

# Using declarative parallelism in C++

Igor Sfiligoi – San Diego Supercomputing Center (UCSD)

# Pragma-based parallelism with OpenMP

- OpenMP has emerged as the major pragma-based parallelization paradigm
- Explicitly tell the compiler that parallelization is both possible and desirable



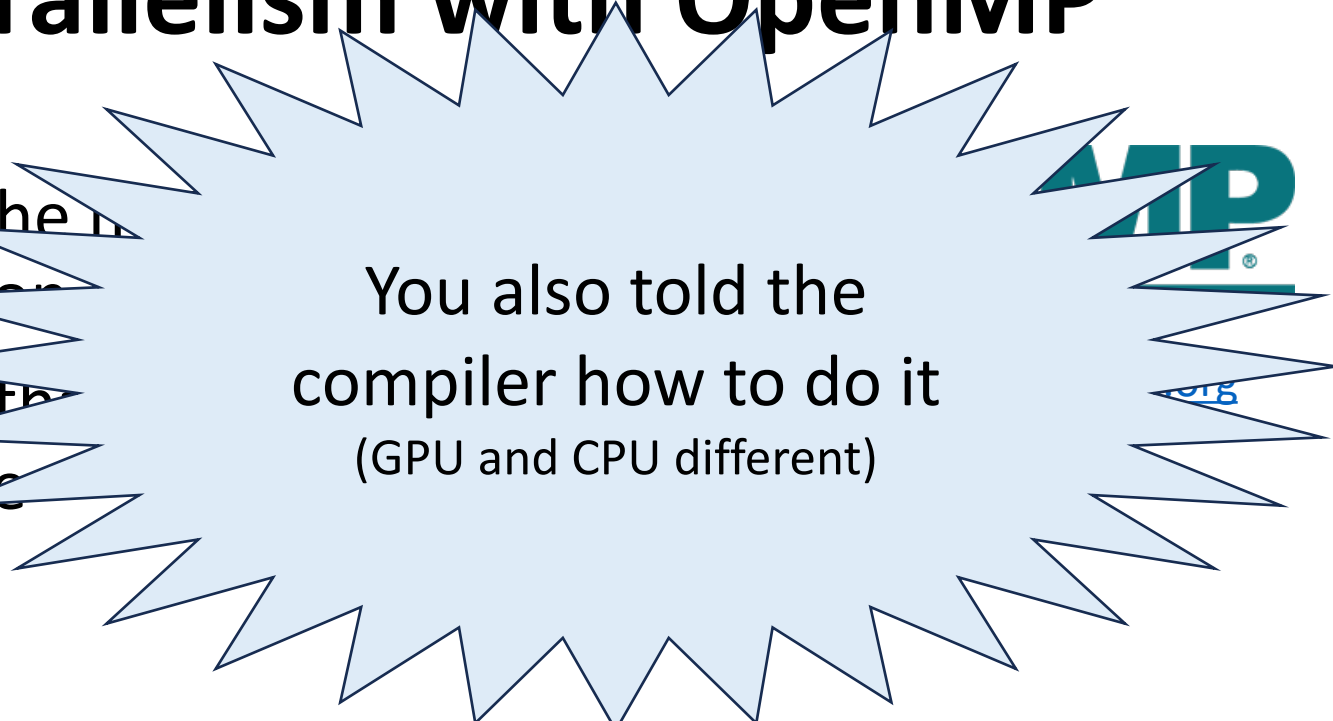
<https://www.openmp.org>

```
#pragma omp target teams distribute parallel for simd  
for (int i=0; i<N; i++) {  
    A[i] += 3*B[i]  
}
```

```
#pragma omp parallel for  
for (int i=0; i<N; i++) {  
    A[i] += 3*B[i]  
}
```

# Pragma-based parallelism with OpenMP

- OpenMP has emerged as the most popular pragma-based parallelization technique
- Explicitly tell the compiler to parallelize both possible and desirable



You also told the  
compiler how to do it  
(GPU and CPU different)

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
}
```

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
}
```

# Declarative parallelism with std C++

- C++ has support for parallel loops since C++17
- Has several variants of parallel for loops and support reductions, too
  - All assume you will operate on a buffer/container
- The "for body" becomes a function invocation
  - Most often expressed as an inline lambda function

```
std::transform(std::execution::par_unseq,  
    A, A+N, B, A,  
    [] (auto a, auto b) -> auto {  
        return a + 3*b;  
    }  
);
```

# Declarative parallelism with std C++

- C++ has support for parallel loops since C++17
- Has several variants of parallel for loops and support reductions, too
  - All assume you will operate on a buffer/container
- The "for body" becomes a function invocation
  - Most often expressed as an inline lambda function

```
std::transform(std::execution::par_unseq,  
A, A+N, B, A,  
[] (auto a, auto b) -> auto {  
    return a + 3*b;  
}  
);
```

We explicitly state  
that we allow for  
parallel execution.

Compiler decides  
how to parallelize.  
And if it should run  
on CPU or GPU.

# Declarative parallelism with std C++

- C++ has support for parallel loops since C++17
- Has several variants of parallel for loops and support reductions, too
  - All assume you will operate on a buffer/container
- The "for body" becomes a function in
  - Most often expressed as an inline lambda

Ideal for regular buffer transformations.

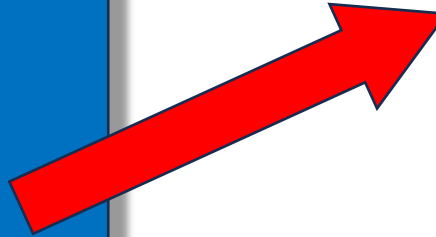
```
std::transform(std::execution::par_unseq,  
              A, A+N, B, A,  
              [] (auto a, auto b) -> auto {  
                  return a + 3*b;  
              }  
              );
```

Often becomes memory-bound!

# Declarative parallelism with std C++

- C++ has support for parallel loops since C++17
- **No support for partitioned memory for GPU compute**
  - Requires unified memory setups
  - Paged unified memory often works, but shared memory obviously better

```
double* A = new double[N];
double* B = new double[N];
...
#pragma omp target teams distribute \
parallel for simd \
    map(tofrom: A[0:N]) map(to: B[0:N])
for (int i=0; i<N; i++) {
    A[i] += 3*B[i];
}
```



```
????
std::transform(std::execution::par_unseq,
    A, A+N, B, A,
    [] (auto a, auto b) -> auto {
        return a + 3*b;
    }
);
```

# Multi-dimensional array problem

- Neither C nor C++ have native support for multi-dimensional arrays
  - I.e., matrices and tensors
  - At least for dynamically allocated memory
- Users typically resort to manual index compute
  - Possibly with the use of a helper function, but will make it explicit here

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[i*M+j]*R[i*M+j];
    }
    A[i] = total;
}
```



# Multi-dimensional array problem

- Neither C nor C++ have native support for multi-
  - I.e., matrices and tensors
  - At least for dynamically allocated memory
- Users typically resort to manual index compute
  - Possibly with the use of a helper function, but will make it explicit here

While this was ideal for CPUs, this is very bad for GPUs!

```
#pragma omp target team private total parallel for
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[i*M+j]*R[i*M+j];
    }
    A[i] = total;
}
```

# Multi-dimensional array problem

- Neither C nor C++ have native support for multi-dimensional arrays
  - I.e., matrices and tensors
  - At least for dynamically allocated memory
- Users typically resort to manual index compute
  - Possibly with the use of a helper function, but will make it explicit here

Due to vectorization at top level, we should switch order!

```
#pragma omp target teams distribute parallel for
for (int i=0; i<N; i++) {
    float total = 0.0;
    for (int j=0; j<M; j++) {
        total += L[j*N+i]*R[j*N+i];
    }
    A[i] = total;
}
```

# Multi-dimensional array problem

- Buffers should be accessed based on expected access order

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[i*M+j]*R[i*M+j];
    }
    A[i] = total;
}
```

} Vectorized on  
inner loop

```
#pragma omp target teams \
    distribute parallel for
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[j*N+i]*R[j*N+i];
    }
    A[i] = total;
}
```

} Vectorized on  
outer loop

# Multi-dimensional array problem

- Buffers should be accessed based on expected access order

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[i*M+j]*R[i*M+j];
    }
    A[i] = total;
}
```

} Vectorized on  
inner loop

No elegant solution for  
both CPU and GPU with  
OpenMP or std::C++  
(yet)

```
#p
c
for (int i=0; i<N; i++) {
    float total = 0 .0;
    for (int j=0; j<M; j++) {
        total += L[j*N+i]*R[j*N+i];
    }
    A[i] = total;
}
```

} Vectorized on  
outer loop

# Kokkos



<https://github.com/kokkos>

- Kokkos is a parallelization C++ library that treats multi-dimensional arrays as a first-class concept
- Based on the same declarative parallelization ideas as std:C++
  - Indeed, they contributed some of the ideas to the C++ standardization body
- Has support for partitioned memory spaces
- Hides even more details from the programmer
  - Picks both how to parallelize the code execution and how to order the memory indexes of the multi-dim. arrays/matrices

# Kokkos for basic parallelism

- Much closer to standard for than C++17



<https://github.com/kokkos>

```
Kokkos::parallel_for("Add B to A", N,  
  KOKKOS_LAMBDA (const int i) {  
    A[i] += 3*B[i];  
  }  
);
```

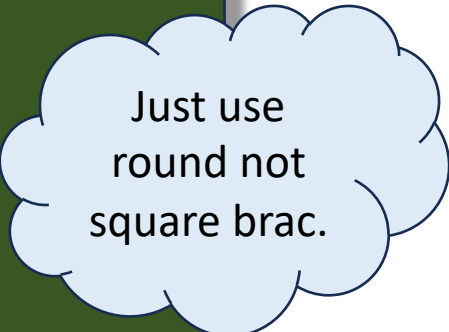
```
#pragma omp parallel for  
for (int i=0; i<N; i++) {  
  A[i] += 3*B[i];  
}
```

# Kokkos views for partitioned memory

- Kokkos views are a way to abstract memory location
  - Behave like `std::shared_ptr`
  - Memory will be allocated where the compute will happen

```
double* A = new double[N];
double* B = new double[N];
...
#pragma omp target teams distribute \
parallel for simd \
map(tofrom: A[0:N]) map(to: B[0:N])
for (int i=0; i<N; i++) {
    A[i] += 3*B[i];
}
```

```
Kokkos::View<double*> A(N);
Kokkos::View<double*> B(N);
...
Kokkos::parallel_for(N,
    KOKKOS_LAMBDA (const int i) {
        A(i) += 3*B(i); • • •
    });
```

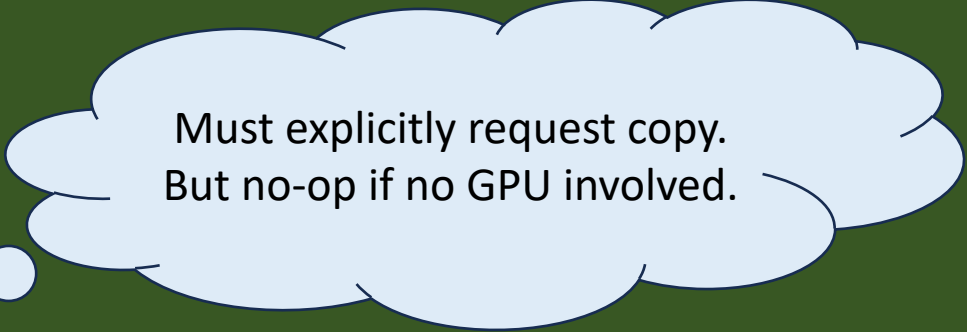


Just use  
round not  
square brac.

# Kokkos views for partitioned memory

- Mixing and matching CPU and GPU compute a bit more complex
  - Views by default live in “default execution space”
  - If default on GPU, buffer not available in CPU code
- Mirrors provide guaranteed access on CPU

```
Kokkos::View<double*> A(N);
typename Kokkos::View<double *>::HostMirror Acpu = Kokkos::create_mirror_view(A);
Kokkos::View<double*> B(N);
...
Kokkos::parallel_for(N,
  KOKKOS_LAMBDA (const int i) {
    A(i) += 3*B(i);
  });
Kokkos::deep_copy (A, Acpu);
std::cout << Acpu(12) << std::endl;
```



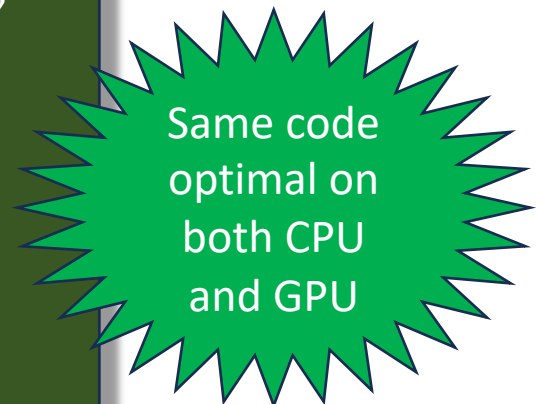
Must explicitly request copy.  
But no-op if no GPU involved.



# Kokkos views are natively multi-dimensional

- Can list up to 8 dimensions
  - Both at construction and at access time
- No need to do index math
  - The View object does that automatically for you
  - Internal layout optimized for the execution target
  - Always use the first index for the external loop
- Same for the Mirror object

```
Kokkos::View<double*> A(N);  
Kokkos::View<double**> L(N,M);  
Kokkos::View<double**> R(N,M);  
  
...  
Kokkos::parallel_for(N,  
  KOKKOS_LAMBDA (const int i) {  
    float total = 0 .0;  
    for (int j=0; j<M; j++) {  
      total += L(i,j) * R(i,j);  
    }  
    A(i) = total;  
  });
```



Same code  
optimal on  
both CPU  
and GPU

# Reductions supported, too

- Kokkos supports reductions, too
- Just a slightly different syntax (a bit like with C++17)



<https://github.com/kokkos>

```
double s = 0.0;
Kokkos::parallel_reduce("My global sum", N,
    KOKKOS_LAMBDA (const int i, double &mysum) {
    mysum += A[i]*sin(B[i]);
}, Kokkos::Sum<double>(s));
// use s here
```

**Ready for some more hands-on?**