

Beyond basics in parallelism

Igor Sfiligoi – San Diego Supercomputing Center (UCSD)

Loop parallelism

- As a reminder, most parallelism in loops

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

Reductions

- When you compute in parallel, you should **never** write to the same variable/memory location from multiple iterations
 - That would lead to unpredictable results
- Common operations across iterations supported as reductions
 - The reduction variable is kept private during compute
 - All intermediate values then combined automatically at the end of the loop

```
s = 0.0;
#pragma omp target teams distribute parallel for simd \
    reduction(+:s)
for (int i=0; i<N; i++) {
    s += A[i]*sin(B[i])
}
// use s here
```

Partitioned memory

- Reminder: CPU and GPU memories are typically separate
 - Buffers must be moved between them as needed
- OpenMP can try to infer what needs to be done
 - But doing it explicitly best

```
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];  
}
```


Classes/Objects and partitioned memory

- Works well for both numeric buffers and buffers of simple objects

```
double* A = new double[N];
MyClass* B = new MyClass[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] += B[i].compute();
}
```

Classes/Objects and partitioned memory

- But will not automatically handle nested buffers!
 - Most compilers also will not automatically detect the need for copying internal buffers
 - Resulting in code crashing
- Using pointers is OK, but the pointed buffer must be handled independently



```
class MyClass {  
    double *inarr;  
public:  
    MyClass() : inarr(new double[34]) {}  
    double compute() {return inarr[2] + inarr[7];}  
};
```

```
double* A = new double[N];  
MyClass* B = new MyClass[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] += B[i].compute();  
}
```

Classes/Objects and unified memory

- Nested buffers allowed when CPU and GPU can access each other's memory
 - **Makes life much easier!**
 - **But only few platforms support it right now!**
E.g. NVIDIA Grace Hopper and AMD MI300A APU

```
#pragma omp target requires unified_shared_memory
```

```
class MyClass {  
    double *inarr;  
public:  
    MyClass() : inarr(new double[34]) {}  
    double compute() {return inarr[2] + inarr[7];}  
};
```

```
double* A = new double[N];  
MyClass* B = new MyClass[N];  
...
```


```
#pragma omp target teams distribute parallel for simd
```

```
for (int i=0; i<N; i++) {  
    A[i] += B[i].compute();  
}
```

No dynamic memory in GPU compute

- Dynamic memory management in GPU code very limited (e.g. new and/or malloc)
 - Slow and typically limited to a KB of memory
 - Normally does not interoperate with CPU-based dynamic memory
- Makes using standard C++ containers virtually impossible (e.g., std::vector)

```
std::vector *A = new std::vector[N];  
...  
#pragma omp target teams distribute parallel for simd  
for (int i=0; i<N; i++) {  
    A[i].push_back(B[i].compute());  
}
```



GPU compute and external functions

- Functions that can be inlined are fine
 - Compiler will treat them like if you cut-and-paste the code
- Separate compilation needs careful handling
 - Must be explicitly declared as GPU-enabled
 - Tends to be rather slow with current compilers

```
double compute(double a, double b) {  
    return a + 3*b;  
}
```

```
#pragma omp declare target to(compute)
```

```
double* A = new double[N];  
double* B = new double[N];
```

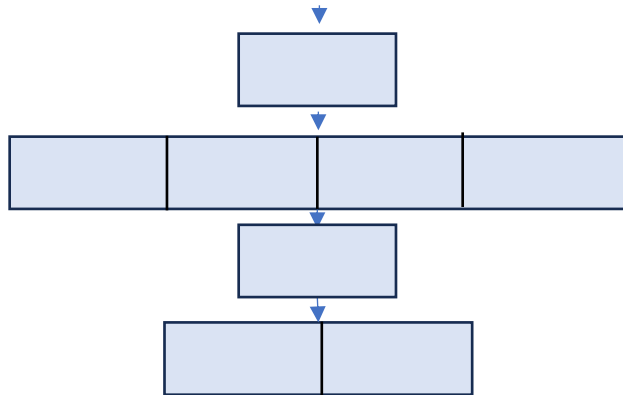
```
...
```

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

Loops in CPUs and GPUs

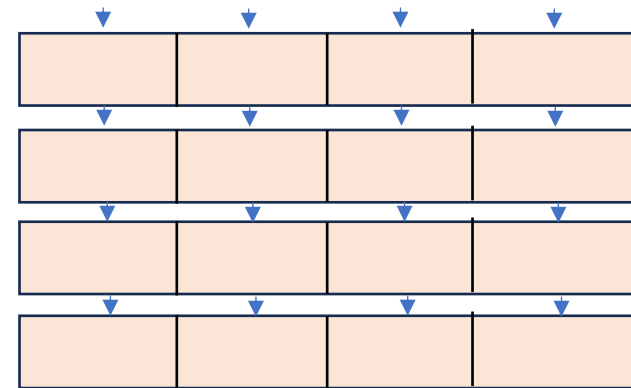
CPU

- Each iteration drives one CPU core
- Steps can be scalar or vector
 - **No (standard) explicit vector syntax**
 - SIMD just compiler "magic"



GPU

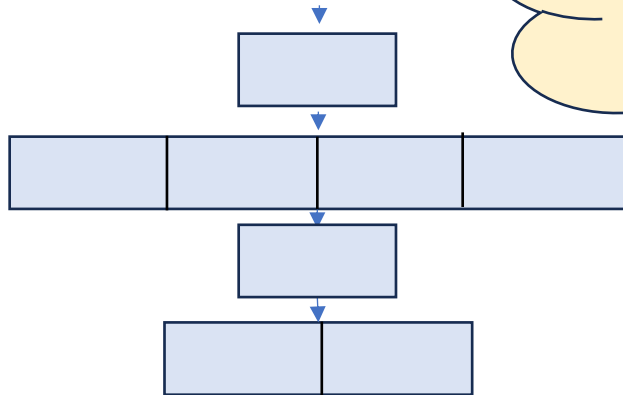
- Each iteration drives one element of the GPU wavefront
- **Several adjacent iterations executed as single vector unit**



Loops in CPUs and GPUs

CPU

- Each iteration drives one CPU core
- Steps can be scalar or vector
 - **No (standard) explicit vectorization**
 - SIMD just compiler “magic”

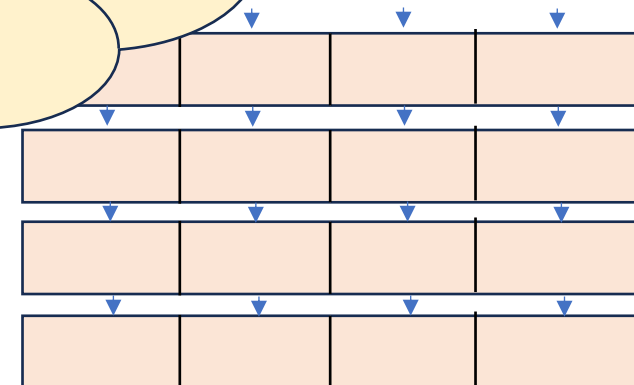


GPU

- Each iteration drives one element of the GPU wavefront

Several iterations often merged by compiler for better vectorization.

Recent iterations merged by compiler for better vectorization.



Loops in CPUs and GPUs

CPU

- Each iteration drives one core
- Steps can be scalar or vector
 - **No (standard) explicit vector syntax**
 - SIMD just compiler "magic"
- Each iteration independent
 - No coordination between iterations (unless compiler merges them)
- Logic (if) just another instruction
 - Typically predicted and/or speculated

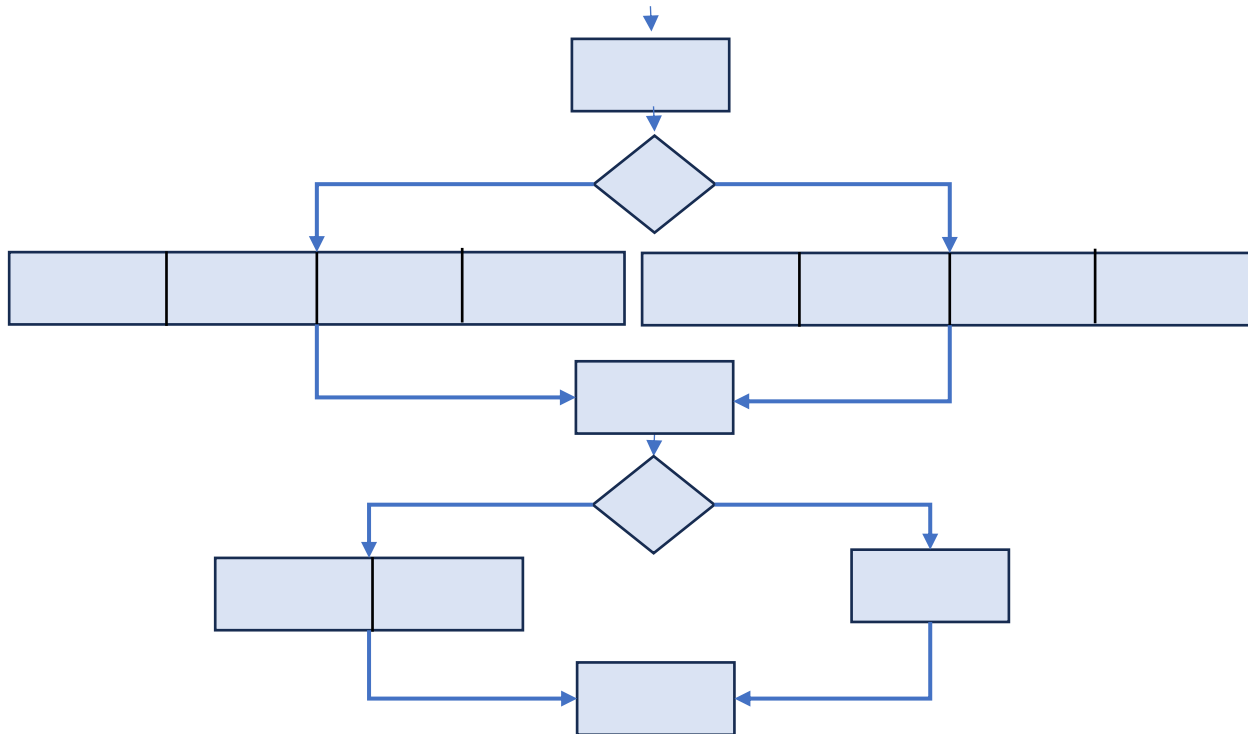
GPU

- Each iteration drives one element in the wavefront vector
 - All steps are always scalar
 - HW groups them in vectors (SIMT)
- All instructions in a vector progress in lockstep
 - But no coordination between vectors (wavefronts)
- Logic (if) may split vector
 - **Reducing parallelism!**
 - And likely to yield until decision

Loops in CPUs and GPUs

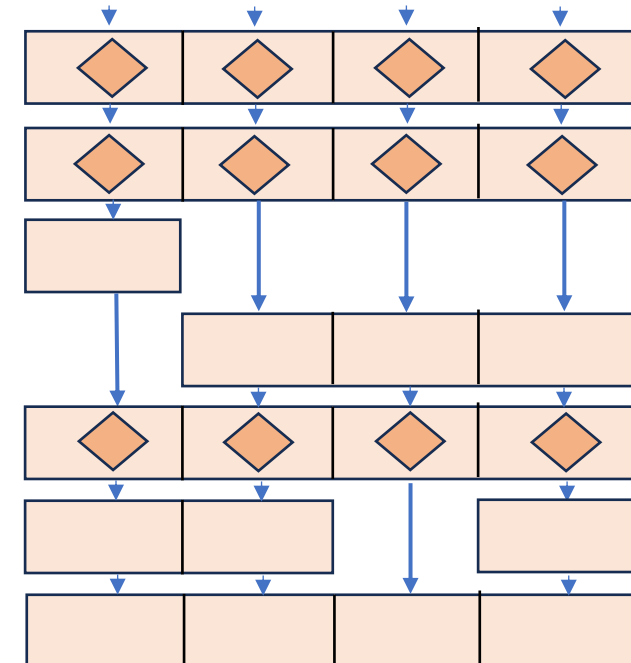
CPU

- Logic (if) just another instruction
 - Typically predicted and/or speculated



GPU

- Logic (if) may split vector (at runtime)
 - **Reducing parallelism!**
 - And likely to yield until decision



Implications on high-level parallelism

- Simple loops are a no-brainer
 - The compiler will always do the right thing
 - Both for CPUs and GPUs

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

All operations are vectorized

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

No logic -> no wavefront splitting

Implications on high-level parallelism

- Complex loops can be tricky
 - On CPUs, logic and nesting may make it hard to vectorize

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    double x = B[i] * C[i];
    double y = 0;
    for (int j=0; j<D[i/32]; j++) {
        if (E[i/4+j]>x) y+=A[i]*E[i/4+j];
    }
    A[i] = y
}
```

GPU equivalent would have no problems “vectorizing” at all.

Implications on high-level parallelism

- Complex loops can be tricky
 - On GPUs, logic can easily fragment the vectors

```
#pragma omp target teams distribute \
    parallel for simd
for (int i=0; i<N; i++) {
    // some complex code
    if (B[i]>1.0) {
        // some complex code
    } else {
        // some complex code
    }
}
```

**Effectively doubling the cost
on GPUs (in many cases)**

And will get worse if
there is more logic inside
the branches

CPUs would have no problems at all.
Each branch would be vectorized separately.
(wherever possible)

Asynchronous execution

- Reminder: CPU and GPU compute proceed independently
- **However, OpenMP is by default blocking**
 - The compiler will insert an implicit wait/barrier
 - **Makes writing code much easier**

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
}
// implicit GPU wait
... // more CPU code here
```

Asynchronous execution

- Reminder: CPU and GPU compute proceed independently
- OpenMP does support async execution
 - But must be explicitly requested
 - Based on the notion of data dependencies

```
#pragma omp target teams distribute parallel for simd map(tofrom: A) nowait
for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
}
... // more independent CPU code here
#pragma omp ... depend(A)
... // GPU finished, can safely use A here
```

Ready for some more hands-on?