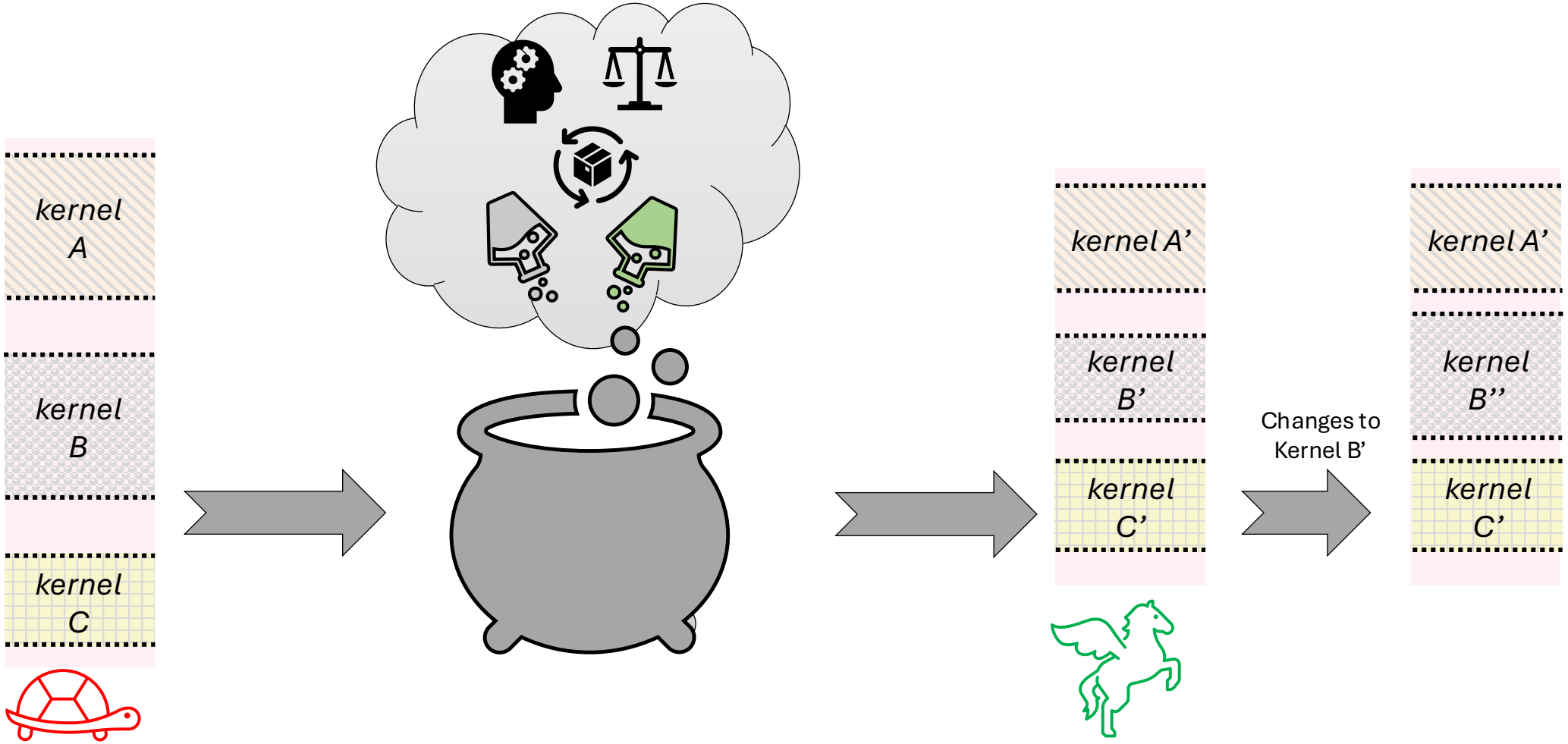
The autotuning approach



Yet, these approaches are impractical and rarely used in large applications as the entire process is impractical

Performance Optimization in a Nutshell

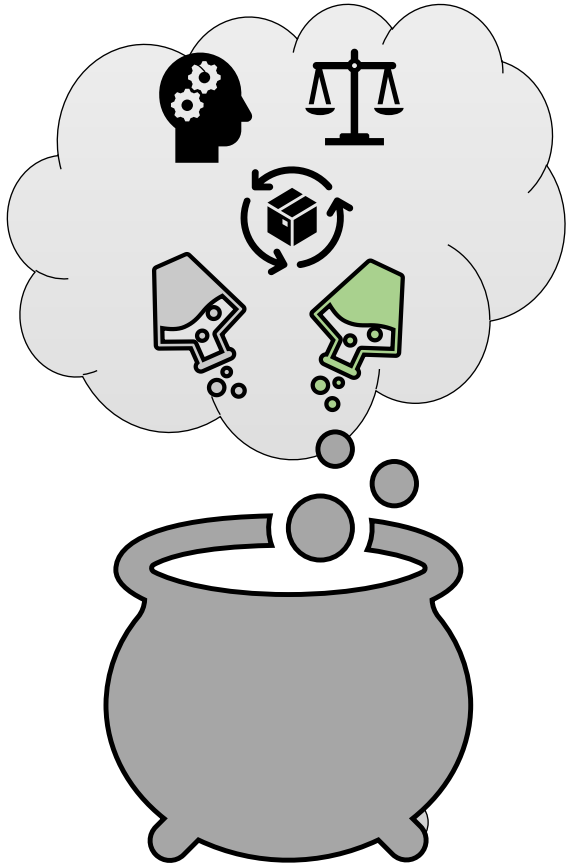
What happens when code is modified?

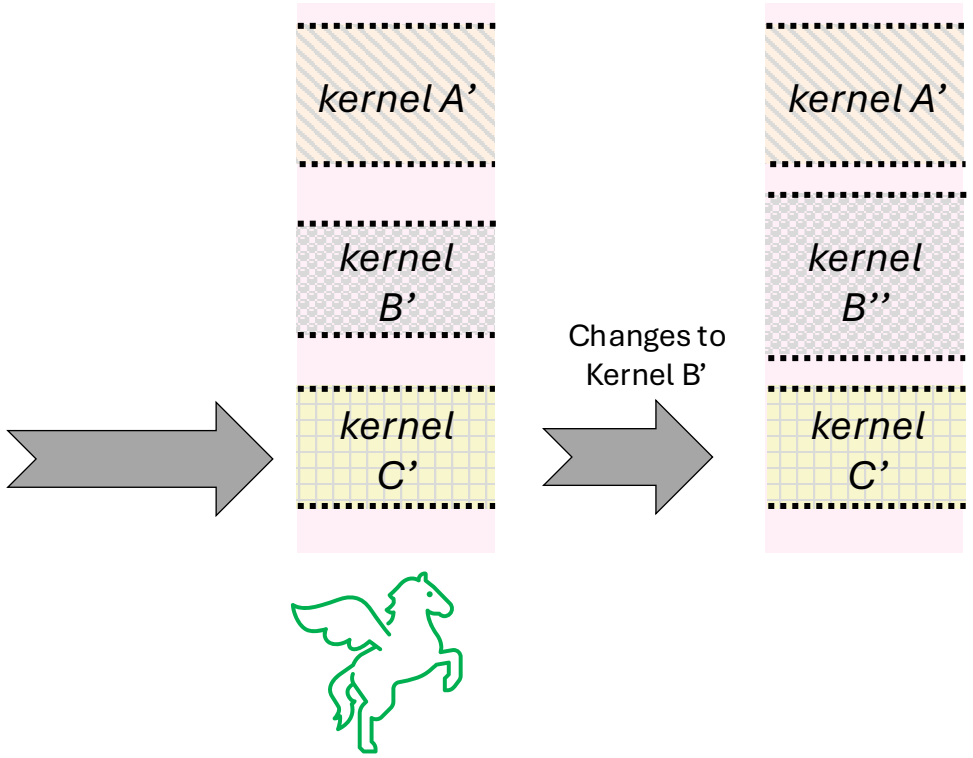


Performance Optimization in a Nutshell

The entire process starts from scratch

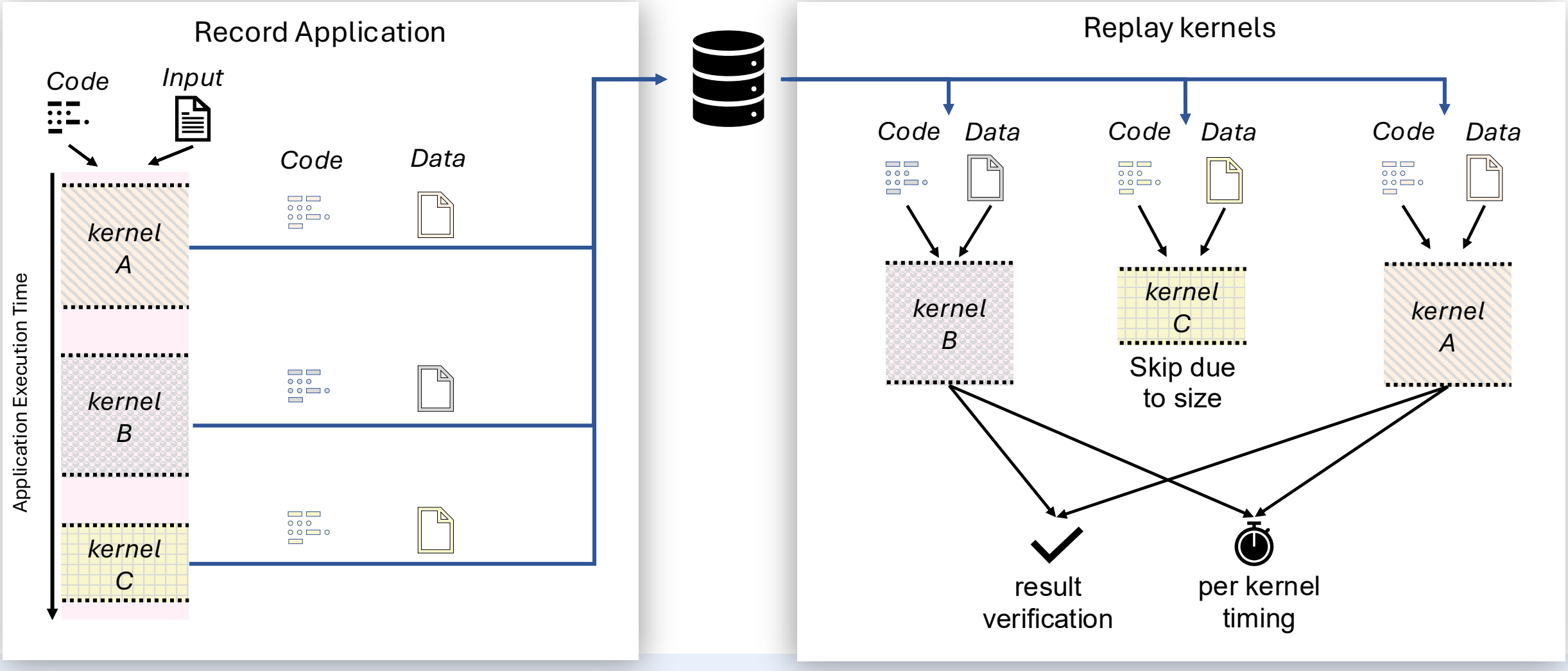
To optimize a **single kernel**, auto-tuners optimize and require the resources of the **entire application**





How do you make autotuning practical?

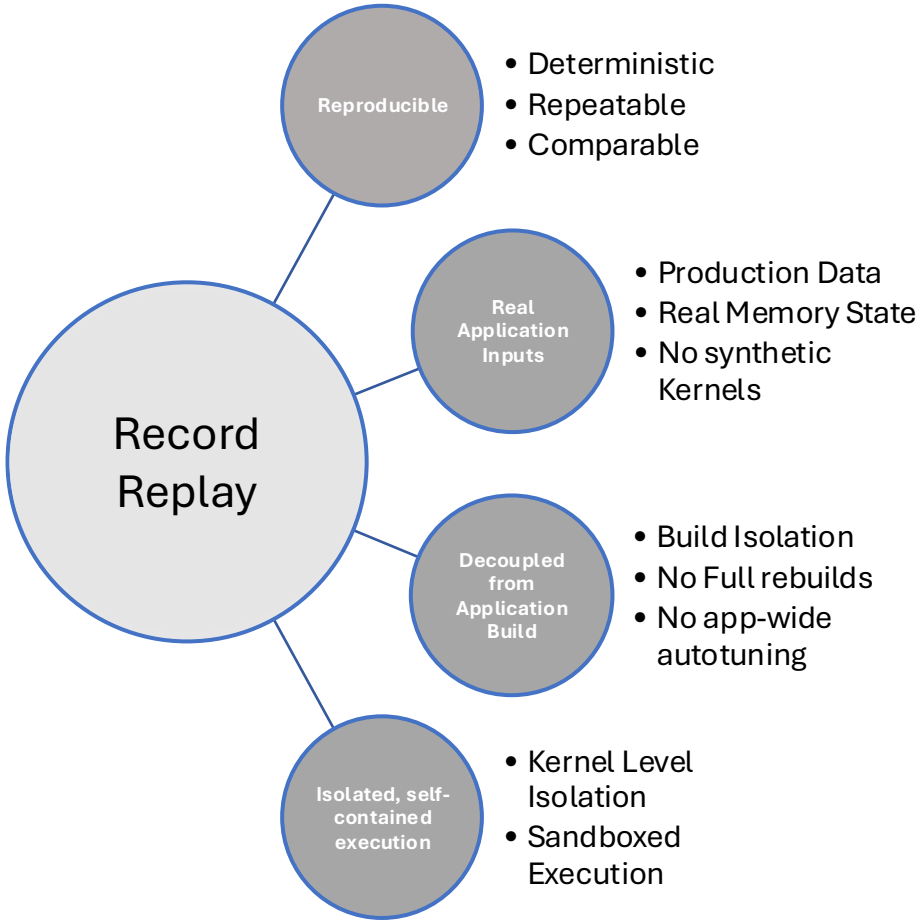
A multi-phase solution based on the concept of record replay



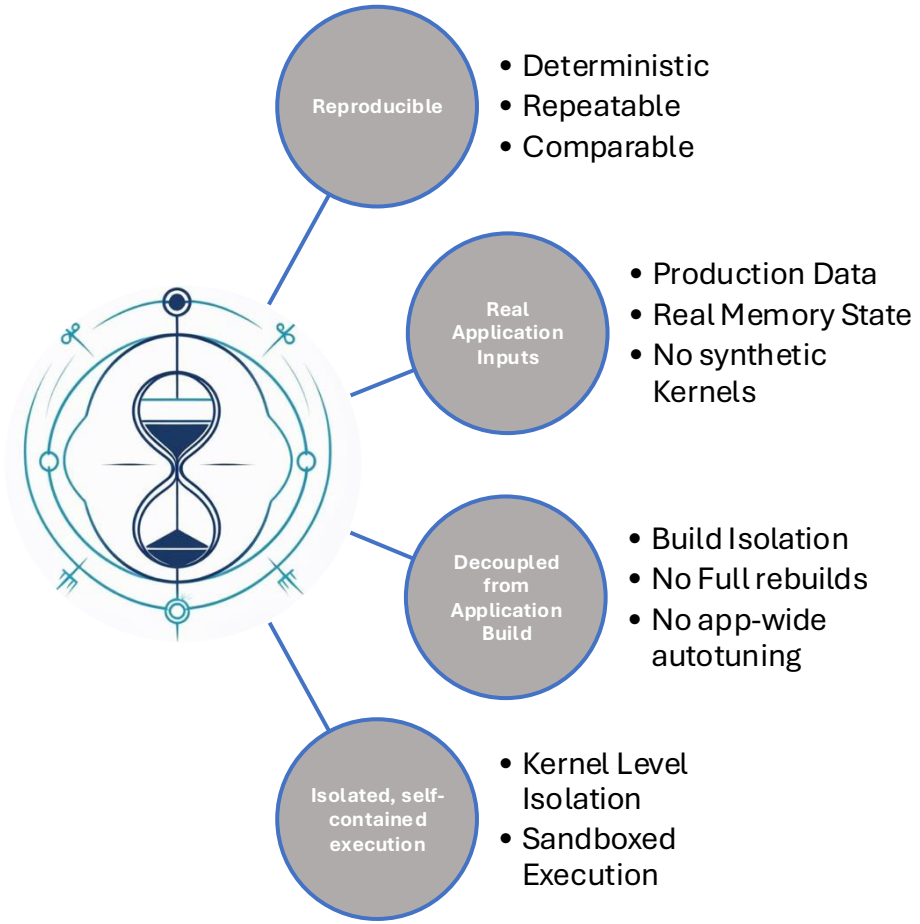
Overview

1. Motivation & Problem Statement
2. **Record–Replay Concept**
3. Recording & Replay Mechanics
4. Autotuning (Search Spaces & Derivation)
5. Advanced Tuning (Optuna)
6. Deployment with Proteus

Record Replay can provide



Mneme: Record–Replay for Scalable Optimization



- Implements record–replay for GPU
- Decouples tuning from application dependencies
- Integrates with LLVM
  - Python accessors to Functions, Blocks, Instructions etc.
  - Proteus is the execution engine and applies optimizations
- Exposes replayed kernels to Python ecosystem
- Enables autotuning, analysis, and experimentation

Making autotuning and compiler experimentation practical

Overview

1. Motivation & Problem Statement
2. Record–Replay Concept
3. **Recording & Replay Mechanics**
4. Autotuning (Search Spaces & Derivation)
5. Advanced Tuning (Optuna)
6. Deployment with Proteus

High Level Of Execution Phases

Build “**Mneme**”

- export LLVM_INSTALL_PATH=${ROCM_PATH}
- `pip install https://github.com/Olympus-HPC/Mneme`

Create a “**recordable executable**”

- Apply instrumentation pass to the code

Record the execution of an application

- Check the generated artifacts

Replay a single Kernel

- Verify outputs
- Create your own autotuner

Build Mneme

```
> export LLVM_INSTALL_PATH=${LLVM_PATH}
> pip install https://github.com/Olympus-HPC/Mneme
```

AMD	Python 3.9	Python 3.10	Python 3.11	Python 3.12
ROCm 6.3 – LLVM 18	✅	✅	✅	✅
ROCm 6.4 – LLVM 19	✅	✅	✅	✅
ROCm 7.1 – LLVM 20	✅	✅	✅	✅

NVIDIA (cuda@12.2)	Python 3.9	Python 3.10	Python 3.11	Python 3.12
LLVM 18	✅	✅	✅	✅
LLVM 19	✅	✅	✅	✅
LLVM 20	✅	✅	✅	✅

On ROCm systems (AMD) LLVM_PATH=${ROCM_PATH}

Create a “**recordable executable**”

1 Include Mneme on build process

```
> cat CMakeLists.txt
...
find_package(HIP REQUIRED)
find_package(mneme REQUIRED)
add_executable(tutorial.exe tutorial.hip)
add_mneme(tutorial.exe)
...
```

2 Configure & Build

```
> cmake -B BUILD -S SRC_PRJ \
          -DCMAKE_C_COMPILER=$(mneme config cc) \
          -DCMAKE_CXX_COMPILER=$(mneme config cxx) \
          -DCMAKE_PREFIX_PATH=$(mneme config cmakedir)
> cmake --build BUILD/
```

The executable carries its own compiler IR

What:

- Embedded LLVM IR

Why it matters:

- Enables **post-mortem analysis and recompilation**
- No need to recover IR from build system or source tree

All kernel executions become observable and interceptable

What:

- Kernel launches go through **Proteus API**
- Vendor launch APIs are not invoked directly

Why it matters:

- Intercept kernel launches, arguments, launch configurations etc.
  - These can be “tunable parameters” at replay time

Record the execution of an application

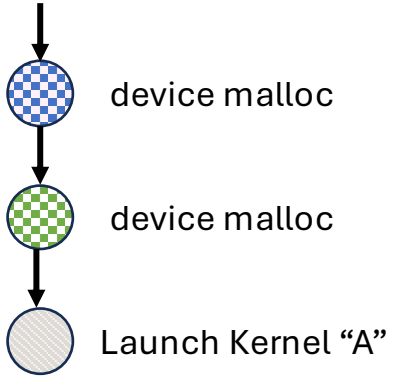
1 Wrap “recordable executable” execution with mneme

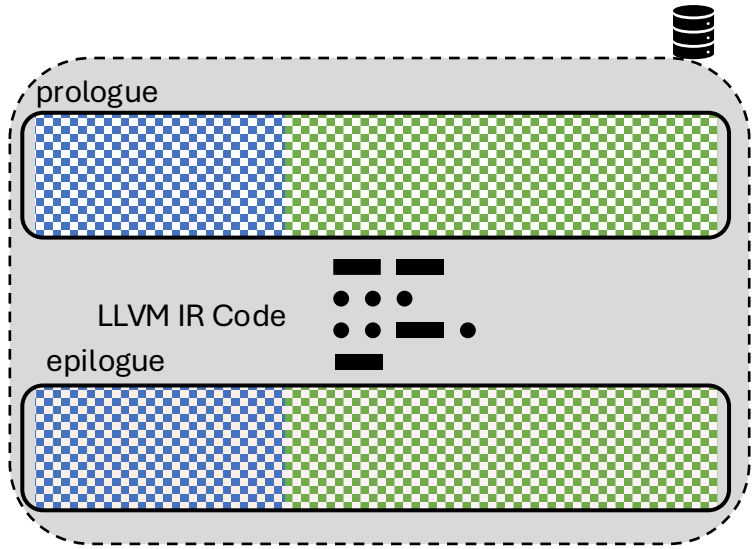
```
> mneme record -rdb record-db-dir/ -vass X \
                -- <recordable-executable> \
                    <arguments>
```

Address Space Managed by Mneme

(|HighAdress – LowAdress| = “vass”)

Trace of host-device events





1) Store Mneme Memory to persistent storage (prologue)

2) Query proteus for LLVM IR of the kernel and store into storage

3) Launch Kernel (synchronously)

4) Store Mneme Memory to persistent storage (epilogue)

Record the execution of an application

1 Wrap “recordable executable” execution with mneme

```
> mneme record -rdb record-db-dir/ -vass X \
               -- <recordable-executable> \
                   <arguments>
```

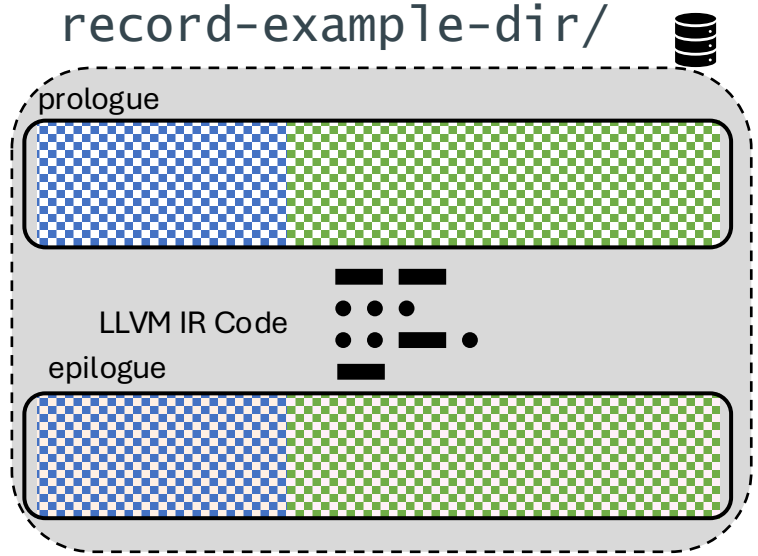
2 Recording artifacts are stored under “record-db-dir”

```
> tree record-db-dir/
── <static-hash>.json
── DeviceState.epilogue.<static-hash>.<dynamic-hash>.mneme
── DeviceState.prologue.<static-hash>.<dynamic-hash>.mneme
── RecordedIR_<static-hash>.bc
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> “default<O3>”
```

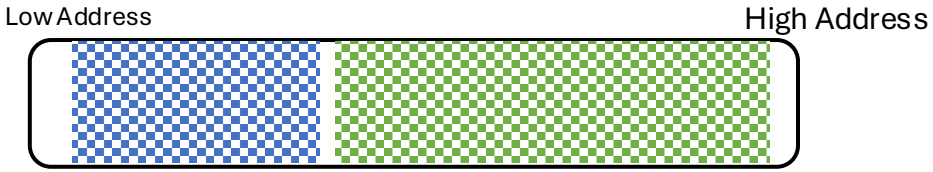


Trace of host-device events

- Instantiate Device Memory Space
- Initialize Memory
- Compile and execute code through Proteus
- Compare device memory with epilogue

Address Space Managed by Mneme

(|HighAdress – LowAdress| = “vass”)



Automated verification

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> “default<O3>”
```

2 Execution emits a key-value dictionary describing various metrics


```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> “default<O3>”
```

These parameters can be modified

By forming valid configuration ranges of these parameters one can search the space and tune the application in respect to some quantity of interest

2 Execution emits a key-value dictionary describing various metrics


```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> “default<O3>”
```

These parameters can be modified

By forming valid configuration ranges of these parameters one can search the space and tune the application in respect to some quantity of interest

Several quantity of interest are supported

- Execution Time (exec_time)
- Register Usage (reg_usage)
- Binary Size (obj_size)

2 Execution emits a key-value dictionary describing various metrics

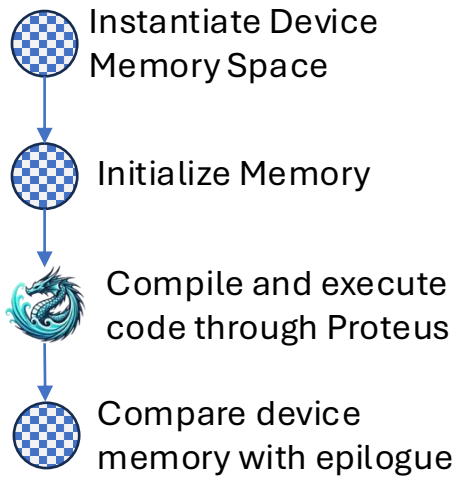

```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

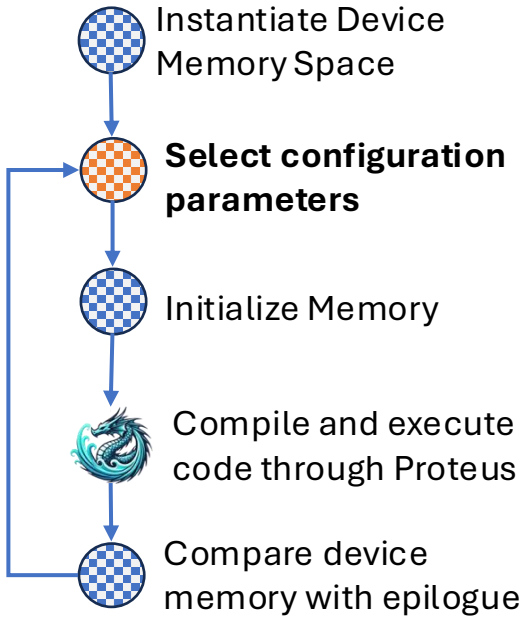
Overview

1. Motivation & Problem Statement
2. Record–Replay Concept
3. Recording & Replay Mechanics
4. **Autotuning (Search Spaces & Derivation)**
5. Advanced Tuning (Optuna)
6. Deployment with Proteus

How do you go from a single replay to autotune?



How do you go from a single replay to autotune?



Mneme Autotuning Concepts

- Mneme users define **search spaces** that describe ranges of possible **parameters**
  - e.g. $ParamBlockDim.x.range \in (0.0, 1.0]$
- A user chosen **Sampler** selects **parameter values**
- Parameters must be derived to **replay configuration points**
  - e.g.: $BlockDim.x = ceil(ParamBlockDim.x.value * 1024)$
- Configuration Points are submitted to a *replay-executor* *and* the *replay-executor* returns *a result*

Define a search space

1. Search Spaces get access to the recorded configuration.
2. The class needs to define a composition of parameters as a space
3. The class is required to override the dimension() function.

```
36 class EntireSpace(SearchSpace):
37     def __init__(self, recorded_kernel: RecordedExecution.KernelInstance):
38         self.grid_dim_x = recorded_kernel.grid_dim.x
39         self.grid_dim_y = recorded_kernel.grid_dim.y
40         self.grid_dim_z = recorded_kernel.grid_dim.z
41
42         self.block_dim_x = recorded_kernel.block_dim.x
43         self.block_dim_y = recorded_kernel.block_dim.y
44         self.block_dim_z = recorded_kernel.block_dim.z
45
46         self.shared_mem = recorded_kernel.shared_mem
47
48         self._search_space = {
49             "specialize": BoolParam("specialize"),
50             "set_launch_bounds": BoolParam("set_launch_bounds"),
51             "specialize_dims": BoolParam("specialize_dims"),
52         }
53
54     def dimensions(self):
55         return self._search_space
```

A user chosen **Sampler** selects **parameter values**

1 Selects a sampling strategy

- ExhaustiveSampling
- OptunaSampingStrategy
  - Parameterized by the study "sampler"
    - TPESampler
    - NSGAIISampler
    - ....

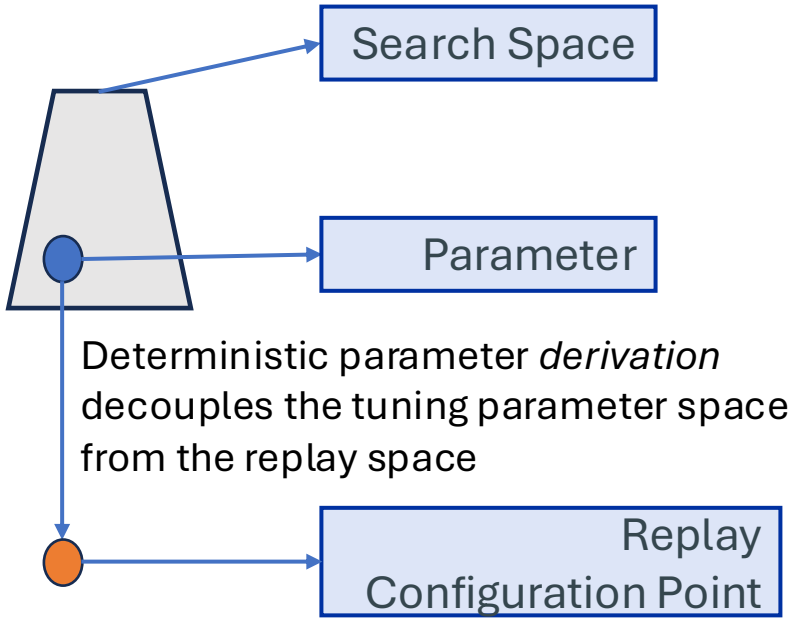
```
150     SS = [ExhaustiveSamplingStrategy(space)]
151     [for i, config in enumerate(SS):]
```

2 Iterate over samples

- Internally we perform a call back to SearchSpace.derived() method

Parameters must be derived to **replay configuration points**

1 Deriving ExperimentConfigurations is extremely powerful




```
57          self._search_space = {
58              "specialize": BoolParam("specialize"),
59              "set_launch_bounds": BoolParam("set_launch_bounds"),
60              "specialize_dims": BoolParam("specialize_dims"),
61          }
```

```
66      def derived(self, params) -> ExperimentConfiguration:
67          derived_config = {
68              "block": {
69                  "x": self.block_dim_x,
70                  "y": self.block_dim_y,
71                  "z": self.block_dim_z,
72              },
73              "grid": {"x": self.grid_dim_x,
74                       "y": self.grid_dim_y,
75                       "z": self.grid_dim_z},
76              "shared_mem": self.shared_mem,
77              "specialize": params["specialize"],
78              "set_launch_bounds": params["set_launch_bounds"],
79              "specialize_dims": params["specialize_dims"],
80          }
81
82          return ExperimentConfiguration.from_dict(derived_config)
```

Configuration Points are submitted to a *replay-executor and* the *replay-executor* returns *a result*


```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```


```
224     executor = AsyncReplayExecutor(
225         record_db=args.record_db,
226         record_id=args.record_id,
227         iterations=5,
228         results_db_dir="./",
229         num_workers=1,
230     )
```

```
158         val = executor.evaluate(config)
```

A simple example with a weather simulation kernel

1 Configure & Build

```
> cmake -B BUILD -S <mneme-repo>/examples/hecbench/ \
          -DCMAKE_C_COMPILER=$(mneme config cc) \
          -DCMAKE_CXX_COMPILER=$(mneme config cxx) \
          -DWITH_MNEME_EXAMPLE_HIP=On \
          -DCMAKE_PREFIX_PATH=$(mneme config cmakedir)
> cmake --build BUILD/
```

2 Wrap wsm5 execution with mneme

```
> mneme record --record-db-dir /var/tmp/wsm5-tutorial/ \
                                        -- ./wsm5/wsm5 1
Average kernel execution time: 598.550455 (ms)
Checksum: rain = 2.759990 snow = 2.759990
```

3 Check generated artifacts

```
> tree /var/tmp/wsm5-tutorial/
/var/tmp/wsm5-tutorial/
├── 2192356271952697806.json
├──DeviceState.epilogue.2192356271952697806.4809330650447
713119.mneme
├──
DeviceState.prologue.2192356271952697806.4809330650447713
119.mneme
└── RecordedIR_2192356271952697806.bc
```

4 Tune the kernel

```
> python ./wsm5/tune.py --record-db \
          /var/tmp/wsm5-tutorial/2192356271952697806.json\
          --record-id 4809330650447713119
Average baseline time 557865817.1428572
...
...

Best config has specialize: True and specilize_dims: True
and set launch bounds: True shows speedup over base line:
1.6735922250391375 and total time: 333334374.28571427
```

Let’s make the configuration space larger

```
59          self._search_space = {
60              "specialize": BoolParam("specialize"),
61              "set_launch_bounds": BoolParam("set_launch_bounds"),
62              "specialize_dims": BoolParam("specialize_dims"),
63              ("passes": CategoricalParam("passes", ["3", "2", "1", "s", "z",],)
64              ),
```

```
78      def derived(self, params) -> ExperimentConfiguration:
79          derived_config = {
80              "block": {
81                  "x": self.block_dim_x,
82                  "y": self.block_dim_y,
83                  "z": self.block_dim_z,
84              },
85              "grid": {"x": self.grid_dim_x, "y": self.grid_dim_y, "z": self.grid_dim_z},
86              "shared_mem": self.shared_mem,
87              "specialize": params["specialize"],
88              "set_launch_bounds": params["set_launch_bounds"],
89              "specialize_dims": params["specialize_dims"],
90              ( "passes": f"default<O{params['passes']}>", )
91          }
92
93          return ExperimentConfiguration.from_dict(derived_config)
```

Let's make the configuration space larger

```
59          self._search_space = {
60              "specialize": BoolParam("specialize"),
61              "set_launch_bounds": BoolParam("set_launch_bounds"),
62              "specialize_dims": BoolParam("specialize_dims"),
63              {"passes": CategoricalParam("passes", ["3", "2", "1", "s", "z", ]),
```

Tune the kernel

```
> python ./wsm5/tune.py --record-db \
          /var/tmp/wsm5-tutorial/2192356271952697806.json\  --record-id 4809330650447713119
Average baseline time 557865817.1428572
...
...

Best config has optimization pipeline default<O1>, specialize: True and specilize_dims: False and
 set launch bounds: True shows speedup over base line : 1.9499789602050337 and total time: 285934005.8571428
```

```
79          derived_config = {
                                                                                    },
90              { "passes": f"default<O{params['passes']}>", }
91          }
92
93          return ExperimentConfiguration.from_dict(derived_config)
```

No need to record or reconfigure the application, just increase the search space and execution time drops from 333ms to 285ms

Overview

1. Motivation & Problem Statement
2. Record–Replay Concept
3. Recording & Replay Mechanics
4. Autotuning (Search Spaces & Derivation)
5. **Advanced Tuning (Optuna)**
6. Deployment with Proteus

Optuna & Continuous Search Spaces

Use miniFE as an example.

- Sparse *MatVec is a:*
  - Grid-stride loop kernel
  - Launch-agnostic kernel
  - Execution-configuration independent kernel
- How can someone tune:
  - Launch Bounds
  - Grid imensions
  - Block Dimensions

```
481 #if defined(MINIFE_CSR_MATRIX)
482   template<typename MatrixType>
483     __global__ void matvec_kernel(const MINIFE_LOCAL_ORDINAL rows_size,
484         const typename MatrixType::LocalOrdinalType *Arowoffsets,
485         const typename MatrixType::GlobalOrdinalType *Acols,
486         const typename MatrixType::ScalarType *Acoefs,
487         const typename MatrixType::ScalarType *xcoefs,
488         typename MatrixType::ScalarType *ycoefs)
489     {
490       MINIFE_LOCAL_ORDINAL stride = blockDim.x * gridDim.x;
491       MINIFE_LOCAL_ORDINAL start = blockIdx.x * blockDim.x + threadIdx.x;
492       for(MINIFE_LOCAL_ORDINAL row = start; row < rows_size; row+=stride) {
493         MINIFE_GLOBAL_ORDINAL row_start = Arowoffsets[row];
494         MINIFE_GLOBAL_ORDINAL row_end   = Arowoffsets[row+1];
495         MINIFE_SCALAR sum = 0;
496
497         // Use the unroll factor in the OpenMP program·
498 #pragma unroll 27
499         for(MINIFE_GLOBAL_ORDINAL i = row_start; i < row_end; ++i) {
500           sum += Acoefs[i] * xcoefs[Acols[i]];
501         }
502         ycoefs[row] = sum;
503       }
504     }
```

Optuna & Continuous Search Spaces

miniFE as an example. How do capture such “conditional” / “constraint” space

- Generate a unit hypercube $[0,1]^n$
  - Every dim in the hypercube represents the range of minimum to maximum valid values

```
35 class EntireSpace(SearchSpace):
36     def __init__(self, recorded_kernel: RecordedExecution.KernelInstance):
37         self.recorded_kernel = recorded_kernel
38         self.original_size = recorded_kernel.grid_dim.x * recorded_kernel.block_dim.x
39         _pos_min_val = sys.float_info.min * sys.float_info.epsilon
40         self._search_space = {
41             "warp_fraction": RealRangeParam("warp_fraction", _pos_min_val, 1.0),
42             "grid_fraction": RealRangeParam("grid_fraction", _pos_min_val, 1.0),
43             "max_threads": RealRangeParam("max_threads", _pos_min_val, 1.0),
44         }
```

Optuna & Continuous Search Spaces

miniFE as an example. Derive fractions to concrete replay values

1. Compute number of threads by computing maximum number of warps * fraction
2. Compute the maximum number of blocks by taking into account the number of threads
3. Do the same for launch bounds. Launch bounds need to be larger or equal to the number of threads and smaller to 1024
4. Map the computed values to the derived config

```
49      def derived(self, params) -> ExperimentConfiguration:
50          # Compute the number of active Warps and map that to the numThreads.
51          maxWarpsInBlock = 1024 / 64
52          numActiveWarps = min(
53              math.ceil(params["warp_fraction"] * maxWarpsInBlock), maxWarpsInBlock
54          )
55          numThreads = int(numActiveWarps * 64)
56
57          # How many blocks exist out of the total size taking into account the new block size
58          maxNumBlocks = int(math.ceil(self.original_size / numThreads))
59
60          # Compute the fraction of that maximum
61          gridDimx = int(
62              min(math.ceil(maxNumBlocks * params["grid_fraction"]), maxNumBlocks)
63          )
64
65          # max threads needs to always be >= numThreads and smaller than 1024.
66          # This is a constraint provided by the vendors.
67          # So here we compute the value:
68          max_threads_range = 1024 - numThreads
69          max_thread_value = numThreads + int(
70              min(math.ceil(max_threads_range * params["max_threads"]), max_threads_range)
71          )
72
73          derived_config = {
74              "block": {"x": numThreads, "y": 1, "z": 1},
75              "grid": {"x": gridDimx, "y": 1, "z": 1},
76              "shared_mem": 0,
77              "specialize": True,
78              "max_threads": max_thread_value,
79              "set_launch_bounds": True,
80              "specialize_dims": True,
81          }
```

Optuna & Continuous Search Spaces

miniFE as an example. Use Optuna Studies to traverse the combinatorial space

1. Create an optuna study. Use any optuna sampler (TPESample is the SOTA optuna sampler)
2. Bind study to Mneme sampler and define number of samples
3. Provide feedback to “optuna” study

```
153     study = optuna.create_study(
154         direction="maximize",
155         sampler=optuna.samplers.TPESampler(),
156     )
157
158     SS = OptunaSamplingStrategy(space, study, 200)
159
160     for i, (config, ctrial) in enumerate(SS):
161         if not config.is_valid():
162             continue
163
164         val = executor.evaluate(config)
165         if val.verified:
166             avg_time = statistics.mean(val.exec_time)
167             speedup = baseline_time / avg_time
168
169             ctrial.set_user_attr("mneme.config", config.to_dict())
170             ctrial.set_user_attr("mneme.result", val.to_dict())
171
172             study.tell(ctrial, speedup)
173             print(
174                 f"\tExperiment {i+1} with options MaxThreads:{config.max_threads}
175                 GridX: {config.grid.x} BlockX: {config.block.x} shows speedup over
176                 base line : {speedup} and total time: {avg_time}"
177             )
178         else:
179             study.tell(ctrial, (1 << 64) - 1)
180             print(i, config.hash(), f"Experiment failed with {val.error}")
```

Optuna & Continuous Search Spaces

miniFE as an example. Execute the tuner

```
Experiment 196 with options MaxThreads:135 GridX:
Experiment 197 with options MaxThreads:198 GridX:
Experiment 198 with options MaxThreads:218 GridX:
Experiment 199 with options MaxThreads:156 GridX:
Experiment 200 with options MaxThreads:83 GridX: 2
Optimal speedup is 1.9270604357800594
Tuning time was 36.69966793060303 to perform 200 samples,
```

- Time to build miniFE:
  - 10s clean build, 4 seconds modifying single source file
- Time to execute miniFE:
  - No Recording : 36 seconds
  - With Recording: 38 seconds
    - This cost is paid once
    - Size the GPU snapshot and speed of IO define slowdown
- Back-of-the-envelope calculation:
  - To run 200 experiments and optimize a single kernel, we would need roughly:
    1. Run MiniFE : 200 * (4 (compile-time) + 7 (number of experiments to reduce noise) * 36) = **51200 seconds = 0.004 observation/second**
    2. Use sub process + standalone replay tool: 38 + 200 * (7 seconds) = **1438 seconds = 0.13 observations/second** ← SC-23
    3. Use python mneme interface (single worker): 38 + 120 (seconds) = **158 seconds = 1.26 observations/second** ← Mneme

Asynchronous Execution With Multiple Workers

```
60          pipelines = self.pipeline_manager.generate(50, 120, 20, True, 0)
61          pipelines = [self.pipeline_manager.to_string(p) for p in pipelines]
62          pipelines += [
63              "default<O3>",
64              "default<O2>",
65              "default<O1>",
66              "default<Os>",
67              "default<Oz>",
68          ]
69
70          self._search_space = {
71              "specialize": BoolParam("specialize"),
72              "set_launch_bounds": BoolParam("set_launch_bounds"),
73              "specialize_dims": BoolParam("specialize_dims"),
74              "passes": CategoricalParam("passes", pipelines),
75          }
```

Asynchronous Execution With Multiple Workers

1. Create the sampling strategy
2. Invoke submit that returns a future
3. Get results (blocking call)

```
167     SS = ExhaustiveSamplingStrategy(space)
168
169     for i, config in enumerate(SS):
170         if not config.is_valid():
171             continue
172         futures.append((config, executor.submit(config)))
173
174     for i, (config, future) in enumerate(futures):
175         val = future.result()
```

Asynchronous Execution With Multiple Workers

```
> time python bezier-surface/tune.py --record-db \
<record-db>.json --record-id <dynamic-hash>
...
real    9m19.579s
user    9m43.129s
sys     8m22.552s
```

```
241     executor = AsyncReplayExecutor(
242         record_db=args.record_db,
243         record_id=args.record_id,
244         iterations=5,
245         results_db_dir="./",
246         num_workers=1,
247     )
```

```
> time python bezier-surface/tune.py --record-db \
<record-db>.json --record-id <dynamic-hash>
...

real    2m30.036s
user    10m13.150s
sys     8m21.479s
```

```
241     executor = AsyncReplayExecutor(
242         record_db=args.record_db,
243         record_id=args.record_id,
244         iterations=5,
245         results_db_dir="./",
246         num_workers=4,
247     )
```

Overview

1. Motivation & Problem Statement
2. Record–Replay Concept
3. Recording & Replay Mechanics
4. Autotuning (Search Spaces & Derivation)
5. Advanced Tuning (Optuna)
6. **Deployment with Proteus**

Serve Mneme configurations to Proteus

```
> ./bezier-surface/bezier-orig -f ../bezier- \
          surface/input/control.txt -n 8192
host execution time: 17372 ms
kernel execution time: 142 ms
PASS
```

Proteus Only (No Cache)

```
> ./bezier-surface/bezier-proteus -f ../bezier- \
          surface/input/control.txt -n 8192
host execution time: 17289 ms
kernel execution time: 119 ms
PASS
```

Proteus + Mneme Tuning (No Cache)

```
> export PROTEUS_TUNED_KERNELS=bezier-tuned.json
> ./bezier-surface/bezier-proteus -f ../bezier- \
          surface/input/control.txt -n 8192
host execution time: 17260 ms
kernel execution time: 120 ms
PASS
```

Proteus Only (With Cache)

```
> ./bezier-surface/bezier-proteus -f ../bezier- \
          surface/input/control.txt -n 8192
host execution time: 17296 ms
kernel execution time: 17 ms
PASS
```

Proteus + Mneme Tuning (With Cache)

```
> export PROTEUS_TUNED_KERNELS=bezier-tuned.json
> ./bezier-surface/bezier-proteus -f ../bezier- \
          surface/input/control.txt -n 8192
host execution time: 17407 ms
kernel execution time: 10 ms
PASS
```

Summary of Covered Components

What did we cover

- Described core concepts of Mneme
- Build instructions
- Basic Usage:
  - Record
  - Replay
  - Tune
    - Search Space
    - Derive Configurations
    - Evaluate Configurations
- Advanced Concepts
  - Composing with Optuna hyperparameter search
  - Multiple workers
  - Asynchronous submission
  - Running Mneme configurations in proteus

Additional capabilities (beyond this tutorial)

- Experiment persistency
  - Check optuna storage + studies using databases
- Experiment Visualization
  - pip install optuna-dashboard
  - optuna-dashboard sqlite:///my_study.db
- Multi Objective optimizations
- Hierarchical Search spaces
- Constraint experiments
- Grounding experiments

Olympus-HPC

Proteus

- Make the compiler Dynamic
  - Empowered by LLVM
  - Hybrid JIT (AoT prepares JIT)
- Specialize code using **runtime knowledge**
  - Arguments
  - Shapes
  - Configs
- Low overhead, cacheable, incrementally intrusive to existing codebases

+

Mneme

- **Record & replay** GPU kernels in isolation
  - Debug capability
  - Autotuning
  - Introspection of LLVM optimizations
- Orders-of-magnitude faster **feedback loops**
  - Optuna empowered Hyperparameter tuning
- Enables large-scale data collection for **ML-guided optimization**

=

Olympus-HPC

- Bridge **compile time ↔ runtime ↔ data**
- Turn expensive end-to-end experiments into **fast inner loops**
- Enable practical ML-driven optimization inside real applications
- Enable deployment of auto tuned kernels with no user interaction

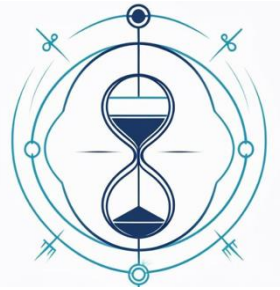
Olympus-HPC turns LLVM into an interactive optimization platform — not a one-shot compiler.

The Team

Giorgis Georgakoudis (PI)

David Beckingsale

Konstantinos Parasyris



John Bowen

Zane Fink

Daniel Nichols

Tal Ben Nun

Thomas Stitt

Tapasya Patki

Loic Pottier

Questions and Feedback

- Reproduce the tutorial examples locally
- Star the repos if you find them useful
- Engage
  - Issues
  - Questions
  - Contributions
- … are more than welcome