OMSCS-CS8803 Project-3: GPU Warp Scheduling

In this assignment, you will implement different warp scheduling algorithms on a GPU simulator which will give you an opportunity to learn how warp scheduling algorithms affect the performance of a GPU program.

Warp Scheduling in GPUs

Warp is a group of threads that execute the same instruction parallelly on different data. GPUs dynamically schedule warps to the available cores in each cycle in a fine-grained multi-threaded manner. Essentially, a GPU core has multiple warp contexts and in each cycle, the fetch stage of GPU fetches an instruction from a different warp. This helps the GPU to hide the latency of a long-running event such as memory access. The warp scheduler orchestrates the scheduling of warps on a core using various scheduling policies. An efficient scheduling algorithm ensures better utilization of resources, minimizing stall times and maximizing throughput.

Warp Scheduling Algorithms

Round Robin

In the round-robin scheduling algorithm, the scheduler cyclically schedules warps from the active warps pool (dispatch queue in macsim) each cycle.

- Ensures equal distribution of GPU time to all the warps.
- May cause more cache misses.

Greedy then Oldest

In greedy then oldest scheduling algorithm, the scheduler will pick the same warp every cycle until there is a long latency event, in which case, it switches to another warp.

- Reduces overall stall times.
- Better utilization of cache.

Simplified Cache-Conscious Wavefront Scheduling (CCWS)

In this algorithm, the number of scheduled warps is dynamically reduced to enhance cache locality. While decreasing the number of scheduled warps may impact thread-level parallelism, CCWS relies on a runtime monitoring mechanism to mitigate this effect.

Assignment Tasks

Task-1: Implement Greedy then Oldest Scheduler

A function outline for implementing the GTO scheduler is available in `core.cpp::schedule_warps_gto()`. This function is called each cycle by the scheduler (implemented in `core.cpp::run_a_cycle()` and `core.cpp::schedule_warps()`). To schedule a warp for execution using GTO your code must do the following things:

1. Check if the last scheduled warp is still in the active warps pool, if yes, schedule it again.
2. If not, search through all the warps in the active warps pool and schedule the oldest one.

> Hint: You must implement a per-warp timestamp marker and update it when the warp gets dispatched to the core.

> Hint: look for "// TODO:" comments!

Task-2: Simplified Cache-Conscious Wavefront Scheduler

In this task, you will implement the Cache-Conscious Wavefront Scheduling (CCWS) Algorithm under Round Robin.

The CCWS scheme tries to provide a warp with more exclusive access to the cache by limiting the number of warps that can be scheduled on the core. The goal of the scheme is to increase the intra-warp (aka wavefront) locality. Each warp is given a *Lost Locality Score* (LLS) based on how much intra-warp locality it has lost. The warps in the dispatch queue are then sorted in descending order of the LLS scores and the group of warps whose cumulative scores are within the *Cumulative LLS cutoff* (starting from the beginning) are allowed to be scheduled by the underlying scheduler (Round Robin in this case). the LLS scores are assigned to warps by using a feedback mechanism from the cache.

In our implementation, for each warp(`macsim.h::warp_s`), we keep an instance of the `ccws_vta.h::ccws_vta` class to stimulate that warp's VTA entry. VTA class simulates a fully-associative victim cache with LRU replacement policy and is used to store the evicted tags from L1 cache. We also use a variable in `macsim.h::warp_s` to keep track of a warp's LLS score and assign its value to the base LLS score when the warp is dispatched.

To implement CCWS your code must do the following things:

Setting up the feedback mechanism from the cache subsystem:

1. While repairing an L1 cache miss, a line from L1 cache could get evicted. In this case, we need to insert the tag corresponding to the evicted line address in corresponding warp's VTA entry. **[Task 2.1a and 2.1b]**

> In macsim, L1 cache insertion (and potentially eviction) happens in the following 2 cases:
>
> - **L1 Miss, L2 hit**: In this case L1 can evict a line when it repairs the miss by getting data from the l2 cache.
> - **L1 Miss, L2 Miss**: In this case, a memory request is sent and when the response is returned, the data gets inserted in both L1 and L2 caches.

2. Whenever there is a miss in the L1 cache, check if the tag is present in the VTA, if yes (VTA Hit!), increment the VTA hits counter and update the LLS score for the warp. **[Task 2.2a and 2.2b]** `LLS = ( VTA_Hits_Total * K_Throttle * Cum_LLS_Cutoff) / Num_Instr` where:
   - `Cum_LLS_Cutoff = Num_Active_Warps * Base_Locality_Score`
   - *VTA_Hits_Total* is the number of VTA hits across all warps on the core.
   - *K_Throttle* is the throttling parameter (value is given in `macsim.h`).
   - *Num_Active_Warps* is the number of warps in the dispatch queue.
   - *Base_Locality_Score* is the base locality score (value is given in `macsim.h`).

> Note: LLS score cannot be less than the Base LLS score for any warp at any point in time, you need to enforce this while updating the LLS scores.

3. In each cycle decrease the LLS scores by 1 for all the warps in the core (running, active, and suspended) until they reach the minimum value equal to the `Base_Locality_Score`. **[Task 2.3]**

Implementing CCWS Scheduler:
4. The outline for implementing the CCWS is available in `core.cpp::schedule_warps_ccws()`.

- Determine the Cumulative LLS cutoff value using: `Cum_LLS_Cutoff = Num_Active_Warps * Base_Locality_Score` **[Task 2.4a]**
- Construct the set of warps from the dispatch queue which are allowed to be scheduled (*scheduleable set*): **[Task 2.4b]**
  - Create a copy of the dispatch queue, and sort it in descending order.
  - Collect the the warps with highest LLS scores (until we reach the cumulative cutoff) to construct the *scheduleable warps* set.
- Use Round Robin as baseline scheduling logic to schedule warps from the dispatch queue only if the warp is present in the *scheduleable warps* set. **[Task 2.4c]**

You may refer to the CCWS paper (Specifically Section 3.3)
T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, BC, Canada, 2012, pp. 72-83, doi: 10.1109/MICRO.2012.16.

> Hint: look for "// TODO: Task 2.xx:" comments!

Task-3: Report

In this task, you'll write a report containing the following things:

- For Task-1:
  - A short explanation of your implementation.
  - A comparison of the performance of the round-robin scheduling algorithm with the GTO scheduling algorithm across provided benchmarks.
  - Any interesting things you observed (Optional).
- For Task-2:
  - A short explanation of your implementation.
  - A comparison of the performance of the round-robin, GTO, and CCWS scheduling algorithms across provided benchmarks.

> Note: Keep your report within 2 pages, and submit a PDF file.

> Report grades are typically assessed on a pass/fail basis, where submission guarantees full credit. Additionally, Reports may contribute bonus points to compensate for any potential loss in grades for Task 1 and Task 2.

Grading

- The assignment is worth **15 points** and is divided into 3 components, Task-1, Task-2 and a report.
- **0 points** if you don't submit anything by the deadline.
- **Task-1 is worth 6 points**, **Task-2 is worth 7 points**, and the **report is worth 2 points**.
- Rubric for Task-1 and Task-2:
    - **+50% points**: if you submit a code that compiles and runs without any errors/warnings but does **NOT** match the stats within the acceptable tolerance.
    - **+40% points**: if your stats match the reference stats within acceptable tolerance **(+-5%)**.
    - **+10% points**: if your stats exactly match the reference stats.

> For Task-1 We will use the *NUM_STALL_CYCLES* values from the logs to award points.

> For Task-2 We will use the *MISSES_PER_1000_INSTR* values from the logs to award points.

Submission Guidelines

- You must submit all the deliverables on GradeScope (Which can be accessed through the Canvas menu).
- **Deliverables**
    - A tar file containing all the code. Use the `make submit` command to generate the tar file.
    - A PDF report.

Macsim Simulator

This simulator is a stripped-down version of the original Macsim simulator developed by HpArch Lab at GT. It is a trace-based simulator where traces for various benchmarks are captured using NVBit. The traces contain information about warps and the instructions they execute.

Macsim Architecture

Macsim simulator consists of a trace reader (`trace.h`), several GPU cores (`core.cpp`), L1/L2 caches (`cache.cpp`) and a basic fixed latency memory (`ram.cpp`). Each core has a local L1 cache and all cores share an L2 cache. When we invoke macsim with a `kernel_config.txt` which contains the trace metadata, macsim retrieves information about how many kernels need to be executed and trace file for each warp. During simulation, macsim launches the kernels in a trace sequentially. In each kernel, macsim sets up the cores and caches.

A cycle in macsim is simulated in `macsim::run_a_cycle()` method which performs the following actions:

- Incrementing the global cycle counter (`macsim::m_cycle`).
- Call `core::run_a_cycle()` for each core,
- Check if any memory responses have been returned from the memory subsystem and send them to the corresponding core.

Each core mainly consists of an L1 cache, a dispatch queue (aka active warps pool, `core::c_dispatched_warps`), a suspended queue (`core::c_suspended_warps`), and a warp scheduler (`core.cpp::schedule_warps`). Since we are only concerned about the scheduling of warps and performance of different warp scheduling algorithms in terms of parameters like number of stall cycles, cache hits, etc. We only simulate load-store instructions and do not simulate any compute operations (like add, multiply etc.).

Warp Scheduling in Macsim

Macsim uses a 2-level warp scheduling scheme as discussed in the lecture. The warp scheduler in a core is responsible to schedule one warp from the dispatch queue every cycle. The warp runs for exactly one cycle before going back to the dispatch queue (only 1 instruction is executed). If the instruction is a load/store instruction, we send a read/write request to the memory subsystem starting at the local L1 cache. In this case, instead of going back to the dispatch queue, it is moved to the suspended queue and it stays there until the response comes back from the memory subsystem. Upon receiving a response, the warp is moved to the dispatch queue again in the background.

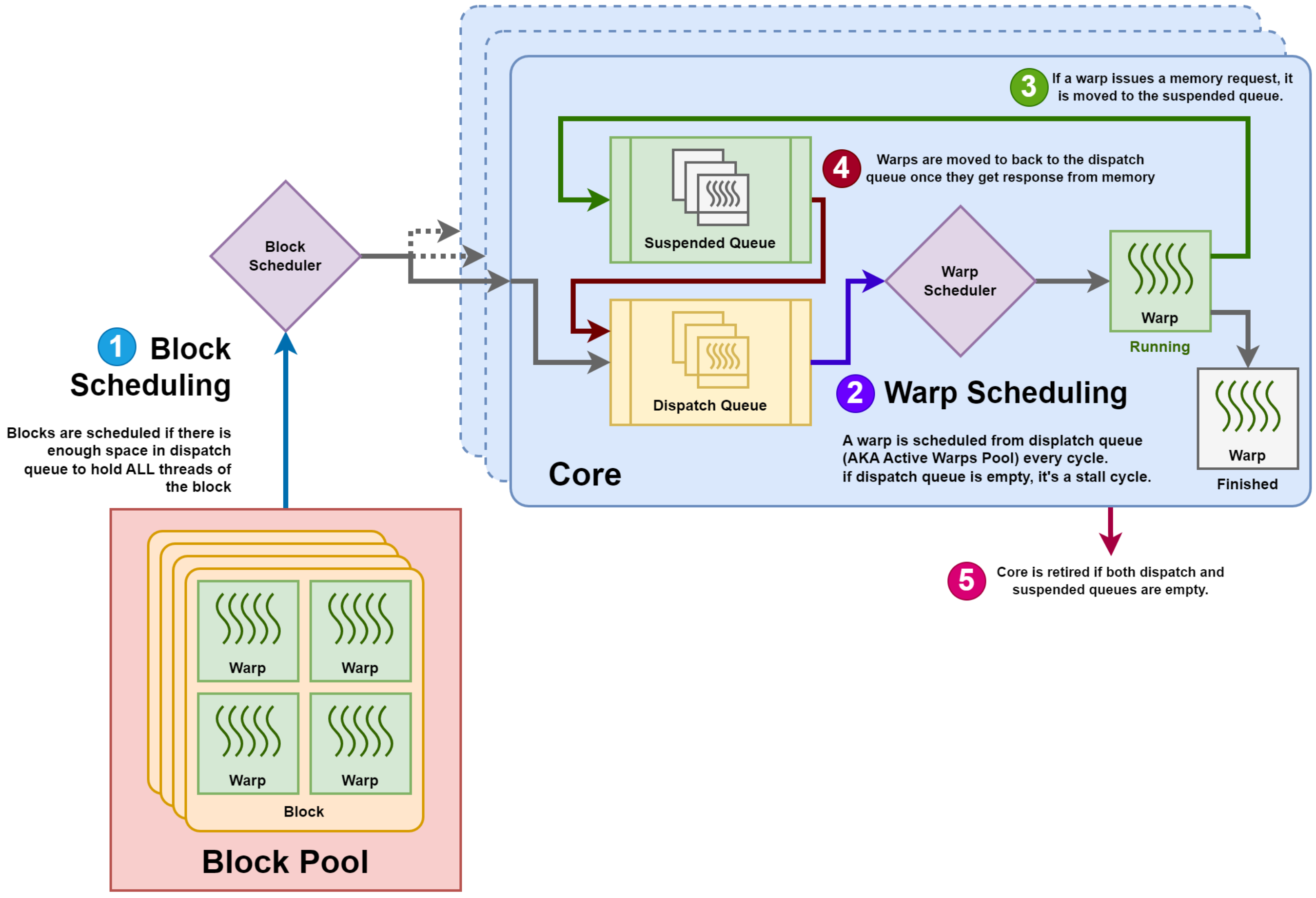
Therefore in summary, when the `core::run_a_cycle()` method is called, it performs the following actions:

- Checking if we got any responses from memory and moving the corresponding warps from suspended_queue to dispatch queue.
- Moving currently executing warp back to the dispatch queue.
- If there are no warps in both the dispatch queue and the suspended queue, then the core calls the `macsim::dispatch_warps()` method to refill its dispatch queue.
- Scheduling a new warp from the dispatch queue using a warp scheduling policy.
- Executing 1 instruction for the scheduled warp.

A cycle is called a **stall cycle** if the warp scheduler fails to schedule a new warp (perhaps because all warps are in the suspended queue waiting for memory response).

When the `macsim::dispatch_warps()` method is called and there are no warps to be dispatched (i.e. Block pool is empty), the core **retires**. The simulation ends when all the cores retire after executing the last kernel in the trace.

The following figure shows the overall simulation flow of Macsim.


3 If a warp issues a memory request, it is moved to the suspended queue.
4 Warps are moved to back to the dispatch queue once they get response from memory
Suspended Queue
Block Scheduler
Warp Scheduler
Warp
Running
Dispatch Queue
1 Block Scheduling
Blocks are scheduled if there is enough space in dispatch queue to hold ALL threads of the block
2 Warp Scheduling
A warp is scheduled from displatch queue (AKA Active Warps Pool) every cycle. if dispatch queue is empty, it's a stall cycle.
Warp
Finished
Core
5 Core is retired if both dispatch and suspended queues are empty.
Warp
Warp
Warp
Warp
Block
Block Pool

Prerequisites

- Linux-based OS/WSL with python3 and GNU C/C++ compiler is required.
- **Using PACE-ICE cluster:**
  - **Using PACE-ICE cluster:**
    - We recommend using the following settings while allocating a machine on PACE cluster (Interactive Shell/Desktop):
      - Node Type: CPU (first avail)
      - Cores Per Node: 2
      - Memory Per Core: 2 GB
  - Traces are located at `/storage/ice-shared/cs8803o21/macsim_traces`. The simulator will automatically pick them up.
- **Using your own machine**:
  - After running `make traces` command, traces will be automatically downloaded to the local `macsim_traces` directory.
  - Macsim needs zlib to uncompress trace files. zlib and correspoding headers can be installed using `$ sudo apt install zlib1g zlib1g-dev`.

> Note: Traces are ~150MB in size and may take a couple of minutes to download and uncompress.

You can either clone the repository or download a zip from GitHub.

```
$ git clone <url>             # Clone the repo
$ cd Macsim_cs8803
```

Plotting graphs requires `matplotlib` package, we've provided a bash script that you can source to setup a local Python virtual environment.

```
$ source sourceme              # source the sourceme script to setup
environment
```

Build Instructions

```
$ make -j`nproc`               # Build Macsim
# OR
$ make DEBUG=1 -j`nproc`       # Build Macsim for debugging with GDB
```

Try `make help` to see what else the makefile can do!

Running benchmarks and plotting results

To run a single benchmark trace use the following command:

```
$ ./macsim -g <GPU config> -t <trace_path/kernel_config.txt>
```

- GPU configs are XML files that define parameters such as the number of cores, scheduling algorithm to use, etc. These are located under `xmls` directory. We've provided 2 XML configs which are identical except the warp scheduling algorithm they use.
- Traces are located under `/storage/ice-shared/cs8803o21/macsim_traces` on the PACE cluster. If you are using a local machine, these will be downloaded to the `macsim_traces` directory.

For your convenience, we've provided Make targets to run all benchmarks and plot the results.

```
$ make run_task1               # Run the simulator for task-1
$ make run_task2               # Run the simulator for task-2
$ make plot                    # Generate plots
```

The 1st and 2nd commands will run macsim for all the benchmarks and will generate log files in the `log` directory. The 3rd command will pick up logs and plot the stats in a bar graph (output in the `log` directory).

Collaboration/Plagiarism Policy

- Feel free to use Ed for doubts/discussions, but **DO NOT** share your code snippets or discuss any implementation details.
- You are not allowed to publicly post your solutions online. (such as on GitHub)
- All submitted code will be checked for plagiarism, any violators will receive a 0.

Additional Information

Using GDB

GDB is a powerful tool if you want to resolve issues with your code or get a better understanding of the control flow, especially while working with a new codebase. To use GDB, follow these steps:

```
$ make DEBUG=1                              # Compile the project with debug flags
$ gdb --args ./macsim <macsim arguments>  # invoke gdb
```

FAQ

Can we modify beyond the TODO session?

Yes, you can modify other parts, but please make sure your final outputs are not changed.