* Introduction.

Backpropagation, a fundamental strategy employed in contemporary artificial intelligence (AI) model training, plays a pivotal role in enhancing the accuracy and effectiveness of neural networks. In typical AI models, comprising multiple layers, each with numerous neurons, the input undergoes a feedforward process, resulting in the generation of corresponding outputs. Following this phase, the model's predictions are compared with the ground truth labels, prompting adjustments to the weights and biases associated with each layer of neurons. This iterative training process ultimately yields a neural network capable of performing classification or generating meaningful outputs given unknown inputs.

While our interest in the AI field has led us to leverage existing packages for model training, our current endeavor focuses on a deeper exploration of backpropagation through CUDA code.The primary objective is to understand how to implement backpropagation using CUDA and explore potential avenues that are best for its performance.

* Methodology.

Our primary objective was to identify optimal settings that minimize execution time. To achieve this, our exploration focused on two key dimensions: varying data types and adopting diverse strategies for CUDA memory access. Through systematic investigation and analysis, we aimed to uncover configurations that would significantly enhance the overall performance of the benchmark, providing valuable insights into the interplay between data types and CUDA memory access methodologies.

*Datatypes*

Reducing data precision can be beneficial for optimizing memory usage and accelerating computations. By employing lower precision data types, such as fp8, half, and bfloat16, we can achieve faster data transfers between CPU and GPU, reduce memory footprint, and accelerate arithmetic operations. These data types sacrifice some level of numerical precision in exchange for improved performance, making them particularly suitable for deep learning tasks where a slight loss in precision is acceptable.

* fp8 (8-bit Floating Point):

fp8 represents an 8-bit floating-point data type in CUDA. Although it offers lower precision compared to the standard single-precision (fp32) or double-precision (fp64) formats, fp8 can significantly reduce memory requirements and enhance the throughput of parallel computations, making it suitable for applications where fine-grained precision is not critical.

* half (16-bit Floating Point):

The half data type in CUDA is a 16-bit floating-point format. It strikes a balance between reduced memory usage and sufficient precision for many deep learning tasks. Utilizing half precision can accelerate neural network training and inference while maintaining a reasonable level of numerical accuracy.

* bfloat16 (Brain Floating Point 16):

bfloat16 is a 16-bit floating-point format optimized for machine learning workloads, such as backpropagation in neural networks. It retains the advantages of reduced memory footprint and faster computation, making it suitable for tasks where the benefits of lower precision outweigh the potential loss of precision in certain applications.

*CUDA Memory Access*

In our exploration of CUDA memory access, we investigated three distinct memory allocation strategies: pageable memory, pinned memory, and unified memory.

* Pageable Memory:

Pageable memory represents the fundamental memory allocation method in CUDA. This approach involves initially allocating memory on the host side and subsequently utilizing cudaMemcpy to transfer the data to the device side. While pageable memory is easy to use and widely supported, it may introduce overhead due to the necessity of copying data between the host and device.

* Pinned Memory:

Pinned memory circumvents the data copying process on the host side, aiming to enhance performance. By doing so, it eliminates the need for data transfers between host and device during kernel execution. However, a notable drawback is that pinned memory occupies a significant portion of the limited host memory, potentially leading to resource contention and limiting the overall scalability of applications.

* Unified Memory:

Unified memory employs a page table maintained on both the host and device sides. Data transfers occur on-demand, i.e., when the data is accessed, eliminating the need for explicit data copying between host and device. This method, often referred to as on-demand migration, streamlines memory management and provides a unified view of memory accessible from both the host and the device. Unified memory simplifies programming by automatically handling data migrations, but careful consideration is necessary to optimize performance, especially for applications with specific memory access patterns.

Our experimental procedure involved the following steps:

1. Initial Execution and Profiling: We commenced by executing the existing program and subsequently analyzed the timeline data using tools such as nvprof and the NVIDIA Visual Profiler. This step aimed to identify potential bottlenecks within the implemented backpropagation algorithm.

2. Runtime Comparison Across Data Types: A comparative analysis of the program's running time was conducted, specifically focusing on variations in data types. This step provided insights into the impact of different data types on overall execution efficiency.

3. Memory Allocation Optimization: To enhance memory allocation, we implemented various optimization techniques. These techniques included the adoption of unified memory and pinned memory strategies, aimed at improving overall program performance by efficiently managing memory resources.

* Experimental Setup.
* Experimental Platform:

Our project utilized the Tesla T4 GPU, stationed in the Google Colaboratory environment, with cuda toolkit version 12.2.

* Tools for Profiling and Analysis:

We employed profiling and analysis tools provided by Nvidia, such as nvprof, nsight, and the NVIDIA Visual Profiler. These tools facilitated a thorough examination of outcomes and timelines across diverse experimental settings.

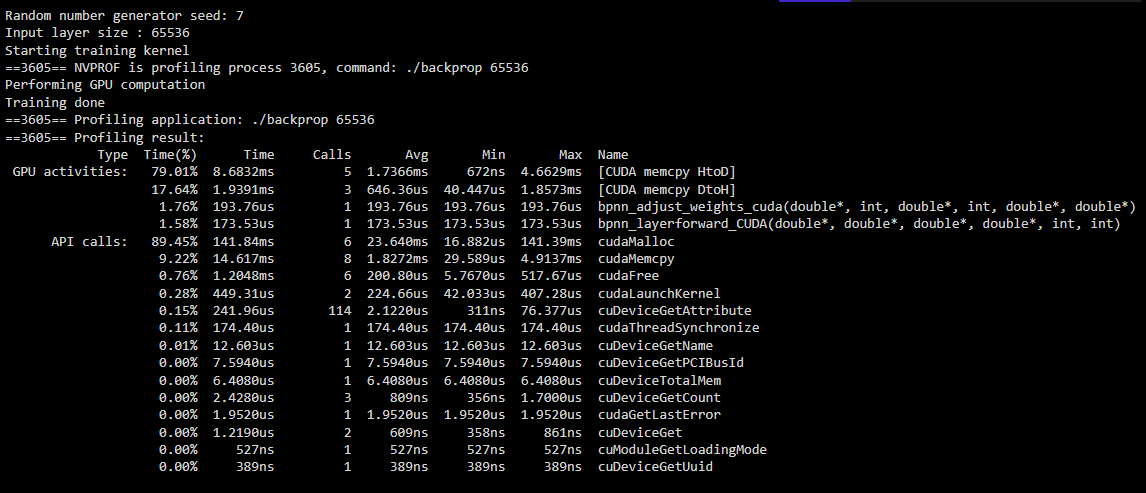
* Data Type Comparison:

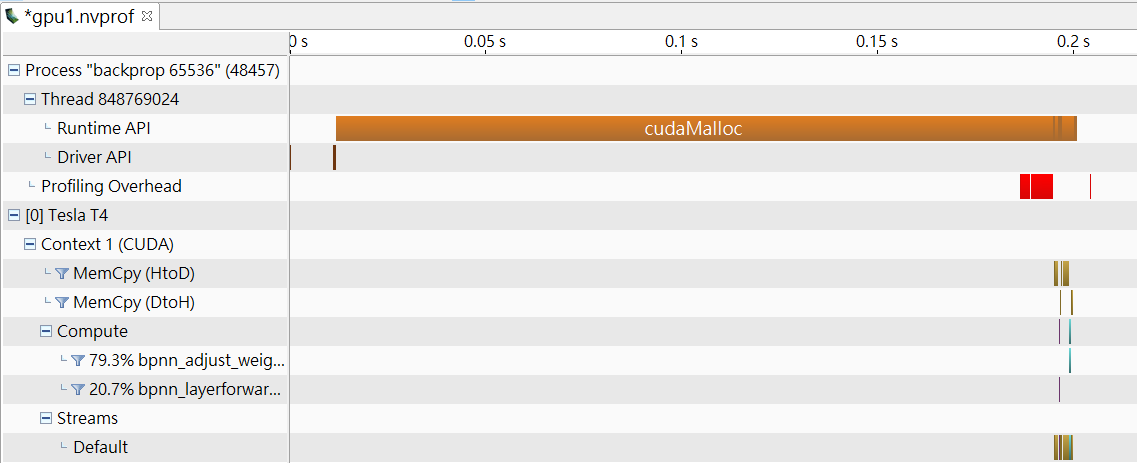
In comparing different data types, we adhered to the original setup, executing backpropagation training with a single epoch. This focused approach allowed for the evaluation of diverse data types and their impact on the process.

* Memory Allocation Comparison:

To assess memory allocation efficiency and pinpoint time-consuming segments (considering initialization costs and potential data reuse scenarios), we iterated the backpropagation process 1000 times. This method provided a more realistic analysis of memory efficiency under varied settings.

* Results and Discussion.

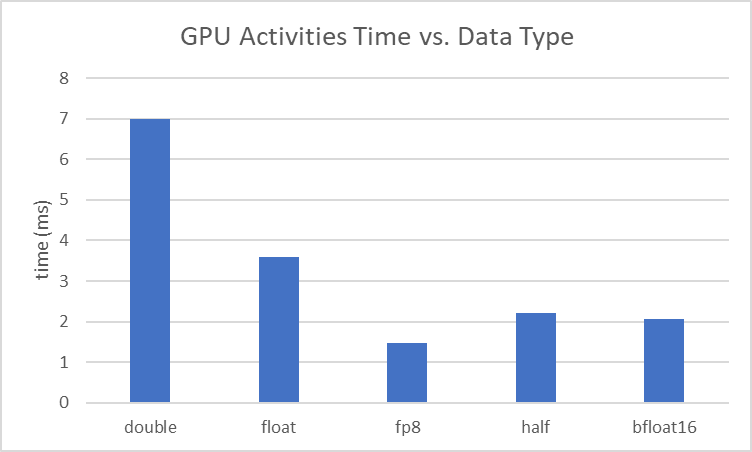




1. We initiated our analysis by examining the original backpropagation program within the Rodinia benchmark. Leveraging profiling tools such as nvprof and Nvidia Nsight, we gained valuable insights into the program's performance characteristics. Upon careful examination of the results, a key observation emerged: the memory allocation method was a critical factor influencing overall efficiency. This realization prompted us to prioritize the optimization of memory allocation strategies.

In tandem with this optimization effort, we drew inspiration from recent advancements in mixed precision learning (Micikevicius, Paulius, et al., 2017). Intrigued by the potential benefits of reduced precision data types in terms of both computational efficiency and memory consumption, we decided to explore their integration into the data storage and calculation processes. This dual focus on memory allocation optimization and the adoption of reduced precision data types forms the basis of our strategy to enhance the backpropagation program's overall performance.

2. Reduced Data Precision.



In our experimentation, we initiated the transition of data types from double to float, a straightforward process supported by both C and CUDA. Concurrently, we extended our implementation to incorporate three additional data types: fp8, half, and bfloat16. This augmentation involved converting float inputs into the desired data types and leveraging corresponding CUDA library functions for computation on the device. Notably, due to the absence of specific mathematical functions for fp8, we resorted to converting it back to float for calculations, followed by a subsequent conversion back to fp8.

Profiling the outcomes using tools like nvprof and Nsight, we observed a reduction in GPU activity times, encompassing both calculation and memory copying times. Although these reductions were relatively small when compared to the overall execution time, they were nevertheless discernible. Moreover, the memory occupation witnessed a substantial decrease, often halving or more in comparison to the original version.

In summary, while the impact of reduced data precision on the total execution time proved to be minor in our specific case, it significantly contributed to saving GPU execution time and memory. This optimization allows us to undertake backpropagation on more extensive datasets or extend training across additional epochs, showcasing the practical implications of the trade-off between precision and resource efficiency in the context of deep learning tasks.

3. Cuda Memory Allocation

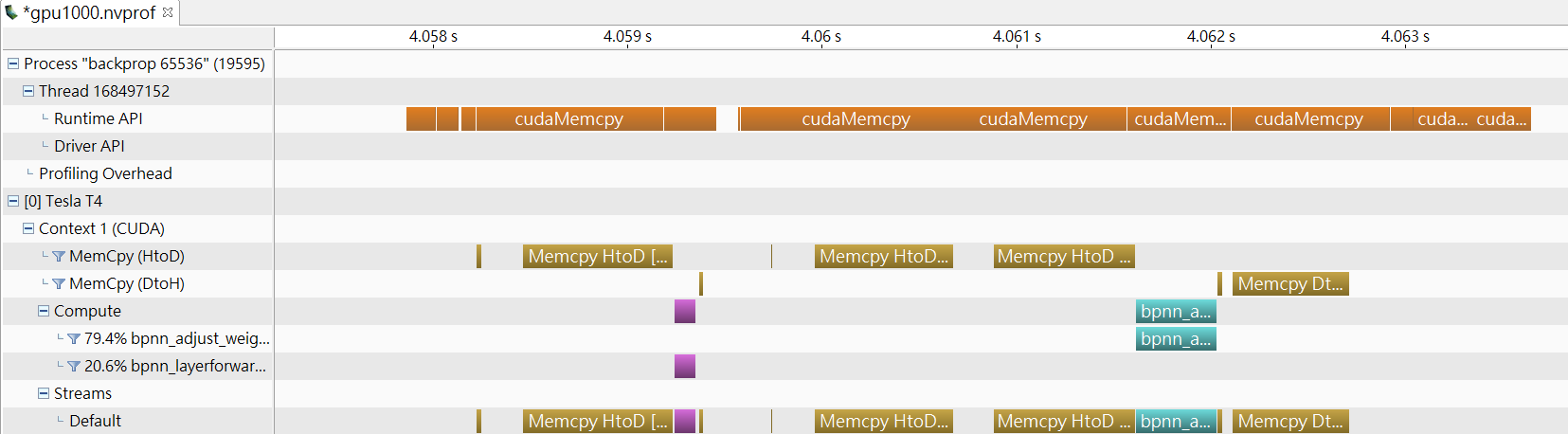
We use float to be our datatype in this setting, since it gains the best performance in the overall reduced data precision experiment. And 65536 is used as our input size.

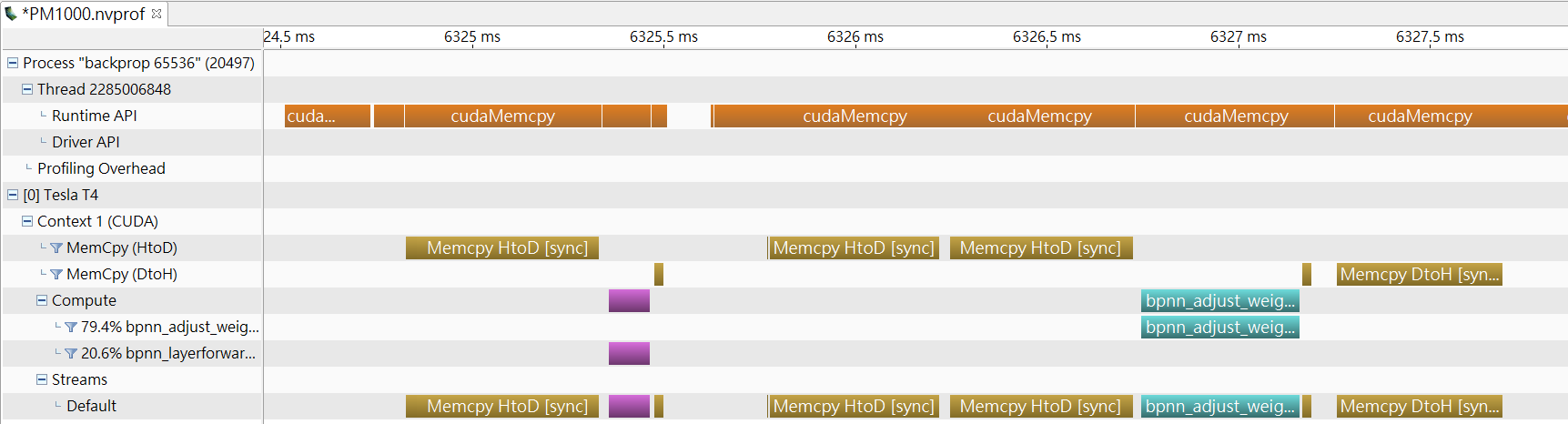
↓ Table 1. total execution time comparison between distinct memory allocation strategies (note that the *italics time* is the session time of data migration)

|  | Pageable memory | Pinned memory | Unified memory (w/o prefetch) | Unified memory (w/ prefetch) |
| --- | --- | --- | --- | --- |
| total execution time | 8.10s | 13.11s | 21.03s | 17.40s |
| kernel execution time | 519.67ms | 519.58ms | 5.60s | 948.38ms |
| Host to Device | 2.38s | 1.47s | *68.07ms* | *46.07ms* |
| Device to Host | 690.76ms | 487.69ms | *1.12s* | *802.25ms* |

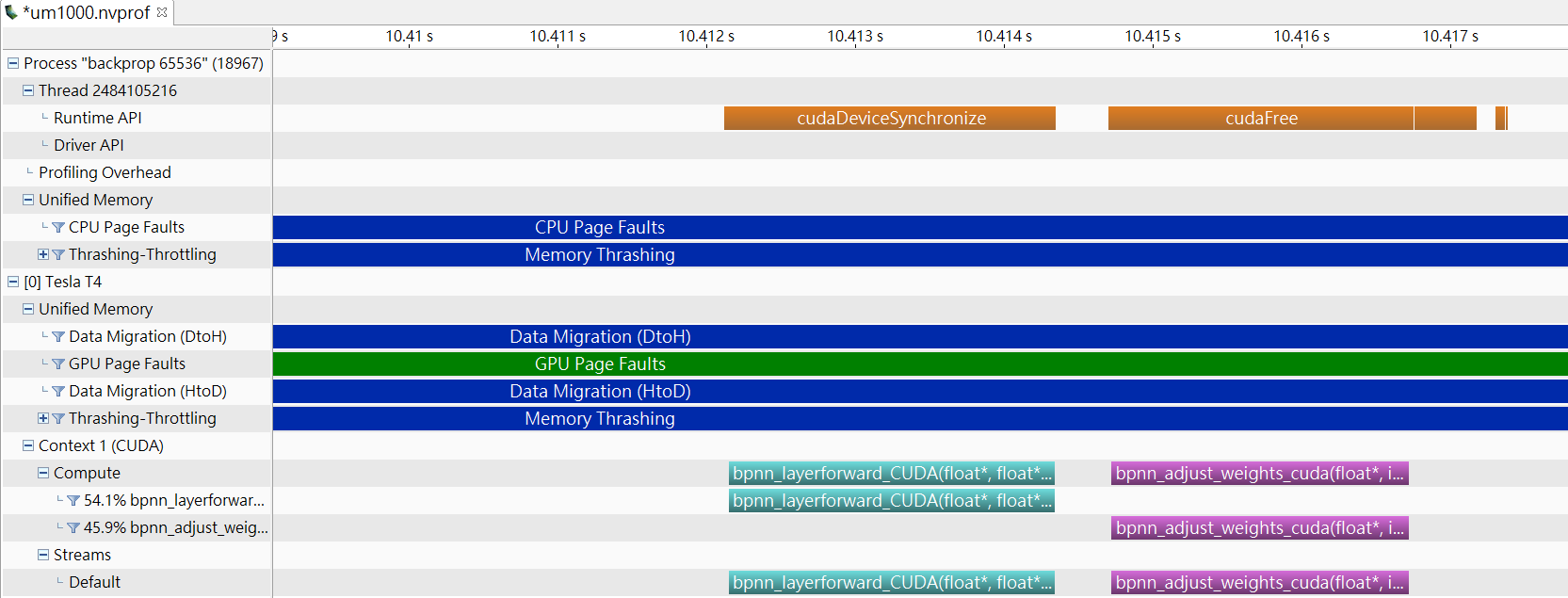
From the total execution time, we can find out that pageable memory allocation method takes the least time. To figure out the memory allocation situation, we provide the visualization timeline of each strategy below.

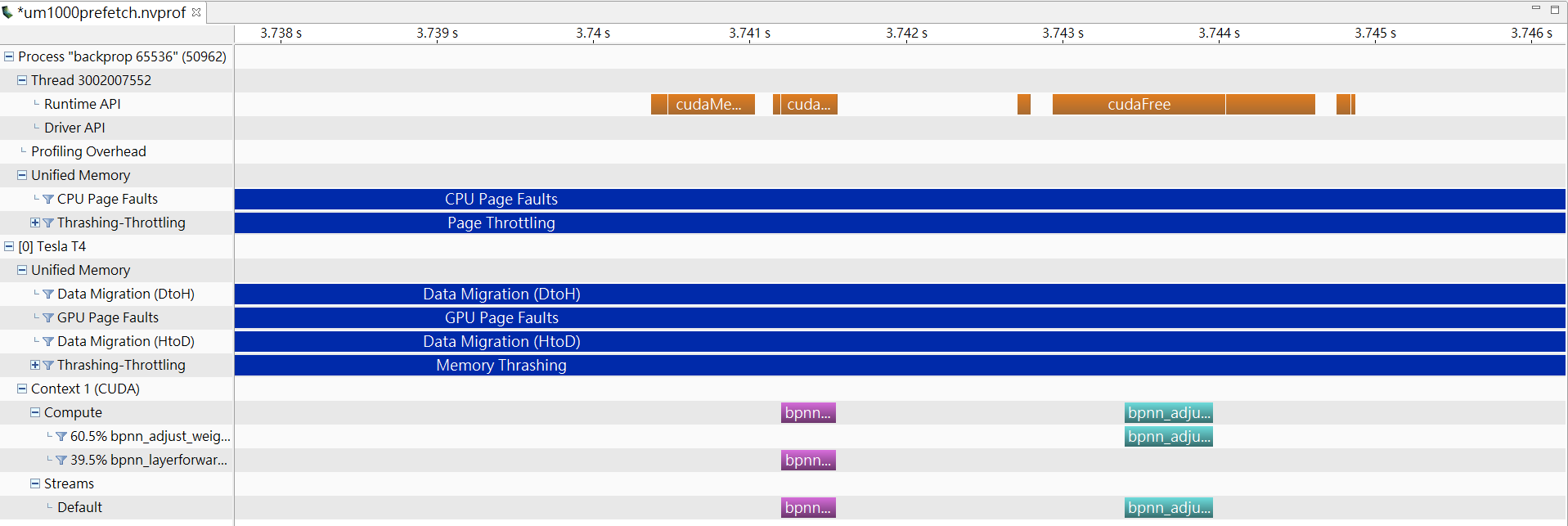
Visualization of timeline:

* Pageable memory
* Pinned memory



* Unified memory (w/o prefetch)



* Unified memory (w/ prefetch)

From the visualization analysis, it is evident that data transfer latency in pageable memory allocation is not improved by pinned memory. Also, the other memory allocation strategies introduce additional costs, such as the use of cudaHostAlloc in pinned memory. This method requires allocating contiguous physical pages, contributing to slower performance. Notably, frequent memory copies between the host and device are observed, and in the case of pinned memory, the possibility of uncached memory access in the main memory may lead to a slowdown in performance.

Unified memory, although providing a cleaner code without explicit data copies, is affected by page faults, constituting a substantial portion of execution time (represented by the green segment, exceeding 30% of total time). This factor significantly degrades the performance of unified memory. To address the page fault issue, we employed "cudaMemPrefetchAsync," resulting in a notable improvement in total execution time compared to the original version. However, it remains inferior to the pageable version, the possible reason might be due to the overhead introduced by the prefetching operation. While cudaMemPrefetchAsync helps mitigate page faults by prefetching data to the GPU in advance, it incurs an additional cost in terms of execution time and resource utilization. The prefetching process itself may introduce delays and synchronization overhead, partially offsetting the gains achieved in reducing page faults.

* Limitation and Conclusion.

In this project, we try to figure out which scene is best fit for the backpropagation provided in the rodina benchmark. The most optimal setting in datatype is float point while for memory allocation is pageable allocation. But it's essential that the performance of each memory allocation strategy and datatype are contingent on the specific requirements and characteristics of the application. The trade-offs between simplicity of code, data transfer efficiency, and overhead from memory management operations should be carefully considered in selecting the most suitable memory allocation strategy for a given GPU computing task.

Also, the backpropagation in the rodina benchmark doesn’t equip with the ability to train with the real world data, so the generated data and labels are all fake, which makes it hard to evaluate the accuracy of training. If we have more time, we would like to apply it with the real training situation and examine all the strategies we’ve mentioned to justify the overall performance.

* References.

1. S. Che *et al*., "Rodinia: A benchmark suite for heterogeneous computing," *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2009, pp. 44-54, doi: 10.1109/IISWC.2009.5306797.
2. Micikevicius, Paulius, et al. "Mixed precision training." *arXiv preprint arXiv:1710.03740* (2017).
3. [CUDA Memory Access: Global, Zero-Copy, Unified](https://migocpp.wordpress.com/2018/06/08/cuda-memory-access-global-zero-copy-unified/)
4. [Pinned Memory slower than pageable memory](https://forums.developer.nvidia.com/t/pinned-memory-slower-than-pageable-memory/18821)

* A summary of each group member's contributions.

We do all the work together.

* Github Links: https://github.com/LaiYuHong/DD2360HT23
* Expected Grade: B
* Group member: SHAO-YU WENG, YU-HONG LAI