Into to C/C++ pragma-based parallelism

Igor Sfiligoi – San Diego Supercomputing Center (UCSD)

We will mostly focus on C

- “Full” C++ hides a lot of details from the final user
  - In C, you explicitly tell how to do things, step by step
  - C better for understanding how things work
- Most of the presented concepts still apply to “full” C++
  - But you may need to understand the minute details that the compiler normally tries to hide from you!
  - Wherever appropriate, will call out known “gotchas”

We will focus on portable methods

- Life is too short to re-implement your apps for every architecture!
  - That’s why we use a compiler and not ASM to start with!

- Picking a vendor specific toolkit may give you better performance
  But with dominant vendors changing, hard to maintain/port
  - CPUs - pthreads
  - NVIDIA GPUs – CUDA
  - AMD GPUs – ROCm/HIP
  - Intel GPUs – OneAPI/SYCL

We will focus on portable methods

- Life is too short to re-implement your apps for every architecture!
  - … start with!
  - … better performance
  - … to maintain/port
  - NVIDIA GPUs – CU…
  - AMD GPUs – ROCm/HIP
  - Intel GPUs – OneAPI/SYCL

No magic solutions, some porting/tuning always needed with new platforms.

We will focus on portable methods

- Life is too short to re-implement your apps for every architecture!
    - That’s why we use a compiler and n... ASM to start with!

- Picking a vendor specifi...
  But with dominant v...
    - CPUs - pthreads
    - NVIDIA GPUs – CU...
    - AMD GPUs – ROCm/HIP
    - Intel GPUs – OneAPI/SYCL

Will focus on mature solutions, so you can start using them now.

(Some cutting-edge solutions do promise to make your life easier)

Virtually all parallelism in loops

- Compilers have hard time parallelizing sequential code
  - Not completely impossible, but very limited potential
  - Sometimes you can help
    - But often makes code hard to maintain

```
a1 += 2*b1;
a2 += 2*b2;
a3 += 2*b3;
a4 += 2*b4;
```

- Loops are the natural concept for expressing parallelism, if
  - Each iteration independent
  - Number of iterations known in advance

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

Virtually all parallelism in loops

- Compilers have hard time parallelizing sequential code
  - Not completely impossible, but very limited potential
  - Sometimes you can help
    - But often makes code hard to maintain

```
a1 += 3*b1;
a2 += 3*b2;
a3 += 3*b3;
a4 += 3*b4;
```

- Loops are the natural concept for expressing parallelism, if
  - Each iteration independent
  - Number of iterations known in advance

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

A good compiler will issue vectorized code for both snippets

Loops and dependencies

- Loops are the natural concept for expressing parallelism, if
  - Each iteration independent
  - Number of iterations known in advance

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

As long as A and B do not overlap, it is pretty obvious each iteration is independent.

Dependencies hard to infer

- Loops are the natural concept for expressing parallelism, if
  - Each iteration independent
  - Number of iterations known in advance

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

- Doesn’t take much for compiler to give up
  - 1st priority for the compiler is to generate correct code
  - Optimization/speed is distant 2nd

```
for (int i=0; i<N; i++) {
  A[i] += 3*myfunc(A,B,i)
}
```

Good luck, if
myfunc not inlined
and/or not trivial.

Dependencies hard to infer

- Loops are the natural concept for expressing parallelism, if
  - Each iteration independent
  - Number of iterations known in advance

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

- Doesn't take much for compiler to give up
  - 1st priority for the compiler is to generate correct code
  - Optimization/speed is distant 2nd

```
for (int i=0; i<N; i++) {
  A[i] += 3*myfunc(A,B,i)
}
```

- Not to mention "smart" users

```
for (int i=0; i<N; i++) {
  A[i] += 3*B[i]
  if(A[i]>3) N=A[i];
}
```

Helping the compiler with pragmas

- OpenMP has emerged as the major pragma-based parallelization paradigm
  - Especially on CPUs
  - Just decorate the loop with a pragma
- Explicitly tell the compiler that parallelization is both possible and desirable

https://www.openmp.org

```
#pragma omp parallel for
  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*myfunc(A,B,i)
  }
```

Helping the compiler with pragmas

- OpenMP h
  paralleli
  -
  - Ju
- Explicitly … on is
  both possible and desira…

Compiler will trust you!
If the code was not independent,
you will get unexpected results.

https://www.openmp.org

```
#pragma omp parallel for
  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*myfunc(A,B,i)
  }
```

GPU extensions

- OpenMP recently added GPU support
  - The notion of “OMP TARGET”
- Slightly different syntax, but similar in concept

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

https://www.openmp.org

GPU pragmas - OpenACC

- OpenACC was the pioneer in GPU-based pragma-based parallelization
- Only supported by NVIDIA compilers

https://www.openacc.org

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

Very unfortunate.

Much easier to use than
OpenMP Target in my opinion.

Remember the launch cost

- Every parallel section incurs a “launch cost”
  - Very expensive on GPUs
  - But not negligible on CPUs, either
- Pack as much as possible in a single parallel section
  - Can be order of magnitude faster

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
  }
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
    C[i] += 4*D[i]
  }
```

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
    A[i] += 3*B[i]
    C[i] += 4*D[i]
  }
```

But don’t break semantics!

- Every parallel section incurs a “launch cost”
  - Very expensive on GPUs
  - But not negligible on CPUs, either
- Pack as much as possible in a single parallel section
  - Can be order of magnitude faster
  - **But make sure you preserve the independence of the iterations!**

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     A[i] += 3*B[i]
  }
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     C[i] += 4*A[1+(2*i+5)%N]
  }
```

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     A[i] += 3*B[i]
     C[i] += 4*A[1+(2*i+5)%N]
  }
```

Remember memory locality

- Do not iterate over same buffer multiple times
  - If at all possible (dependencies)
- Pack as much as possible in a single parallel section
  - Also use temp variables when appropriate
  - **Will be** order of magnitude faster on large buffers

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     A[i] += 3*B[i]
  }
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     C[i] += 4*A[i] + 3*B(i)
  }
```

```
#pragma omp target teams \
  distribute parallel for simd
  for (int i=0; i<N; i++) {
     b3 = 3*B[i]
     a2 = A[i] + b3
     A[i] = a2
     C[i] += 4*a2 + b3
  }
```

Dealing with partitioned memory

- Remainder: CPU and GPU memories are (typically) independent
- By default, OpenMP will copy all buffers
  - From CPU to GPU memory before GPU kernel start
  - From GPU to CPU memory after GPU kernel completion

Slow!

- If buffers only needed on one side, you can avoid the transfer
  - Resulting in drastic speedup!

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

Dealing with partitioned memory

- OpenMP Target memory uses reference counters
  - Will only transfer once,
    even if mentioned several times
- 3+1 possible directions:
  - “to” – CPU ->GPU
  - “from” – GPU -> CPU
  - “tofrom” – both ways
  - May create uninitialized buffer

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

Dealing with partitioned memory

Copy “A” CPU->GPU

- OpenMP Target … uses reference counters
  - Will …
    ev… …nes
- 3+1 poss… …ons:
  - “…
  - … t… … ways
  - May cr…

Use GPU’s “A”

Copy “B” CPU->GPU

Release GPU’s “B”

Release GPU’s “A”

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

Dealing with partitioned memory

Copy “A” CPU->GPU

- OpenMP Target ... uses refere...
  - Will ... ev... nes

Use GPU’s “A”

- 3+1 poss... ...ons:
  - “...
  - ...to ... ways

Copy “B” CPU->GPU

  - May cr...

Release GPU’s “B”

Release GPU’s “A”

In this example,
CPU’s “A” never updated

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

Managed memory

- Remainder: CPU and GPU memories are (typically) independent
  - **But most modern GPUs can hide this through managed memory**
  - No need to explicitly manage memory in that case
- Only works on buffers explicitly allocated as managed
  - Either explicitly,
    by replacing malloc with e.g. (not portable)
    `cudaMallocManaged`
  - Or **implicitly**,
    by using compiler directives, e.g.
    `nvc++ -gpu=managed`
- GPU driver will page those buffers as needed between CPU and GPU memories

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

Managed memory

- Remainder: CPU and GPU memories are (typically) independent
    - **But most modern GPUs can hide this through managed memory**
    - No need to explicitly manage memory in that case
- Only works on buffers explicitly allocated as managed
    - Either explicitly,
      by replacing malloc with e.g. (not portable)
      `cudaMallocManaged`
    - Or **implicitly**,
      by using compiler directives, e.g.
      `nvc++ –gpu=managed`
- GPU driver will page those buffers as needed between CPU and GPU memories

```
cudaMallo
 …
#pragma om                d
 for (int i=0; i<N; i
   A[i] += 3*B[i]
 }
```

Be careful not to start trashing! (swap like)

Ready for some hands-on?