Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory

Delaram Talaashrafi [1], Marc Moreno Maza [1], Johannes Doerfert [2]
[1]Western University, [2]Lawrence Livermore National Laboratory

Overview and Background (1/3)

OpenMP supported **device offloading** since version 4.0.
LLVM/Clang supports programs with offloaded regions since version 11.

- Clang's approach for OpenMP programs is **outlining**
- more info: "Compiler view of OpenMP"
  https://www.youtube.com/watch?v=eIMpgez61r4

`OpenMPOpt` **pass** implements several OpenMP GPU optimization techniques

- Inter-procedural LLVM pass,
- more info: 2021 LLVM Dev Mtg "Optimizing OpenMP GPU Execution in LLVM"
  https://www.youtube.com/watch?v=4EPl7De3cKg.

Overview and Background (2/3)

Global memory of a GPU that is accessible by all (blocks) teams,

- it is off-chip, with high access latency,
- it is desired to reduce the number of transactions to/from global memory.

An alternative is to use the **shared memory**,

- it is a limited, on-chip space, allocated for each team
- by prefetching data to the shared memory we can reduce access to the global memory.

Overview and Background (3/3)

Our goal is:

- **prefetch** some of the global locations into the shared buffer automatically,
- improves performance as each original load is **faster served from shared memory**.

We achieve this by:

- program's IR manipulation at compile-time,
  - LLVM `SCEV`, and
  - the infrastructure provide by `OpenMPOpt`.

Problem Statement

```
#pragma omp target teams map(to:v1[0:N*M])
   #pragma omp distribute parallel for
      // work-sharing loop
      for (int i=0; i<N; i++)
         for(int j=0; j<N; j++)
            // access loop
            for(int k=0; k<M; k++)
           // eligible access for prefetching
               sum += v1[i*M+k] * 3;
```

An array that is **read in a loop** in the parallel region.

- read access is in a loop-nest of the depth of at least two.
- the outer-most loop is work-sharing.
- the access relation is a function of IVs of work sharing loop and access loop.

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop’s chunks of each team,

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop's chunks of each team,

```
//distribute parallel
for(i=0; i<6; i++)
   for(j=0; j<2; j++)
      ... = A[i][j];
```

a00	a01
a10	a11
a20	a21
a30	a31
a40	a41
a50	a51

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop's chunks of each team,
  - computed by runtime function
  - team `t` runs iterations from $\texttt{LB}_t$ to $\texttt{UB}_t$

```
//distribute parallel
for(i=0; i<6; i++)
   for(j=0; j<2; j++)
      ... = A[i][j];
```

a00	a01
a10	a11
a20	a21
a30	a31
a40	a41
a50	a51

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop's chunks of each team,
  - computed by runtime function
  - team t runs iterations from $\mathtt{LB}_t$ to $\mathtt{UB}_t$
- the memory locations accessed in each iteration
  - for iteration i:
    $(\mathtt{Base}_i + k \times \mathtt{Step}_i), 0 \leq k < \mathtt{Number}_i$

```
//distribute parallel
for(i=0; i<6; i++)
   for(j=0; j<2; j++)
      ... = A[i][j];
```

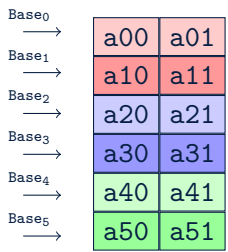
a00	a01
a10	a11
a20	a21
a30	a31
a40	a41
a50	a51

Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop's chunks of each team,
  - computed by runtime function
  - team $\texttt{t}$ runs iterations from $\texttt{LB}_\texttt{t}$ to $\texttt{UB}_\texttt{t}$
- the memory locations accessed in each iteration
  - for iteration $\texttt{i}$:
    $(\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step}_\texttt{i}), 0 \leq \texttt{k} < \texttt{Number}_\texttt{i}$

```
//distribute parallel
for(i=0; i<6; i++)
   for(j=0; j<2; j++)
      ... = A[i][j];
```



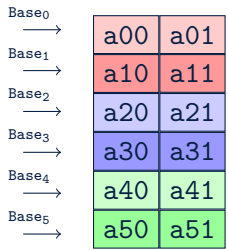
Algorithm (1/3)

To find global memory locations of the array, read by **each team**, we need:

- work-sharing loop's chunks of each team,
  - computed by runtime function
  - team $\texttt{t}$ runs iterations from $\texttt{LB}_\texttt{t}$ to $\texttt{UB}_\texttt{t}$
- the memory locations accessed in each iteration
  - for iteration $\texttt{i}$:
    $(\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step}_\texttt{i}), 0 \leq \texttt{k} < \texttt{Number}_\texttt{i}$

We get the parameters by **scalar evolution** analysis and inserting code in the kernel.

```
//distribute parallel
for(i=0; i<6; i++)
   for(j=0; j<2; j++)
      ... = A[i][j];
```



Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

The first location accessed in a team:

$$\texttt{A}[\texttt{Base}_{\texttt{i}}],\ \texttt{i} = \texttt{LB}_{\texttt{t}} \rightarrow \texttt{A\_sh}[0]$$

Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

The first location accessed in a team:

$$\texttt{A}[\texttt{Base}_{\texttt{i}}],\ \texttt{i} = \texttt{LB}_{\texttt{t}} \rightarrow \texttt{A\_sh}[0]$$

Locations accessed in the first iteration:

$$\texttt{A}[\texttt{Base}_{\texttt{LB}_{\texttt{t}}} + \texttt{k} \times \texttt{Number}],\ 0 \leq \texttt{k} < \texttt{Number} \rightarrow \texttt{A\_sh}[0] \ldots \texttt{A\_sh}[\texttt{Number}]$$

Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

The first location accessed in a team:

$$A[Base_i],\ i = LB_t \rightarrow A\_sh[0]$$

Locations accessed in the first iteration:

$$A[Base_{LB_t} + k \times Number],\ 0 \leq k < Number \rightarrow A\_sh[0] \ldots A\_sh[Number]$$

Locations accessed in the ith iteration:

$$A[Base_i + k \times Number],\ 0 \leq k < Number \rightarrow A\_sh[S_i] \ldots A\_sh[S_i + Number]$$

$$S_i = (i - LB_t) \times Number$$

Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

The first location accessed in a team:

$$\texttt{A}[\texttt{Base}_{\texttt{i}}],\ \texttt{i} = \texttt{LB}_{\texttt{t}} \rightarrow \texttt{A\_sh}[0]$$

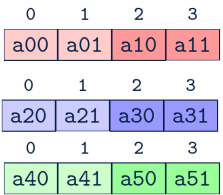
Locations accessed in the first iteration:

$$\texttt{A}[\texttt{Base}_{\texttt{LB}_{\texttt{t}}} + \texttt{k} \times \texttt{Number}],\ 0 \leq \texttt{k} < \texttt{Number} \rightarrow \texttt{A\_sh}[0] \ldots \texttt{A\_sh}[\texttt{Number}]$$

Locations accessed in the $\texttt{i}$th iteration:

$$\texttt{A}[\texttt{Base}_{\texttt{i}} + \texttt{k} \times \texttt{Number}],\ 0 \leq \texttt{k} < \texttt{Number} \rightarrow \texttt{A\_sh}[\texttt{S}_{\texttt{i}}] \ldots \texttt{A\_sh}[\texttt{S}_{\texttt{i}} + \texttt{Number}]$$

$$\texttt{S}_{\texttt{i}} = (\texttt{i} - \texttt{LB}_{\texttt{t}}) \times \texttt{Number}$$



Algorithm (2/3)

Prefetch locations of the global memory to **consecutive locations** in the shared memory.

The first location accessed in a team:

$$\texttt{A}[\texttt{Base}_{\texttt{i}}],\ \texttt{i} = \texttt{LB}_{\texttt{t}} \rightarrow \texttt{A\_sh}[0]$$

Locations accessed in the first iteration:

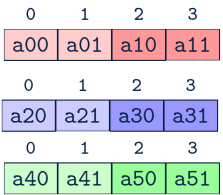
$$\texttt{A}[\texttt{Base}_{\texttt{LB}_{\texttt{t}}} + \texttt{k} \times \texttt{Number}],\ 0 \leq \texttt{k} < \texttt{Number} \rightarrow \texttt{A\_sh}[0] \ldots \texttt{A\_sh}[\texttt{Number}]$$

Locations accessed in the `i`th iteration:

$$\texttt{A}[\texttt{Base}_{\texttt{i}} + \texttt{k} \times \texttt{Number}],\ 0 \leq \texttt{k} < \texttt{Number} \rightarrow \texttt{A\_sh}[\texttt{S}_{\texttt{i}}] \ldots \texttt{A\_sh}[\texttt{S}_{\texttt{i}} + \texttt{Number}]$$

$$\texttt{S}_{\texttt{i}} = (\texttt{i} - \texttt{LB}_{\texttt{t}}) \times \texttt{Number}$$

We write a high-level function for these operation `copy_to_shared_mem`.



Algorithm (3/3)

The final step is to **replace** global memory accesses with the right shared memory accesses.

Algorithm (3/3)

The final step is to **replace** global memory accesses with the right shared memory accesses.

```
A[0][0] → A_sh[0]
A[0][1] → A_sh[1]
A[1][0] → A_sh[2]
A[1][1] → A_sh[3]
```

Algorithm (3/3)

The final step is to **replace** global memory accesses with the right shared memory accesses.

1- get the pointer to the dynamic shared memory,
2- generate code for shared memory accesses,
3- replace global accesses with the shared accesses.

```
A[0][0] → A_sh[0]
A[0][1] → A_sh[1]
A[1][0] → A_sh[2]
A[1][1] → A_sh[3]
```

Optimization

Bank conflict might happen when using GPU's shared memory.
In our kernels, the number of accesses with bank conflict depends on the value of `Number`.
To solve this problem, we use the **padding technique**.

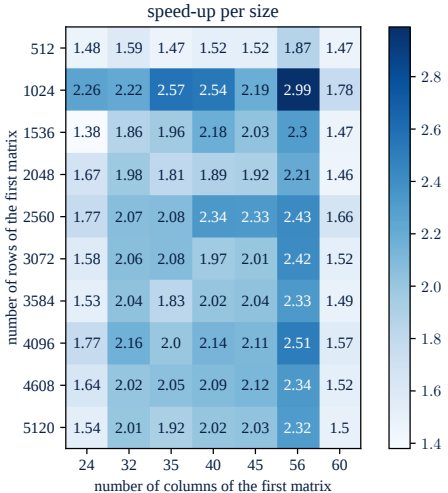
- if `Number` is a multiple of 32, we store one invalid data in the shared memory.

Evaluation

Prefetching speedup of the **matrix multiplication**, without padding.

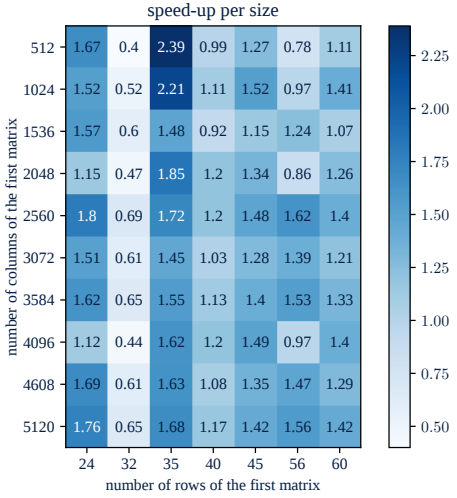
Evaluation

Prefetching speedup of the **matrix multiplication**, with padding.



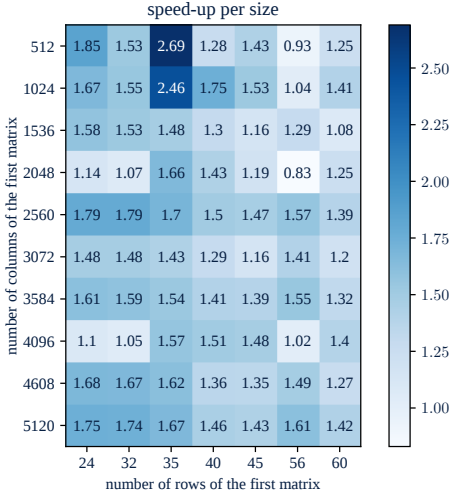
Evaluation

Prefetching speedup of the **matrix transpose multiplication**, without padding.



Evaluation

Prefetching speedup of the **matrix transpose multiplication**, with padding.



Thank You!