Introduction to OpenMP GPU-offloading

Tianle Wang
07/08/2022
Overview

- OpenMP is an API for multithreading that was first developed in 1997.
- Originally it only supported Shared-Memory parallel computing on multicore architectures.
- Since OpenMP 4.0, it added support for “target offloading” on heterogenous architectures, such as CPU+GPU.
- OpenMP uses a set of compiler directives and API function calls.

```
#pragma omp parallel
{
    printf("Hello from process: %d\n", omp_get_thread_num());
}
```
Loop-Level Parallelism with OpenMP

OpenMP directives are expressed in pragmas. In C/C++, they begin with `#pragma omp`

If OpenMP is not supported or enabled, the `#pragma`’s will be ignored by the compiler.

The most useful OpenMP pragma may be the one for loop-level parallelism:
- `#pragma omp parallel for`
  - Which needs to be followed immediately with a for loop

The compiler takes the directive and parallelizes the loop for you.
- You can set the number of parallel threads through the `OMP_NUM_THREADS` environment variable at runtime
- Or you can set it in the code through `omp_set_num_threads()`

Other commonly used constructs: reduction, private, task, simd
CPU+GPU Heterogeneous Computing

• At present, GPUs are used as accelerators to the host CPUs.
• The CPU launches the program and dispatches the work to either the CPU or the GPU.
• Data will usually first be loaded to the CPU memory from storage.
• To perform GPU computation, you need to make sure data is available on the GPU memory. Two ways to do this:
  • Explicit data management
    • map clause
    • omp_target_alloc()
  • Unified shared memory/managed memory
OpenMP for GPU Computing

• To enable GPU computing, OpenMP uses the “target offloading” model.
• When the target region is encountered, the main thread will attempt to initiate the computation on the target device, e.g., the GPU in this case.
• Data will be moved to/from the GPU as needed/specified by the user.
• OpenMP is a specification; actual support and implementations for different GPU architectures depend on the compilers. See https://www.openmp.org/resources/openmp-compilers-tools/
Example: axpy

```c
void axpy(double a, double *x, double *y, int sz)
{
    #pragma omp target teams distribute parallel for \
        map(to:x[0:sz]) \ 
        map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++)
    {
        y[i] = a * x[i] + y[i];
    }
}
```

**LLVM:** clang++ -fopenmp -fopenmp-targets=nvptx64-nvidia
-Xopenmp-target -march=<gpu_arch>

**GNU:** g++ -fopenmp -fopenmp-targets=nvptx64-nvidia
-Xopenmp-target -march=<gpu_arch>

**NVIDIA:** nvc++ -mp=gpu

target – indicates the code block below will be executed on the target device.
teams distribute – indicates there will be a league of teams doing the work
parallel for – the work will be shared by parallel threads
How teams distribute/parallel map to the GPU architectures depends on the compiler
map – copy the data associated with the variables to or from the target memory, or just allocate target memory
Different from Python: `data[st:sz]` instead of `data[st:ed]`
Optimizing Data Transfers 1: Explicit map

• If map clauses are not added to target constructs, presence checks determine if data is already available in the device data environment.
• If not, a map(tofrom:...) is added for that data item.
• If data is a pointer to the first element of the array, need to specify the size.

```c
double x[sz];
double y[sz];
#pragma omp target teams distribute parallel for
for (int i = 0; i < sz; i++)
{
    y[i] = a * x[i] + y[i];
}
```

```c
double x[sz];
double y[sz];
#pragma omp target teams distribute parallel for
\map(to:x[0:sz]) \map(tofrom:y[0:sz])
for (int i = 0; i < sz; i++)
{
    y[i] = a * x[i] + y[i];
}
```
Optimizing Data Transfers 2: Data environment

{ 
  double *x = ...
  double *y = ...
  axpy(a, x, y, sz);
  axpy(a, x, y, sz);
}

• The above code copy x from host to device twice

• Use data environment to remove the unnecessary data copy

{ 
  double *x = ...
  double *y = ...
  #pragma omp target enter data map(to:x[0:sz])
  axpy(a, x, y, sz);
  axpy(a, x, y, sz);
#pragma omp target exit data map(delete:x[0:sz])
}

With careful data environment setup (enter data/ exit data), we can remove map clause inside axpy to remove the unnecessary data copy
```c
{  
    int h = omp_get_initial_device(); //host id
    int t = omp_get_default_device(); //device id
    double * x_d = (double *)omp_target_alloc(sizeof(double) * N, t);
    #pragma omp target is_device_ptr(x_d) device(t)
    #pragma omp teams distribute parallel for
    for(int i=0;i<N;i++)
        x_d[i] = sqrt(double(i));
    omp_target_free(x_d, t);
}
```

- Don't need to allocate memory on host
- Usage more similar to CUDA
- Can still use `omp_target_memcpy()` to do memory transfer between host/device and device/device
Atomic operation and scattering add

• One important kernel in wire-cell is scattering add:
  
  \[ \text{res[idx[i]] += data[i]} \]

• Use atomic construct to implement this for parallel execution

  ```
  #pragma omp target teams distribute parallel for simd
  for(int i=0; i<N; i++)
  {
    #pragma omp atomic update
    res[pos[i]] += data[i];
  }
  ```

• Or we can use use_device_ptr with a wrapper of cuda/hip atomicAdd API
Atomic operation and scattering add

\[ N = 1024 \times 1024 \times 32 \]
\[ \text{res.size()} = 1024 \times 128 \]

Use float as data type

<table>
<thead>
<tr>
<th>Compiler/CPU Settings</th>
<th>Performance (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang 15, NVIDIA V100 (BNL lambda), no -O3</td>
<td>0.016413</td>
</tr>
<tr>
<td>clang 15, NVIDIA V100 (BNL lambda), with -O3</td>
<td>0.00908089</td>
</tr>
<tr>
<td>nvcc 21.9, NVIDIA V100 (Cori nersc)</td>
<td>0.0454831</td>
</tr>
<tr>
<td>clang 13, AMD gfx906</td>
<td>0.0453169</td>
</tr>
</tbody>
</table>

- Openmp with atomic: 1.28145
- Cuda/hip: 0.00890096
- Openmp with use_device_ptr: 0.00870609

- Must turn on -O3 with clang compiler to get good performance
- Performance is still a factor of two lower than cuda
- Some strange bug appears, see
  https://github.com/GKNB/test-benchmark-OpenMP-atomic/tree/main/test_use_device_ptr_atomic
Use vendor library

• OpenMP does not provide libraries like BLAS, FFT, random number

• Currently, one way to do that is to call those functions in vendor library using "use_device_ptr" clause

• The use_device_ptr map type allows OpenMP device arrays to be passed to accelerated libraries.

• Don't confuse that with is_device_ptr, which does the opposite.

```c
float *x = (float*)malloc(sizeof(float) * sz)
float *y = (float*)malloc(sizeof(float) * sz)
float a = 2.0

#pragma omp target data map(to:x[0:sz]) map(tofrom:y[0:sz])
{
  #pragma omp target data use_device_ptr(x,y)
  {
    cublasSaxpy(handle, sz, &a, x, 1, y, 1);
  }
}

cudaMalloc((void**)&x_d, sz * sizeof(float));
cudaMalloc((void**)&y_d, sz * sizeof(float));
saxpy(n, a, x_d, y_d);
cudaMemcpy(y, y_d, ...);
```
Random number generator

• We build our own library for generating random numbers, which is basically a wrapper for curand/rocrand/std::random/random123
• https://github.com/GKNB/test-benchmark-OpenMP-RNG
• Various architectures support: NVIDIA GPU, AMD GPU, CPU running in serial, CPU running in parallel
• Currently support uniform/normal distribution of float/double
• Various engine used (not for all architectures ): philox, xorwow, mt19937, ...
• Simple usage for GPU:

1).
   double* data_d = (double*)omp_target_alloc(sizeof(double) * sz, device_id);
   omp_get_rng_normal_double(data_d, sz, 0.0, 10.0, 1234ull, generator_enum::philox);

2).
   #pragma omp target data map(alloc:data[0:sz]) use_device_ptr(data)
   omp_get_rng_normal_double(data, sz, 0.0, 10.0, 1234ull, generator_enum::philox);

• Currently trying to add more features into the library and still under testing
OpenMP also allows explicit use of shared memory

```c
#pragma omp target teams distribute
for(int ib=0; ib<N/BLOCK_SIZE; ib++)
{
    int temp[BLOCK_SIZE + 2 * RADIUS];
#pragma omp allocate(temp) allocator(omp_pteam_mem_alloc)
#pragma omp parallel for num_threads(BLOCK_SIZE)
    for(int it=0; it<BLOCK_SIZE; it++)
    {
        int gindex = it + ib * BLOCK_SIZE;
        int lindex = it + RADIUS;
        temp[lindex] = in[gindex + RADIUS];
        if(it < RADIUS)
        {
            temp[lindex - RADIUS] = in[gindex - RADIUS + RADIUS];
            temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE + RADIUS];
        }
    }
#pragma omp parallel for num_threads(BLOCK_SIZE)
    for(int it=0; it<BLOCK_SIZE; it++)
    {
        int gindex = it + ib * BLOCK_SIZE;
        int lindex = it + RADIUS;
        int result = 0;
        for (int offset = -RADIUS; offset <= RADIUS; offset++)
            result += temp[lindex + offset];
        out[gindex + RADIUS] = result;
    }
}
```

Example of 1D stencil, use `omp_pteam_mem_alloc` to indicate the type of memory (shared memory) we allocate.

- Can observe the usage of shared memory with nsight-compute.
- If allocate too much shared memory, a compile-time error would occur.
Asynchronous Offloads

OpenMP target constructs are synchronous by default

• The encountering host thread awaits the end of the target region before continuing

• The nowait clause makes the target constructs asynchronous

• We can have asynchronous behavior between different kernels and data movement
Example of Asynchronous Offloads

```c
for(int batch = 0; batch<nb; batch++)
{
    int st = N / nb * batch;
    int sz = N / nb;
    int *anext = a + st;
    int *bnext = b + st;
    int *cnext = c + st;
    #pragma omp target enter data map(to:anext[0:sz],bnext[0:sz],cnext[0:sz]) nowait depend(out:anext[0:sz],bnext[0:sz],cnext[0:sz])
    #pragma omp target teams distribute parallel for nowait depend(inout:anext[0:sz],bnext[0:sz],cnext[0:sz])
    for(int i=0; i<sz; i++)
    {
        compute_c_from_a_and_b();
    }
    #pragma omp target exit data map(release:anext[0:sz],bnext[0:sz]) nowait depend(in:anext[0:sz],bnext[0:sz])
    #pragma omp target exit data map(from:cnext[0:sz]) nowait depend(in:cnext[0:sz])
} //batch
#pragma omp taskwait
```
Example of Asynchronous Offloads

For nvc++, the earlier code will not output Asynchronous behavior
Future

• Testing and apply other syntax
  • unroll
  • tile
  • Loop
  • managed memory

• OpenMP side
  • Support for target scan (prefix sum)
  • General wrapper for vendor library
  • Bug fixing