Extending OpenMP map Clause to Bridge Storage and Device Memory

Kewe Yan, Anjia Wang, Xinyao Yi, Yonghong Yan
University of North Carolina at Charlotte, NC, USA

MCHPC'19: Workshop on Memory Centric High Performance Computing
Outlines

- Background
- Motivation
- Extension to OpenMP `map` clause
- Prototype implementation for the runtime
- Benefits
- Future work
- Conclusion
- Acknowledgement
Background

Illustrations for offloading work flow for GPU programming:

Step 1: Copy input data from CPU memory to GPU memory
Background

Illustrations for offloading work flow for GPU programming:

Step1: Copy input data from CPU memory to GPU memory

Step2: Load GPU program and execute, caching data on chip for performance
Background

Illustrations for offloading work flow for GPU programming:

Step1: Copy input data from CPU memory to GPU memory

Step2: Load GPU program and execute, caching data on chip for performance

Step3: Copy results from GPU memory to CPU memory
Background

A brief example for OpenMP `map` clause:

```
#pragma omp target \
  map(to:A[0:numElements],B[0:numElements]) \
  map(from:C[0:numElements])
```

**map-types:**

- **to**: copy A and B from host to device
- **from**: copy computing result from device to host
Motivation

Expand `map` clause to enable data copy from storage to device

- Bridge storage and device memory
- Reduce programming effort
- Handle complex data type

*in terms of programming*
Extension to OpenMP **map** clause

First glance of the extended **map** clause:

```c
#pragma omp target \
    map(to:A[0:numElements]="data/vectorA.data", \n        B[0:numElements]="data/vectorB.data") \n    map(from:C[0:numElements]="data/vectorC.data")
```

Compared with **map** clause:

- A and B: file for reading from storage
- C: file for writing back to storage
Extension to OpenMP \texttt{map} clause

An \texttt{optional} field for list item of locator-list:

\begin{verbatim}
list-item [= \{[data-format-driver:] data-location[, place-modifier][, \ 
metadata([place-modifier,] meta-identifier)]]}
\end{verbatim}

- \texttt{data-format-driver} : posix, jpeg, png, ...
- \texttt{data-location} : local file, storage device, URL
- \texttt{place-modifier} : host, hostonly
- \texttt{meta-identifier} : meta_in, meta_out
Extension to OpenMP map clause

Extended OpenMP map clause example - loading image data

```c
#pragma omp target \\    map(to:imgin={jpeg:“image_in.jpg”, \ 
    metadata(host:meta_in)}) \ 
map(from:imgout={jpeg:“image_out.jpg”, \ 
    metadata(host:meta_out)})
```

- **list item**: imgin, imgout
- **data-format-driver**: jpeg
- **data-location**: image_in.jpg, image_out.jpg
- **place-modifier(for the metadata)**: host
- **meta-identifier**: meta_in, meta_out
The main idea for the implementation is that applying host memory as bounce buffer.

Two stages of data copy:
- storage to host: fread or imgread
- host to device: CUDA function calls – global memory and unified memory

*in terms of implementation
Prototype implementation for the runtime

Example 1: POSIX stream data: Matrix Multiplication on GPU

```c
fd = fopen("data/vectorA.data", "rb");
fread(tA, sizeof(float), N*K, fd);
fclose(fd);
...
cudaMalloc(&A, sizeof(float)*N*K);
cudaMemcpy(A, tA, sizeof(float)*N*K, cudaMemcpyHostToDevice);
...
cudaMalloc(&C, sizeof(float)*N*M);
...
float *h_C = (float*)malloc(sizeof(float)*N*M);
...
// MM kernel
...
cudaMemcpy(h_C, C, sizeof(float)*N*M, cudaMemcpyDeviceToHost);
...
FILE *f3;
f3 = fopen("data/vectorC.data", "wb");
fwrite(h_C, sizeof(float), N*M, f3);
fclose(f3);
```

map (to:A[0:numElements]={"data/vectorA.data"}

host memory is used as bulk bounce buffer

map(from:C[0:numElements]={"data/vectorC.data"})
Prototype implementation for the runtime

Example 2: POSIX stream data: Matrix Multiplication on GPU

```c
int fd = fopen("data/vectorA.data", "rb");
cudaMallocManaged(&A, sizeof(float)*N*K);
fwrite(A, sizeof(float), N*K, fd);
fclose(fd);
...

cudaMallocManaged(&C, sizeof(float)*N*M);
...

// MM kernel
...

FILE *f3;
f3 = fopen("data/vectorC.data", "wb");
fwrite(C, sizeof(float), N*M, f3);
fclose(f3);
```

map(to:A[0:numElements ]={“data/vectorA.data”})

host memory is used as page bounce buffer

map(from:C[0:numElements ]={“data/vectorC.data”})
Prototype implementation for the runtime

Example 3: image data: Image Smoothing on GPU

```cpp
uchar* imgin_d, imgout_d, imgout_h;
uchar* gpu_filter(uchar*);
Mat image = cv::imread("image_in.jpg");
size_t img_size = input.ncols * input.nrows;
cudaMalloc(imgin_d, img_size);
cudaMalloc(imgout_d, img_size);
malloc(imgout, img_size);
// copy data HtoD
cudaMemcpy(imgin_d, image.data, img_size, cudaMemcpyHostToDevice);
// run GPU kernel
imgout_d = gpu_filter(imgin_d);
// copy dataDtoH
cudamemcpy(imgout_h, imgout_d, img_size, cudamemcpyDeviceToDevice);
// write result to a new file
image.data = imgout_h;
cv::imwrite("image_out.jpg", image);
```

map(to: imgin={jpeg: "image_in.jpg", metadata(host: meta_in)})

host memory is used as bulk bounce buffer

imgout={"image_out.jpg", metadata(host: meta_out)}
Prototype implementation for the runtime

Example 4: image data: Image Smoothing on GPU

```cpp
uchar* imgin, imgout;
uchar* gpu_filter(uchar*);
Mat image = cv::imread("image_in.jpg");
size_t img_size = input.ncols * input.nrows;
cudaMallocManaged(imgin, img_size);
cudaMallocManaged(imgout, img_size);
memcpy(imgin, image.data, img_size);
// run GPU kernel
imgout = gpu_filter(imgin);
// write result to a new file
image.data = imgout;
cv::imwrite("image_out.jpg", image);
```

**NOTE:** `memcpy` is still needed here since the data to be processed is preloaded to the host memory.

map(to: imgin={jpeg: "image_in.jpg", metadata(host: meta_in)})

host memory is used as page bounce buffer

imgout={jpeg: "image_out.jpg", metadata(host: meta_out)}

```cpp
```
Benefits

**image data**: smoothing, optimize paging to obtain higher performance for writing data back to storage.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Input</th>
<th>Output</th>
<th>HtoD</th>
<th>DtoH</th>
<th>Page Fault</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>3</td>
<td>274</td>
<td>0.066</td>
<td>0.061</td>
<td>2.294</td>
<td>2.499</td>
</tr>
<tr>
<td>512x1024</td>
<td>6</td>
<td>629</td>
<td>0.142</td>
<td>0.123</td>
<td>3.096</td>
<td>3.176</td>
</tr>
<tr>
<td>1024x1024</td>
<td>10</td>
<td>1285</td>
<td>0.338</td>
<td>0.706</td>
<td>2.718</td>
<td>3.491</td>
</tr>
<tr>
<td>1024x2048</td>
<td>20</td>
<td>2622</td>
<td>0.694</td>
<td>1.471</td>
<td>5.314</td>
<td>7.241</td>
</tr>
<tr>
<td>2048x2048</td>
<td>35</td>
<td>4833</td>
<td>5.289</td>
<td>3.545</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I**: Breakdown of execution time for image smoothing using global memory (ms)

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Input</th>
<th>Output</th>
<th>HtoD</th>
<th>DtoH</th>
<th>Page Fault</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>3</td>
<td>316</td>
<td>0.250</td>
<td>0.174</td>
<td>2.294</td>
<td>2.499</td>
</tr>
<tr>
<td>512x1024</td>
<td>6</td>
<td>660</td>
<td>0.303</td>
<td>0.239</td>
<td>3.096</td>
<td>3.176</td>
</tr>
<tr>
<td>1024x1024</td>
<td>10</td>
<td>1288</td>
<td>0.381</td>
<td>0.305</td>
<td>2.718</td>
<td>3.491</td>
</tr>
<tr>
<td>1024x2048</td>
<td>19</td>
<td>2637</td>
<td>0.813</td>
<td>0.600</td>
<td>5.314</td>
<td>7.241</td>
</tr>
<tr>
<td>2048x2048</td>
<td>37</td>
<td>4823</td>
<td>1.381</td>
<td>1.085</td>
<td>8.693</td>
<td>11.785</td>
</tr>
</tbody>
</table>

**TABLE II**: Breakdown of execution time for image smoothing using unified memory (ms)

DtoH ranges from about 30% to 150% of the kernel execution time.

Page Fault ranges from about 73% to 96% of the kernel execution time.
Future work

Optimization on data copy between storage and device:

- NVIDIA GPUDirect Storage - no bounce buffer at all
- cudaHostRegister with mmap - host to device, pinned memory applied
- Linux Direct Access(DAX) - storage to host, involving NVDIMM
Conclusion

- Offloading work flow shows that there should be less effort on loading data from storage to device in terms of programming
- Add optional elements to OpenMP `map` clause to get access to data copy from storage to device for users
- Two implementation ideas to use host memory as bulk bounce buffer and page bounce buffer
- Performance results show potential for further optimization
Acknowledgement

- This material is based upon work supported by the National Science Foundation under Grant No. 1833332 and 1652732.
- MCHPC’19 Committee
Thank you!

Questions?