GSOHC: Global Synchronization Optimization for Heterogeneous Computing (Artifact)

Soumik Kumar Basu
Department of Computer Science and Engineering, IIT Hyderabad, India

Jyothi Vedurada
Department of Computer Science and Engineering, IIT Hyderabad, India

Abstract

The use of heterogeneous systems has become widespread and popular in the past decade with more than one type of processor, such as CPUs, GPUs (Graphics Processing Units), and FPGAs (Field Programmable Gate Arrays) etc. A wide range of applications use both CPU and GPU to leverage the benefits of their unique features and strengths. Therefore, collaborative computation between CPU and GPU is essential to achieve high program performance. However, poorly placed global synchronization barriers and synchronous memory transfers are the main bottlenecks to enhanced program performance, preventing CPU and GPU computations from overlapping.

Based on this observation, we propose a new optimization technique called *hetero-sync motion* that can relocate such barrier instructions to new locations, resulting in improved performance in CPU-GPU heterogeneous programs. Further, we propose GSOHC, a compiler analysis and optimization framework that automatically finds opportunities for hetero-sync motion in the input program and then performs code transformation to apply the optimization. Our static analysis is a context-sensitive, flow-sensitive inter-procedural data-flow analysis with three phases to identify the optimization opportunities precisely. We have implemented GSOHC using LLVM/Clang infrastructure. On A4000, P100 and A100 GPUs, our optimization achieves up to 1.8x, up to 1.9x and up to 1.9x speedups over baseline, respectively.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Frameworks; Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, Synchronization, CPU-GPU, Heterogeneous Computing, Parallelization

Digital Object Identifier 10.4230/DARTS.11.2.17

Funding *Soumik Kumar Basu*: Funded by SERB SRG.

Acknowledgements We thank Chinmay Rajesh Ingle for his valuable feedback during the artifact documentation process.

Related Article Soumik Kumar Basu and Jyothi Vedurada, "GSOHC: Global Synchronization Optimization in Heterogeneous Computing", in 39th European Conference on Object-Oriented Programming (ECOOP 2025), LIPIcs, Vol. 333, pp. 21:1–21:30, 2025.
`https://doi.org/10.4230/LIPIcs.ECOOP.2025.21`

Related Conference 39th European Conference on Object-Oriented Programming (ECOOP 2025), June 30–July 2, 2025, Bergen, Norway

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2025 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This section describes the claims (corresponding to RQ1–RQ3 in the paper) that are supported by this artifact, along with detailed instructions for reproducing the reported results. In RQ1 and RQ2, we present statistics related to the static analysis phase of our framework, including the number of lines of code analyzed, the time taken for the analysis, and the number of synchronization calls relocated. In RQ3, we report the speedup gains achieved by the automatically transformed benchmark programs compared to their baseline versions. Comprehensive, step-by-step instructions for reproducing these results are provided in Section C.

Further, to examine the optimization opportunities along with the corresponding source and target line numbers of synchronization statements, you may follow the steps described below. During the compilation process, the compiler generates a log file named `main.cu.txt` within each benchmark's directory. This file documents the results of the static analysis, recording the original line numbers of synchronization statements in the source code alongside their corresponding target line numbers after optimization. Since the recorded entries directly reference the source code's line numbers, users can systematically verify the correctness of the framework by inspecting this log file. An illustrative example of the log file format is provided below:

```
====================================================
  Analysis Results for Module: [module name]
  Generated on: [date, time, year]
====================================================

SUMMARY:
  Total CUDA Memcpy Operations: [number of memcpy operations]
  Total Code Lines Analyzed: [number of Lines (LOC)]
  Analysis Duration: [time] milliseconds
MEMCPY TO TARGET MAPPINGS:
----------------------------------------------------
MEMCPY LINE          | TARGET LINE
----------------------------------------------------
...                  | ...
----------------------------------------------------

End of Analysis Report
```

2 Content

The artifact package includes:

- Docker Image tar
  Name: `gsohc_artifact_image.tar`
- The Docker container contains
  - LLVM Project
    Name: `llvm-project`
    Purpose: Compiling the programs with the GSOHC custom pass
  - Source Code
    Location: `llvm-project/llvm/lib/Transforms/gsohc/*`
  - Benchmark Programs
    Location: `GSOHC_Benchmarks/HeCBench/*`, `GSOHC_Benchmarks/PolyBench/*`
  - Scripts
    Location: `unzip.sh`, `compile_all.py`, `test_all.py`, `plots.py`, `checker.sh`, `min_eval.py`, `build_llvm.sh`, `stat_run.py`
    Purpose: Automating different tasks

- ReadMe file
  Location: `ReadMe.pdf`
  Purpose: A markdown PDF with proper steps to evaluate the artifact.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at zenodo: `https://zenodo.org/records/15302892`.

4 Tested platforms

4.1 Pre-requisites

4.1.1 Hardware dependencies

- At least one NVIDIA GPU (with 4GiB GPU DRAM) with CUDA support
- CPU with x86 architecture
- At least 60 GiB free disk space on host machine
- Minimum 4 GiB CPU DRAM (preferred 16 GiB)

4.1.2 Software dependencies (Before you build Docker image)

To run the provided Docker image, the following prerequisites must be met:

- **NVIDIA CUDA Toolkit**: Ensure that you have nvidia-cuda-toolkit installed in your local machine. This is required to install the NVIDIA CUDA container. One can follow the official documentation [5] provided by NVIDIA to install nvidia-cuda-toolkit.
- **Docker Installation**: Ensure that Docker is installed on your local machine. Docker is required to build and run the containerized environment. One can install Docker easily by following the steps mentioned here [1].
- **NVIDIA Docker Support**: Install the NVIDIA Docker container (nvidia-docker) to enable the usage of GPU resources. To install the NVIDIA Docker container, you can follow the official installation guide provided by NVIDIA. Here [6] is the link to the official documentation.

4.1.3 Software dependencies (After you build Docker image)

We expect the following tools to be pre-installed in the Docker container:

- CUDA 11.0 or later
- CMake (version >= 3.20.0)
- Python (version >= 3.8)
- zlib (version >= 1.2.3.4)
- GNU Make (version 3.79 or 3.79.1)
- PyYAML (version >= 5.1)
- Linux-based OS (Ubuntu preferred)
- LLVM-14.x (for building the compiler optimizations)

One can verify whether these tools are installed within the container (or in the local system) using the provided `checker.sh`. For more details, please follow the Section A.

4.2 Tested Architectures

We tested the artifact using:

- Ubuntu 22.04 LTS, 64 × AMD EPYC CPU @ 2.20GHz with 128 GB RAM, NVIDIA RTX A4000 (16 GB)
- Debian GNU/Linux 11, 20 × Intel(R) Xeon(R) Silver CPU @ 2.20GHz with 189 GB RAM, NVIDIA P100 (16 GB)
- Ubuntu 22.04 LTS, 64 × Intel(R) Xeon(R) Gold CPU @ 3.50 GHz with 84 GB RAM, NVIDIA A100 (84 GB)

5 License

The artifact is available under the MIT License.

6 MD5 sum of the artifact

3ff143a2c39552543adbc232bb0dfad2

7 Size of the artifact

10.7 GiB

A Quick start guide

A.1 Known common issues

1. **Unintended Extra Spaces in CLI Commands**: A frequent issue encountered when directly copying and pasting commands from this document into the command-line interface (CLI) is the inadvertent inclusion of extra spaces. These spaces may cause errors during execution or unexpected behaviour. To avoid this issue, it is recommended to copy the commands from the provided `readMe.pdf` file, which ensures that only the intended text is copied without additional spaces.
2. **Unable to Start Docker Container: Error Response from Daemon**: Users may encounter the following error when attempting to start the provided Docker container: *Error response from daemon: could not select device driver "" with capabilities: [[gpu]].* This issue often arises when the NVIDIA Container Toolkit hasn't been properly initialized due to Docker not being restarted after the installation. To resolve this, restart Docker (after installing NVIDIA Container Toolkit) by running the following commands:

```
sudo apt-get update
sudo systemctl restart docker
```

A.2 Pre-evaluation steps

1. **Load the Docker Image**: Load the downloaded `tar` file `gsohc_artifact_image.tar` into a Docker image:

```
[sudo] docker load -i gsohc_artifact_image.tar
```

2. **Start the Docker Container**: Run the Docker file with GPU support (required for execution) by using the following command:

```
[sudo] docker run --gpus all --name gsohc -it gsohc_artifact
```

3. **Unzip and Build**: The next step is to unzip the `llvm-project.zip` and build the `llvm-project` from scratch. Use the following command to automate this:

   ```
   pyhton3 unzip.py
   ./build_llvm.sh
   ```

   The `build_llvm.sh` prompts the following:

   ```
   Enter the number of processes for parallel build:
   ```

   The user should provide a suitable number of threads to parallelize the building of llvm. Remember that, providing too many threads will occupy all the CPU processors and may result in a system crash; also, providing too few threads will make the building process longer (may take several hours). Therefore, we recommend providing approximately half the number of your machine's CPU cores.
4. **Check the Dependencies**: Once the container's Command Line Interface (CLI) is open, verify that the required dependencies are correctly set up by running the `checker.sh` script. Use the following commands to execute the script:

   ```
   chmod +x checker.sh
   ./checker.sh
   ```

   If the dependencies are properly installed, the output of running the `checker.sh` script should resemble the following:

   ```
    NVIDIA GPU detected: [Your GPU Card Model]
    CUDA version [CUDA Version] is installed.
    CPU architecture is x86_64.
    CMake version 3.22.1 is installed.
    Python version 3.10.12 is installed.
    zlib is installed.
    GNU Make version 4.3 is installed.
    PyYAML is installed.
    Checking LLVM version from the Git repository...
    LLVM version llvmorg-14.0.6 found in the git repository.
    Rechecking all dependencies...
    All dependencies are correctly installed!
   ```

B Selective run of benchmarks (Minimal evaluation)

To quickly check the correctness of GSOHC, one can either choose to compile and run a random small set of benchmarks or even compile and run any benchmark program of their choice. We have provided the following commands to perform a minimal evaluation of our artifact:

- To compile and run any random n benchmark programs:

  ```
  python3 min_eval.py --num [n]
  ```

- To compile and run a specific benchmark program

  ```
  python3 min_eval.py --one [name of the benchmark folder]
  ```

  For example, to compile and run `adam-cuda` benchmark, you need to execute the following command:

```
python3 min_eval.py --one adam-cuda
```

C Expected behaviour (Functional Badge)

To verify the functional badge, please follow the steps outlined below:

1. **Compile Benchmark Programs**: Once the `checker.sh` script completes successfully, the next step is to compile each benchmark program both with and without the GSOHC custom pass. To verify that the only difference between the baseline and optimized compilation statements is the inclusion of the GSOHC custom pass, review any `run.sh` file in the benchmark programs. Specifically, the optimized code should include the following additional flag:`-flegacy-pass-manager -Xclang -load -Xclang .../lib/gsohc_opt.so`. Here, the flag word `gsohc_opt.so` refers to the shared object file generated during the build of the custom pass.
   Now, to compile all the provided benchmark programs, execute the following Python script:

   ```
   python3 compile_all.py
   ```

   This will invoke the Clang compiler to process the benchmark programs, producing both baseline and optimized versions of the code. The expected output should resemble the following:

   ```
   [SUCCESS] All benchmarks compiled successfully!
   ```

   This confirms that the GSOHC custom pass has successfully transformed each benchmark program.
   Approximate Time: at least 1 hour
2. **Run Benchmarks**: Execute the **test_all.py** script to run all baseline and optimized executables a total of eight times:

   ```
   python3 test_all.py
   ```

   The expected CLI output should be as follows:

   ```
   [SUCCESS] All benchmarks completed successfully!
   ```

   The first three executions will serve as warm-up runs. The script will automatically compute the average execution time of the last five runs and store the result in a CSV file. Each benchmark folder will contain a **benchmark_results.csv** file, which will record the following data:
   - Benchmark name
   - Baseline execution time
   - Optimized execution time
   - Speedup factor

   Approximate Time: at least 3-4 hours
3. **Plot the Bar chart**: After executing you should run the following command to consolidate all the results into a single CSV file, and to create an image of the bar charts.

   ```
   python3 plots.py
   ```

   The speed-ups should be available as a bar chart in the `benchmark_speedups.png` figure.
4. **Static Analysis Table**: You can run the following command to automatically generate the statistics of GSOHC's static analysis corresponding to each benchmark program.

```
python3 stat_run.py
```

The statistics will be available in CSV format as well as an image format in `analysis_report.csv` and in `analysis_report.png` files respectively.

5. **View bar chart and static analysis table:** To view the bar chart and the table generated by the artifact, you need to open another terminal. Note that the Docker container on the previous terminal must not be closed. Run the following command:

```
[sudo] docker cp gsohc:/workspace/benchmark_speedups.png [path/to/host]
[sudo] docker cp gsohc:/workspace/analysis_report.png [path/to/host]
```

D Re-use scenario (Reusable Badge)

Our codes are made open source and we have built GSOHC using the help of an open-source compiler framework [4]. Further, the benchmarks used to evaluate our framework are also open-source [2, 3].

References

1 Docker. Docker installation, November 2024. URL: `https://docs.docker.com/engine/install/`.

2 Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012. `doi:10.1109/InPar.2012.6339595`.

3 Zheming Jin and Jeffrey S. Vetter. A benchmark suite for improving performance portability of the sycl programming model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327, 2023. `doi:10.1109/ISPASS57527.2023.00041`.

4 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

5 NVIDIA. Nvidia cuda toolkit. URL: `https://developer.nvidia.com/cuda-downloads`.

6 NVIDIA. Nvidia docker installation, 2025. URL: `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html`.