EinsumsInCpp

JUSTIN TURNEY

GitHub: github.com/jturney/EinsumsInCpp

What is EinsumsInCpp?

Provides compile-time contraction pattern analysis to determine optimal tensor operation to perform.

```cpp
std::vector<double> C(i * j);
std::vector<double> A(i * k);
std::vector<double> B(k * j);

for (size_t i = 0; i < i0; i++) {
    for (size_t j = 0; j < j0; j++) {
        for (size_t k = 0; k < k0; k++) {
            C[i * j0 + j] +=
                A[i * k0 + k] * B[k * j0 + j];
        }
    }
}
```

→

```cpp
Tensor A{"A", i_, k_};
Tensor B{"B", k_, j_};
Tensor C{"C", i_, j_};

einsum(Indices{i, j}, &C,
       Indices{i, k}, A,
       Indices{k, j}, B);
```

Compile-Time Deduction

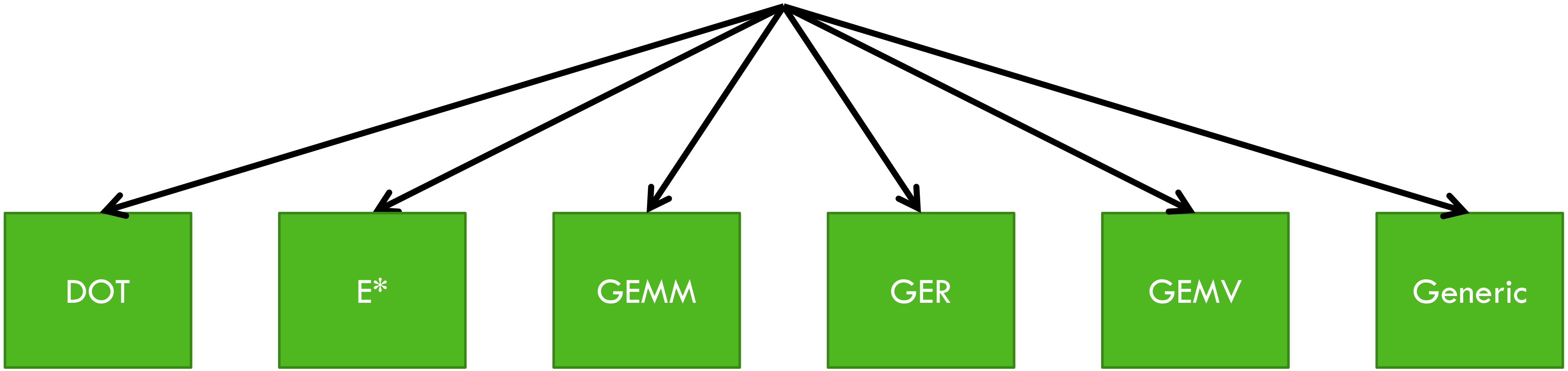
```
std::vector<double> C(i * j);
std::vector<double> A(i * k);
std::vector<double> B(k * j);

for (size_t i = 0; i < i0; i++) {
    for (size_t j = 0; j < j0; j++) {
        for (size_t k = 0; k < k0; k++) {
            C[i * j0 + j] +=
                A[i * k0 + k] * B[k * j0 + j];
        }
    }
}
```

```
Tensor A{"A", i_, k_};
Tensor B{"B", k_, j_};
Tensor C{"C", i_, j_};

einsum(Indices{i, j}, &C,
       Indices{i, k}, A,
       Indices{k, j}, B);
```

GEMM

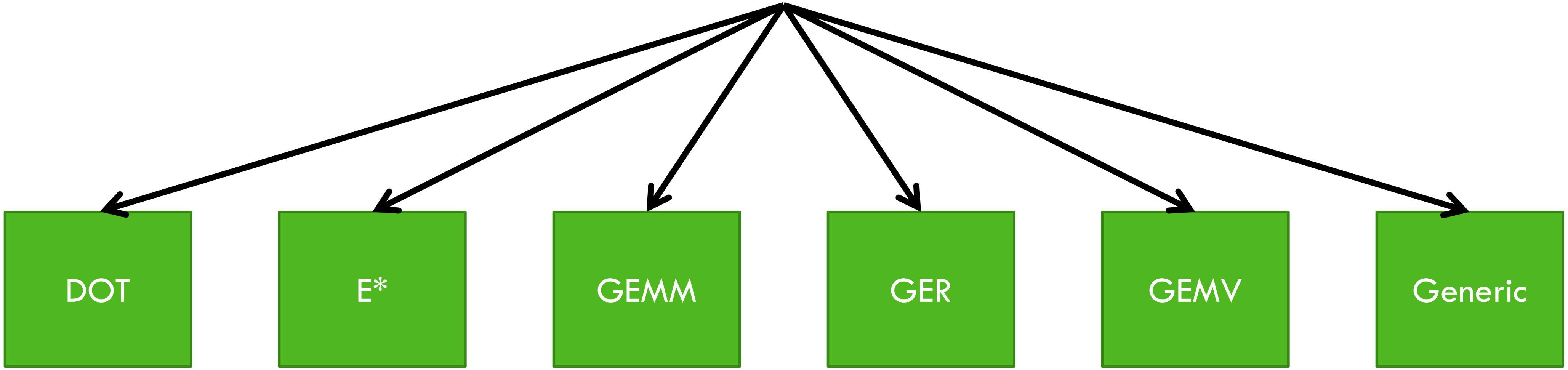
Compile-Time Deduction

```
Tensor A{"A", ....};
Tensor B{"B", ....};
Tensor C{"C", ....};

einsum(Indices{....}, &C,
       Indices{....}, A,
       Indices{....}, B);
```



Compile-Time Deduction

```
Tensor A{"A", ....};
Tensor B{"B", ....};
Tensor C{"C", ....};

einsum(CPrefactor, Indices{....}, &C,
       ABPrefactor,
       Indices{....}, A, Indices{....}, B);
```



How does it work?

```
Tensor A{"A", ....};
Tensor B{"B", ....};
Tensor C{"C", ....};

einsum(Indices{....}, &C,
       Indices{....}, A,
       Indices{....}, B);
```

```
template <typename AType, typename BType, typename CType, typename... CIndices, typename... AIndices,
          typename... BIndices>
auto einsum(const std::tuple<CIndices...> &C_indices, CType *C,
            const std::tuple<AIndices...> &A_indices, const AType &A,
            const std::tuple<BIndices...> &B_indices, const BType &B)
    -> std::enable_if_t<!is_smart_pointer_v<CType> && !is_smart_pointer_v<AType>
                        && !is_smart_pointer_v<BType>> {
    einsum(0, C_indices, C, 1, A_indices, A, B_indices, B);
}
```

How does it work?

```
{
    einsum(0, C_indices, C, 1, A_indices, A, B_indices, B);
}
```

```
template <template <typename, size_t> typename AType, typename ADataType, size_t ARank,
          template <typename, size_t> typename BType, typename BDataType, size_t BRank,
          template <typename, size_t> typename CType, typename CDataType, size_t CRank,
          typename... CIndices, typename... AIndices, typename... BIndices>
auto einsum(const CDataType C_prefactor, const std::tuple<CIndices...>&, CType<CDataType, CRank> *C,
            const std::conditional_t<(sizeof(ADataType) > sizeof(BDataType)), ADataType, BDataType> AB_prefactor,
            const std::tuple<AIndices...> &, const AType<ADataType, ARank> &A,
            const std::tuple<BIndices...> &, const BType<BDataType, BRank> &B)
    -> std::enable_if_t<
            std::is_base_of_v<::einsums::detail::TensorBase<ADataType, ARank>, AType<ADataType, ARank>> &&
            std::is_base_of_v<::einsums::detail::TensorBase<BDataType, BRank>, BType<BDataType, BRank>> &&
            std::is_base_of_v<::einsums::detail::TensorBase<CDataType, CRank>, CType<CDataType, CRank>>>
{
```

How does it work?

It then performs a series of compile-time checks on the provided indices:

- Uniqueness
- Hadamard indices
- Index positions
- Contiguous indices
- And others.

```cpp
constexpr auto A_indices = std::tuple<AIndices...>();
constexpr auto B_indices = std::tuple<BIndices...>();
constexpr auto C_indices = std::tuple<CIndices...>();
using ABDataType = std::conditional_t<(sizeof(ADataType) > sizeof(BDataType)), ADataType, BDataType>;

// 1. Ensure the ranks are correct. (Compile-time check.)
static_assert(sizeof...(CIndices) == CRank, "Rank of C does not match Indices given for C.");
static_assert(sizeof...(AIndices) == ARank, "Rank of A does not match Indices given for A.");
static_assert(sizeof...(BIndices) == BRank, "Rank of B does not match Indices given for B.");

// 2. Determine the links from AIndices and BIndices
constexpr auto linksAB = intersect_t<std::tuple<AIndices...>, std::tuple<BIndices...>>();
// 2a. Remove any links that appear in the target
constexpr auto links = difference_t<decltype(linksAB), std::tuple<CIndices...>>();

// 3. Determine the links between CIndices and AIndices
constexpr auto CAlinks = intersect_t<std::tuple<CIndices...>, std::tuple<AIndices...>>();

// 4. Determine the links between CIndices and BIndices
constexpr auto CBlinks = intersect_t<std::tuple<CIndices...>, std::tuple<BIndices...>>();

// Remove anything from A that exists in C
constexpr auto CminusA = difference_t<std::tuple<CIndices...>, std::tuple<AIndices...>>();
constexpr auto CminusB = difference_t<std::tuple<CIndices...>, std::tuple<BIndices...>>();

constexpr bool have_remaining_indices_in_CminusA = std::tuple_size_v<decltype(CminusA)>;
constexpr bool have_remaining_indices_in_CminusB = std::tuple_size_v<decltype(CminusB)>;

// Determine unique indices in A
constexpr auto A_only = difference_t<std::tuple<AIndices...>, decltype(links)>();
constexpr auto B_only = difference_t<std::tuple<BIndices...>, decltype(links)>();

constexpr auto A_unique = unique_t<std::tuple<AIndices...>>();
constexpr auto B_unique = unique_t<std::tuple<BIndices...>>();
constexpr auto C_unique = unique_t<std::tuple<CIndices...>>();
constexpr auto link_unique = c_unique_t<decltype(links)>();

constexpr bool A_hadamard_found = std::tuple_size_v<std::tuple<AIndices...>> != std::tuple_size_v<decltype(A_unique)>;
constexpr bool B_hadamard_found = std::tuple_size_v<std::tuple<BIndices...>> != std::tuple_size_v<decltype(B_unique)>;
constexpr bool C_hadamard_found = std::tuple_size_v<std::tuple<CIndices...>> != std::tuple_size_v<decltype(C_unique)>;

constexpr auto link_position_in_A = detail::find_type_with_position(link_unique, A_indices);
constexpr auto link_position_in_B = detail::find_type_with_position(link_unique, B_indices);
constexpr auto link_position_in_link = detail::find_type_with_position(link_unique, links);

constexpr auto target_position_in_A = detail::find_type_with_position(C_unique, A_indices);
constexpr auto target_position_in_B = detail::find_type_with_position(C_unique, B_indices);
constexpr auto target_position_in_C = detail::find_type_with_position(C_unique, C_indices);

constexpr auto A_target_position_in_C = detail::find_type_with_position(A_indices, C_indices);
constexpr auto B_target_position_in_C = detail::find_type_with_position(B_indices, C_indices);

auto unique_target_dims = detail::get_dim_ranges_for(*C, detail::unique_find_type_with_position(C_unique, C_indices));
auto unique_link_dims = detail::get_dim_ranges_for(A, link_position_in_A);

constexpr auto contiguous_link_position_in_A = detail::contiguous_positions(link_position_in_A);
constexpr auto contiguous_link_position_in_B = detail::contiguous_positions(link_position_in_B);

constexpr auto contiguous_target_position_in_A = detail::contiguous_positions(target_position_in_A);
constexpr auto contiguous_target_position_in_B = detail::contiguous_positions(target_position_in_B);

constexpr auto contiguous_A_targets_in_C = detail::contiguous_positions(A_target_position_in_C);
constexpr auto contiguous_B_targets_in_C = detail::contiguous_positions(B_target_position_in_C);

constexpr auto same_ordering_link_position_in_AB = detail::is_same_ordering(link_position_in_A, link_position_in_B);
constexpr auto same_ordering_target_position_in_CA = detail::is_same_ordering(target_position_in_A, A_target_position_in_C);
constexpr auto same_ordering_target_position_in_CB = detail::is_same_ordering(target_position_in_B, B_target_position_in_C);

constexpr auto C_exactly_matches_A =
    sizeof...(CIndices) == sizeof...(AIndices) && same_indices<std::tuple<CIndices...>, std::tuple<AIndices...>>();
constexpr auto C_exactly_matches_B =
    sizeof...(CIndices) == sizeof...(BIndices) && same_indices<std::tuple<CIndices...>, std::tuple<BIndices...>>();
constexpr auto A_exactly_matches_B = same_indices<std::tuple<AIndices...>, std::tuple<BIndices...>>();
```

How does it work?

```
// 2. Determine the links from AIndices and BIndices
constexpr auto linksAB = intersect_t<std::tuple<AIndices...>, std::tuple<BIndices...>>();
// 2a. Remove any links that appear in the target
constexpr auto links = difference_t<decltype(linksAB), std::tuple<CIndices...>>();

// 3. Determine the links between CIndices and AIndices
constexpr auto CAlinks = intersect_t<std::tuple<CIndices...>, std::tuple<AIndices...>>();

// 4. Determine the links between CIndices and BIndices
constexpr auto CBlinks = intersect_t<std::tuple<CIndices...>, std::tuple<BIndices...>>();

// Remove anything from A that exists in C
constexpr auto CminusA = difference_t<std::tuple<CIndices...>, std::tuple<AIndices...>>();
constexpr auto CminusB = difference_t<std::tuple<CIndices...>, std::tuple<BIndices...>>();

constexpr bool have_remaining_indices_in_CminusA = std::tuple_size_v<decltype(CminusA)>;
constexpr bool have_remaining_indices_in_CminusB = std::tuple_size_v<decltype(CminusB)>;
```

How does it work?

```
constexpr auto is_gemm_possible = have_remaining_indices_in_CminusA && have_remaining_indices_in_CminusB &&
                                  contiguous_link_position_in_A && contiguous_link_position_in_B && contiguous_target_position_in_A &&
                                  contiguous_target_position_in_B && contiguous_A_targets_in_C && contiguous_B_targets_in_C &&
                                  same_ordering_link_position_in_AB && same_ordering_target_position_in_CA &&
                                  same_ordering_target_position_in_CB && !A_hadamard_found && !B_hadamard_found && !C_hadamard_found;
constexpr auto is_gemv_possible = contiguous_link_position_in_A && contiguous_link_position_in_B && contiguous_target_position_in_A &&
                                  same_ordering_link_position_in_AB && same_ordering_target_position_in_CA &&
                                  !same_ordering_target_position_in_CB && std::tuple_size_v<decltype(B_target_position_in_C)> == 0 &&
                                  !A_hadamard_found && !B_hadamard_found && !C_hadamard_found;

constexpr auto element_wise_multiplication =
    C_exactly_matches_A && C_exactly_matches_B && !A_hadamard_found && !B_hadamard_found && !C_hadamard_found;
constexpr auto dot_product =
    sizeof...(CIndices) == 0 && A_exactly_matches_B && !A_hadamard_found && !B_hadamard_found && !C_hadamard_found;

constexpr auto outer_product = std::tuple_size_v<decltype(linksAB)> == 0 && contiguous_target_position_in_A &&
                               contiguous_target_position_in_B && !A_hadamard_found && !B_hadamard_found && !C_hadamard_found;
```

How does it work?

```
if constexpr (dot_product) {
    CDataType temp = linear_algebra::dot(A, B);
    (*C) *= C_prefactor;
    (*C) += AB_prefactor * temp;

    return;
}
```

How does it work?

```
    else if constexpr (element_wise_multiplication) {
        timer::Timer element_wise_multiplication{"element-wise multiplication"};

        auto target_dims = get_dim_ranges<CRank>(*C);
        auto view = std::apply(ranges::views::cartesian_product, target_dims);

        // Ensure the various tensors passed in are the same dimensionality
        if (((C->dims() != A.dims()) || C->dims() != B.dims())) {
            println_abort("einsum: at least one tensor does not have same dimensionality as destination");
        }

#if defined(__INTEL_LLVM_COMPILER) || defined(__INTEL_COMPILER)
#pragma omp parallel for simd
#else
#pragma omp parallel for
#endif
        for (auto it = view.begin(); it != view.end(); it++) {
            CDataType &target_value = std::apply(*C, *it);
            ABDataType AB_product = std::apply(A, *it) * std::apply(B, *it);
            target_value = C_prefactor * target_value + AB_prefactor * AB_product;
        }

        return;
    }
```

How does it work?

If none of the patterns programmed can be utilized, then the code with use a generic, compile-time, threaded contraction code.

It’s still better to work the equations or the tensors to match a better performing pattern.

```
einsum(1.0, Indices{p, q}, F,  1.0, Indices{p, q, r, s}, g, Indices{r, s}, D);
einsum(1.0, Indices{p, q}, F, -1.0, Indices{p, r, q, s}, g, Indices{r, s}, D);
```

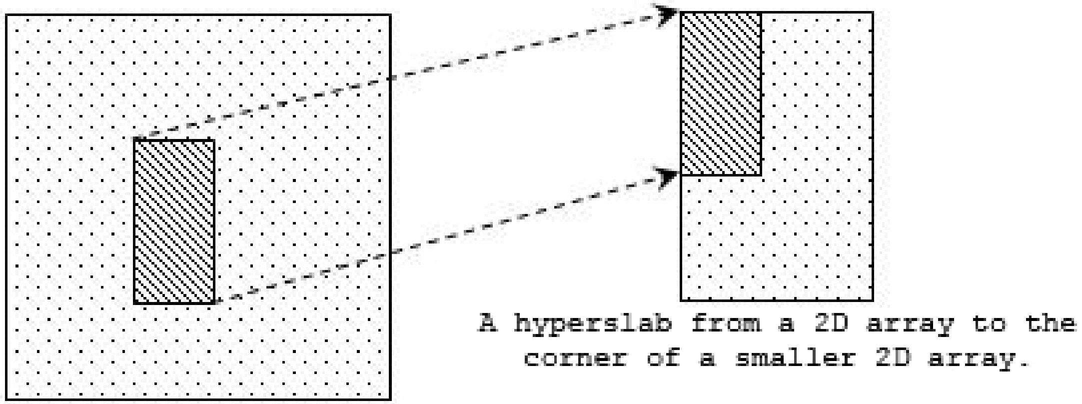
Sorting

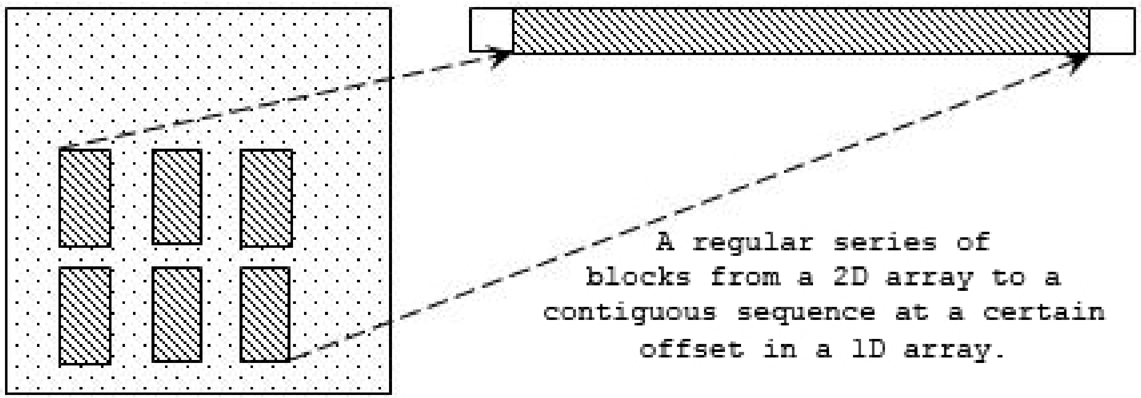
```
sort(Indices{k, j, i}, &B,
     Indices{i, j, k}, A);

sort(0.0, Indices{i, l, k, j}, &B,
     0.5, Indices{k, j, l, i}, A);
```

Disk Tensors

EinsumsInCpp uses HDF5 library to store tensors to disk. It utilizes the h5cpp wrapper library to enable automatic tensor loading and storing to an HDF5 file.

HDF5 stores data into a hierarchal structure (think a file system within a file).





Disk Tensors

Storing the tensor properly on disk can be vitally important. EinsumsInCpp uses HDF5 which is sensitive to how the data is “chunked” on disk.

```
Timing information:

       0 ms :  1000 calls :     0 ms per call                          :     Timer Overhead
    9206 ms :     1 calls :  9206 ms per call                          :     Creating random tensor 64 64 64 64
       0 ms :     1 calls :     0 ms per call                          :     Creating disk tensor
    2499 ms :     1 calls :  2499 ms per call                          :     disk write (everything at once)
    2401 ms :     1 calls :  2401 ms per call                          :     disk read (equivalent to chunking)
    2966 ms :     1 calls :  2966 ms per call                          :     disk read (different to chunking)
       0 ms :     1 calls :     0 ms per call                          :     Creating disk tensor2
    2605 ms :     1 calls :  2605 ms per call                          :     disk write2 non-default chunk (everything at once)
    2463 ms :     1 calls :  2463 ms per call                          :     disk read2 (equivalent to chunking)
    2938 ms :     1 calls :  2938 ms per call                          :     disk read2 (different to chunking)
       0 ms :     1 calls :     0 ms per call                          :     Creating disk tensor3 bad chunking
   17620 ms :     1 calls : 17620 ms per call                          :     disk write3 (everything at once)
   49799 ms :     1 calls : 49799 ms per call                          :     disk read3 (different to chunking)
```

Disk Tensors

```
timer::push("Forming energy denominator (E_ijab)");
DiskTensor<double, 4> e_oovv{state::data,
    "/Method/Spin-Orbital/CCD/Energy Denominator oovv",
    nocc, nocc, nvir, nvir};

for (size_t i0 = 0; i0 < nocc; i0++) {
    for (size_t j0 = 0; j0 < nocc; j0++) {
        auto e_view = e_oovv(i0, j0, All, All);

        double e_ij = eocc(i0) + eocc(j0);
        for (size_t a0 = 0; a0 < nvir; a0++) {
            for (size_t b0 = 0; b0 < nvir; b0++) {
                e_view(a0, b0) =
                    1.0 / (e_ij - evir(a0) - evir(b0));
            }
        }
    }
}
timer::pop();
```

```
// Compute MP2 from the antisymmetrized integrals
timer::push("MP2 Check");
double e_mp2{0.0};
for (size_t i0 = 0; i0 < nocc; i0++) {
    for (size_t j0 = 0; j0 < nocc; j0++) {
        auto e_ij = e_oovv(i0, j0, All, All);
        auto g_ij = g_oovv(i0, j0, All, All);

        e_mp2 += linear_algebra::dot(g_ij.get(),
                                     g_ij.get(),
                                     e_ij.get());

    }
}
e_mp2 /= 4.0;
timer::pop();
```

Other Features

Perform contractions on any singular datatype and mixed datatypes.

Tensor Decompositions – Andy Jiang

- Parafac
- Tucker

Elemental Operations

- e.g.: abs, min, max, exp

Khatri-Rao Product

$$\left(A_{ij} \otimes B_{ij}\right)_{ij}$$

Tensor Unfolding

$$A_{ijk} \rightarrow A_{i,\{jk\}}$$

$$A_{ijk} \rightarrow A_{j,\{ik\}}$$

Lyapunov Solver – Andy Jiang

$$AX + XA^T + Q = 0$$

Truncated SVD – Andy Jiang

Initial Implementation of Polynomials

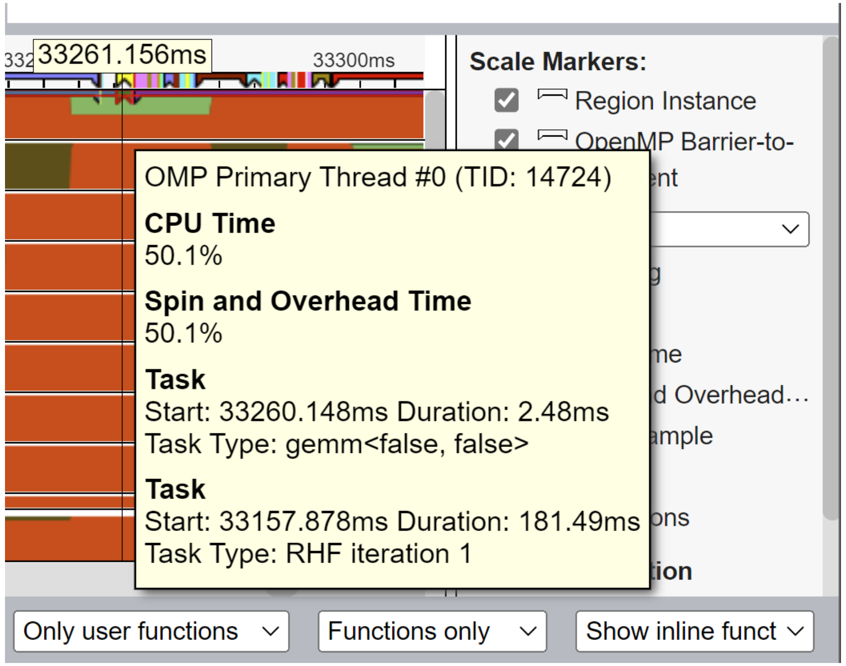
- Gauss-Laguerre

Plays Nice with VTune

If VTune is detected at CMake configure-time, the "Section" class will register with VTune to provide information on where in your code things are happening.

Instructions for running VTune in the EinsumsInCpp Docker container can be found here:

github.com/jturney/EinsumsInCpp/tree/main/.devcontainer




```
{
    Section section(fmt::format("gemm<{}, {}>", TransA, TransB));
    blas::gemm(...);
}
```

Where is it being used?

Tensor Hypercontraction Form of (T) Energy in Coupled-Cluster Theory – Andy Jiang

https://arxiv.org/abs/2210.07035

F12 Methods – Erica Mitchell

Future Plans

- Point group symmetry
- Identify new einsum patterns. E.g. batched gemms.
- Rudimentary auto-disk in einsum
- Interface to FFT
- Utilize oneMKL to provide GPU capabilities
- Incorporate TBLIS of D. Matthews as a backend for better CPU performance

GitHub: github.com/jturney/EinsumsInCpp