Efficient Cache Coherence Mechanisms for Heterogeneous Environments

2230030 Toyohiro Misaki

GPU is one of the primary technologies to support the remarkable progress of AI in recent years. GPU has been known as fast parallel computing units for AI models and its improvement of processing power has contributed to that of AI. The improvement of GPU can be divided into performance enhancement as a computing unit and architectural enhancement as a heterogeneous computing platform. Then, looking closer at the architectural aspect, while data had to be copied from CPU memory to GPU memory in advance of GPU processing in earlier architectures, the enhancement has allowed users to share the memory between CPU and GPU at present. This resolves the performance bottleneck in transferring data between the memories and makes the heterogeneous environment achieve higher performance. On the other hand, this improvement has brought new issues such as cache coherence by sharing memory between CPU and GPU.

A survey article on this issue mainly explains that there are roughly two research areas: coherence and memory management, but the research papers and the articles did not propose their approaches based on memory consistency models in heterogeneous environments. In addition, these proposed approaches focused on how effectively they can absorb the differences in micro-architectures in heterogeneous environments and employed a conventional way of requesting coherence. This study focuses on memory consistency models in the heterogeneous environments and considers an efficient design for issuing coherence requests. To achieve these goals, we analyze user programs for CPU-GPU heterogeneous environments and extract two important characteristics. We exploit them and present two proposed approaches.

The first proposed approach is to leverage Release Consistency model in which Acquire and Release work as a sync operation. This comes from one of the characteristics extracted from the analysis of user programs. In these programs, CPU and GPU share the data with each other. Carefully examining how the shared data are accessed by CPU and GPU, we found that the data were accessed by CPU or GPU alternatively. This clearly demonstrates the exclusive access characteristic, which is that CPU and GPU do not concurrently access the shared data. This unidirectional data access is very distinctive from others and proves these environments are the best suit for the relaxed memory consistency model since we do not need to issue a coherence request every time

CPU or GPU writes data to the shared memory. Therefore, we employed this memory consistency model and implemented Acquire and Release functions in user programs.

The second proposed approach is to exploit the tendency in memory management through which the shared data are likely to be assigned consecutive memory addresses. This also comes from our user program analysis. In the heterogeneous environments, the shared data are prepared for GPU processing and the data structure is typically organized into large chunks such as arrays. We exploit this characteristic and propose a new design for issuing coherence requests. If the system finds that there is a range of consecutive memory addresses at sync operation, a directory controller sends a single coherence request to CPU/GPU cache controllers for invalidation. This coherence request includes multiple addresses for invalidation and the corresponding lines will be invalidated at each cache controller until the last memory address in the address range. This approach could save the overhead of processing coherence requests one by one and achieve the reduction of execution time spent on processing the coherence requests. The proposed approach implemented these mechanisms in the directory controller and each cache controller. This study proposes these approaches in the form of hardware mechanisms.

This study employs gem5 simulator for evaluation as there is no hardware with proposed functions at present and AMD provides its heterogeneous configurations to gem5 platform as sample codes. We customized these default configurations and implemented our proposed approaches. We also leveraged sample user programs provided for gem5 heterogeneous environments by AMD and implemented sync functions on Release Consistency model in these programs. We evaluated three user programs in the customized heterogeneous environment and examined the differences between our proposed approach and the default approach. One of the three programs is square.cpp. In this program, CPU prepares for two arrays and initializes these arrays with some data. GPU processes these arrays in parallel and calculates the square of each element in the arrays. After the calculation, CPU finally checks the calculated values to see if the results are correct. We went through this program by changing the number of elements in arrays. There are 7 variations of the number: 200, 2000, 20000, 40000, 100000, 200000, and 300000. We examined the number of invalidation requests (probes) sent from the directory controller to each cache controller at the sync points. We also examined the execution time (ticks) spent processing those probes during the sync. The results show that our proposed approach reduced the number of probes by about 70% compared to the default configuration in the case of 200 and about 95% in the case of 300000. The execution time shows that our proposed approach achieved a reduction in total ticks of approximately 48% and 70% for 200 and 300000 elements in arrays, respectively.

The second program is 2dshf.cpp and this program transposes a given matrix with _shfl function. There are 7 variations in matrix size: 4×4, 8×8, 16×16, 20×20, 30×30, 40×40, and 50×50. We achieved a reduction in the number of probes of approximately 30% and 70% for 4×4 and 50×50 for the matrix size, respectively and a reduction in total ticks of approximately 5% to 10% for

all other sizes.

The last program is MatrixTranspose.cpp and this is similar to 2dshf.cpp. Both programs transpose given matrices but are different in size. In Matrix-Transpose.cpp, there are 7 variations in size: 16×16, 32×32, 64×64, 128×128, 256×256, 384×384, and 512×512. The results show that our proposed approach reduced the number of probes by about 40% compared to the default configuration in the case of 16×16 and about 90% in the case of 512×512. The execution time shows that our proposed approach achieved a reduction in total ticks of approximately 40% for 16×16 and 60% for all other sizes.

In these evaluations, we focused on differences in the number of coherence requests (probes) and the execution time (ticks) spent processing these probes. Our proposed approach proved better performance in all three scenarios.