Accelerating Simulation of Agent-Based Models on Heterogeneous Architectures

Jin Wang[†], Norman Rubin[‡ *], Haicheng Wu[†], Sudhakar Yalamanchili[†]
[†]Georgia Institute of Technology, [‡]Advanced Micro Devices
jin.wang@gatech.edu, nrubin@nvidia.com, hwu36@gatech.edu, sudha@ece.gatech.edu

ABSTRACT

The wide usage of GPGPU programming models and compiler techniques enables the optimization of data-parallel programs on commodity GPUs. However, mapping GPGPU applications running on discrete parts to emerging integrated heterogeneous architectures such as the AMD Fusion APU and Intel Sandy/Ivy bridge with the CPU and the GPU on the same die has not been well studied.

Classic time-step simulation applications represented by agent-based models have the intrinsic parallel structure that is a good fit for GPGPU architectures. However, when mapping these applications directly to the integrated GPUs, the performance may degrade due to less computation units and lower clock speed.

This paper proposes an optimization to the GPGPU implementation of the agent-based model and illustrates it in the traffic simulation example. The optimization adapts the algorithm by moving part of the workload to the CPU to leverage the integrated architecture and the on-chip memory bus which is faster than the PCIe bus that connects the discrete GPU and the host. The experiments on discrete AMD Radeon GPU and AMD Fusion APU demonstrate that the optimization can achieve **1.08–2.71x** performance speedup on the integrated architecture over the discrete platform.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Heterogeneous (hybrid) systems; I.6.3 [**Computing Methodologies**]: Simulation And Modeling—*Applications*

General Terms

Algorithms, Performance

Keywords

GPGPU, APU, Agent-Based Model, Traffic Simulation

1. INTRODUCTION

The transition to heterogeneous computing has been accompanied by the advance of general-purpose graphics processing unit (GPGPU) technologies. While the discrete GPU architectures such as AMD Southern Island [4] and NVIDIA Fermi [18] have demonstrated their successes both in industry and academia, commodity heterogeneous platforms are moving towards integrated architectures represented by AMD Fusion APUs [2], Intel Sandy Bridge [11] and NVIDIA Echelon [13] each of which have the CPU and the GPU on the same die. The new design methodology, especially the CPU-GPU shared memory hierarchy, introduces challenges to the programming models, compiler techniques, application algorithms, software stacks and power management.

Specifically, mapping traditional GPGPU applications that are running on discrete platforms onto integrated architectures and understanding the impact of architectural difference on algorithm design are increasingly important yet not well studied. The implementations of these applications generally rely on the GPUs as the computation accelerators and limit the CPUs to the role of preparing and transferring the data to the GPUs. The large memory access overhead introduced by the slow PCIe bus that connects the CPU and the GPU precludes the possibility of splitting the computation between them. On the other hand, the integrated architectures have more closely coupled CPU and GPU connected by the on-chip memory bus with higher bandwidth, which enable a new implementation philosophy that can extract more computation power from the CPUs by assigning workloads other than data transfer.

The agent-based models belong to a classic group of problem sets that use time-step simulation to predict or recreate the complex environment behaviors. The parallel implementations of the agent-based models on GPUs have been a recent trend with significant performance improvement reported from the serial counterpart on the CPUs especially for extra-large scale simulations[22, 16, 1]. However, when mapping to integrated architectures, the algorithm needs to be properly adapted to leverage both the CPU computational capability and the memory hierarchy.

In this paper we report an implementation and optimization of an agent-based model on heterogeneous architectures. The experiments demonstrate that the integrated architecture can achieve **1.08–2.71x** speedup depending on the problem size compared to the seemingly faster discrete platform because of both the use of the CPU and the higher bandwidth provided by the on-chip memory bus. Specifically, this paper makes the following main contributions:

- An optimized massively parallel implementation that can run the agent-based model applications on either the discrete or the integrated GPUs.
- An optimization to the above implementation that moves a portion of computation to the CPU to leverage the

*The author is now affiliated with NVIDIA Corp.

shared memory hierarchy of the integrated architectures.

- A systematic performance evaluation and comparison of the implementations on different architectures with emphasis on understanding how to efficiently use integrated CPU-GPU architectures.

The rest of the paper is organized as follows. Section 2 introduces the discrete GPU and the integrated heterogeneous architecture represented by the AMD GPU and Fusion APU, as well as the agent-based model and the traffic simulation used in this paper. Section 3 describes the massively parallel GPU implementation for the traffic simulation. Section 4 proposes an adaption of the implementation for heterogeneous architectures to utilize the computation power from both the CPU and the GPU. Section 5 evaluates and compares the performance on different architectures. Section 6 reviews the related works, followed by the conclusion in Section 7.

2. BACKGROUND

2.1 AMD GPU and Fusion Architecture

This paper targets both the discrete and integrated heterogeneous architectures. The discrete architectures have separate CPUs and GPUs in the system where the GPU architectures are represented by NVIDIA's Fermi [18] as well as AMD's Evergreen, Northern Island and Southern Island [3, 4]. The integrated architectures put the CPU and the GPU on the same die, represented by NVIDIA Echelon [13], Intel Sandy Bridge [11] and AMD Fusion APUs [2].

We specifically use the AMD GPU and Fusion APU in this paper. This section presents an overview of their architectures and introduces the differences in the memory hierarchy. AMD devices use OpenCL [14] as the programming model. Accordingly, this paper adopts OpenCL terminology, where a work-item and a work-group correspond to a thread and a Cooperative Thread Array (CTA) in the Bulk Synchronous Parallel (BSP) model [24] respectively, and the local memory is the memory region that can be accessed and shared only by the work-items in a work-group. However, the methodology and the conclusions also apply to other heterogeneous architectures or programming models [10].

Figure 1 shows the typical AMD GPU architecture, where Figure 1a represents the general architecture for both the Evergreen and Northern Island Devices and Figure 1b shows an overview of the Southern Island Device, also known as the Graphics Core Next (GCN) architecture. These GPU devices comprise several compute units, the number and the structure of which vary with the device family. In Evergreen or Northern Island GPUs, a compute unit has multiple processing elements, each of which executing a work-item. The processing element possesses ALUs either in 4-way VLIW (Northern Island) or 5-way VLIW (Evergreen). In the Southern Island Devices, a compute unit comprises a scalar unit and a vector unit with 4 16-lane SIMDs. The Local Data Share (LDS) memory stores the data shared by the work-group and is used as the local memory in OpenCL. The AMD Radeon HD7950 GPU used in this paper belongs to the Southern Island device family. The AMD A10-5800K APU possesses the Radeon HD7660D GPU which has the Northern architecture. It should be noted that on A10-5800K, the L2 cache in Figure 1a does not exist.

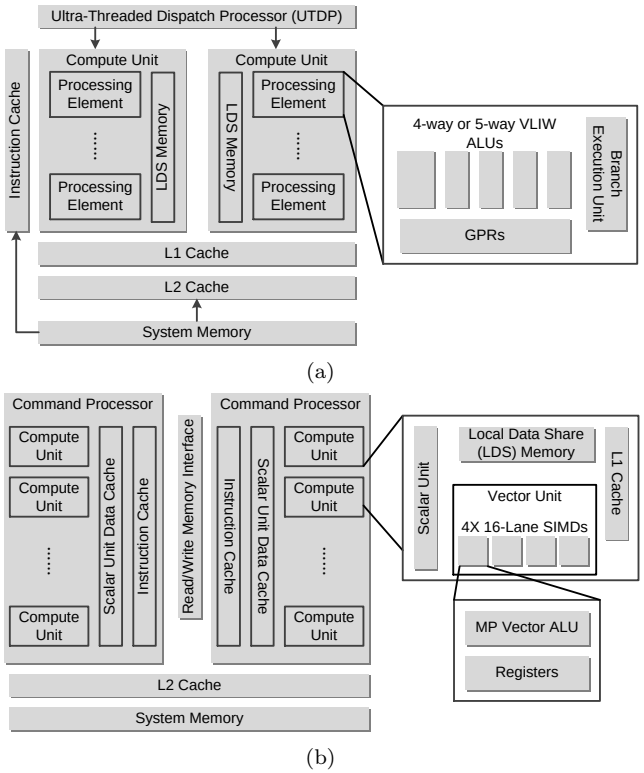


Figure 1: AMD GPU architecture block diagrams for (a) Evergreen or Northern Island and (b) Southern Island.

The AMD Fusion APUs integrate the CPU and the GPU on the same chip. Instead of the PCIe bus on the discrete GPUs that connects the separate device and the host memory, APUs use an on-chip memory bus that can achieve 1.2x-3.9x higher bandwidth. In addition, the physical memory is shared by the CPU and the GPU on the integrated architectures. However, the memory on current APUs is not coherent between the CPU and the GPU. The GPU can probe the CPU memory while the CPU relies on APIs to synchronize the GPU memory. In consequence, current APUs use different data paths shown in Figure 2 to access CPU/GPU memory. For example, the memory access request from the CPU should go through the write combination buffer and unified north bridge to the GPU in order to get the GPU memory physical address.

AMD introduced the zero-copy memory access technique on the GPU and the APU, which allows direct mutual memory accesses between the CPU and the GPU without transferring a copy of the data. Zero-copy is performed through on-chip data paths on APUs and PCIe bus on discrete platforms. The benefit of zero-copy includes two aspects: i) this feature saves memory space because only one copy of the data resides in the system; ii) when there are scattered memory accesses from one side to the other (e.g., CPU modifies a few discrete bytes in the GPU memory), zero-copy read/write for these scattered bytes can be faster than transferring all blocks of data between the CPU and the GPU.

2.2 Agent-Based Model and Traffic Simulation

The agent-based model [17] is a computational model for simulating a group of autonomous agents which usually per-

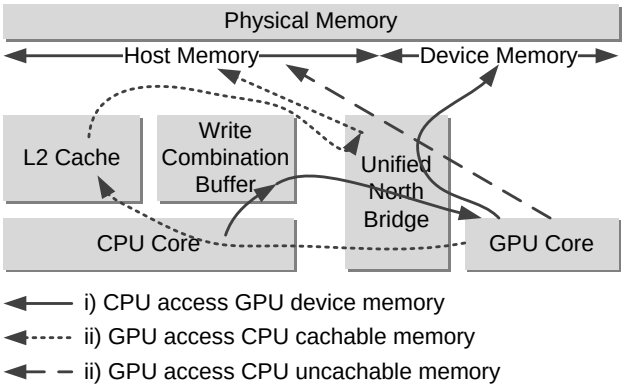


Figure 2: Data paths on AMD Fusion APU for mutual host-device memory access.

State	Description
Type	Either a car or a truck
X Position	The X coordinate of the vehicle in the lane. The direction of X axis is the same as the moving direction of the traffic.
Velocity	Velocity of the vehicle
Lane	Current lane number. For two-lane traffic simulation, lane number is either 0 or 1.

Table 1: List of vehicle states

form simple individual actions but can be intelligent and purposeful when interacting with their neighbors. People have been using the agent-based model for recreating and predicting the appearance of comprehensive systems such as economics, logistics, social networks and biological applications. A typical agent-based model simulation comprises the following four elements: i) agents with states, ii) communication with the neighbors, iii) universal transit functions that apply to each agent to update their states, iv) time-driven or event-driven simulation environment.

This paper uses traffic simulation in [23] as a specific example to illustrate the agent-based model. The traffic simulation is widely used for analyzing the traffic congestion patterns which can help advance road infrastructure and city planning. In this work, the time-driven simulation assumes traffic moving on a two-lane highway as shown in Figure 3a. Each vehicle, either a car or a truck, is an agent that possesses a set of states. Table 1 lists all the states used in the traffic simulation. For simplicity, the traffic simulation described in this paper does not include any intersection or turbulence. However, they can be easily integrated to current framework by introducing extra agent states and restrictions on the time-driven simulation environment.

In every time step, the transit functions update the states of each vehicle by i) computing the acceleration using the Intelligent-Driver Model (IDM) and ii) making lane-changing decisions using the model of Minimizing Overall Braking deceleration Induced by Lane changes (MOBIL) [23]. Both models perform arithmetic operations on the input vehicle states.

IDM determines the acceleration of each vehicle locally

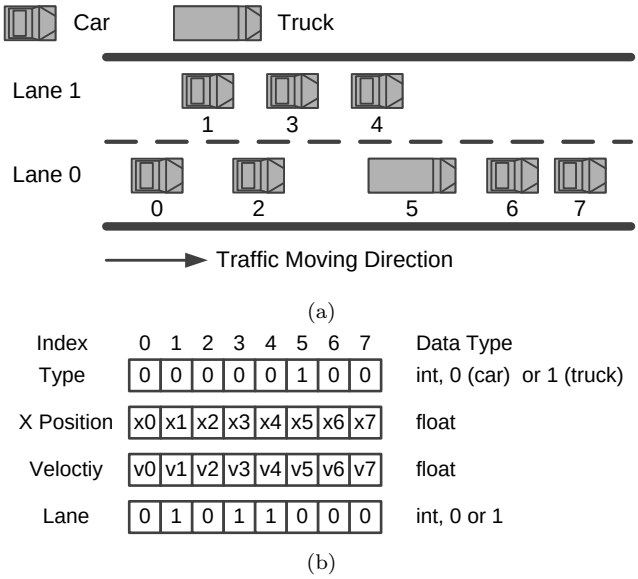


Figure 3: A sample of the traffic simulation problem. (a) is the traffic simulation for a two-lane highway with vehicles indexed and (b) shows the data structure used for the traffic simulation in (a).

from its own states and the states of the preceding vehicle. For example, in Figure 3a, the acceleration of vehicle 3 is computed from its own velocity, the velocity difference and distance from its preceding vehicle 4. IDM also computes the new velocity and X position according to the obtained acceleration in each time step.

The lane-change model MOBIL makes the lane-change decision by checking two criteria. The first criterion is the incentive criterion which is satisfied when the acceleration increase of the vehicle after a possible lane-change is larger than the acceleration decrease of the back vehicle because of the brake (e.g. acceleration increase of vehicle 3 in Figure 3a is larger than the acceleration decrease of vehicle 2 if 3 changes its lane and precedes 2). The second criterion is the safety criterion which is satisfied when the deceleration of the back vehicle after a possible lane change does not exceed a threshold, i.e. vehicle 2 does not brake too hard.

The IDM and MOBIL models use states from three neighbors as their inputs: preceding vehicle in the same lane, preceding vehicle and back vehicle in the other lane.

3. GPU MASSIVELY PARALLEL IMPLEMENTATION

This section discusses the major steps of implementing the traffic simulation on GPUs. The implementation uses a structure of arrays to store all the state data, with each array representing one state for all the vehicles, as shown in Figure 3b. The elements in the arrays are sorted according to the vehicle X position in each time step. The chosen data structure can generate optimized memory access pattern as well as minimize the effort to locate neighbors. While there are other data structure representations for agent-based models such as octree [9], they also rely on some restructuring methods similar to sorting in each time-step. Therefore, the implementation methods and conclu-

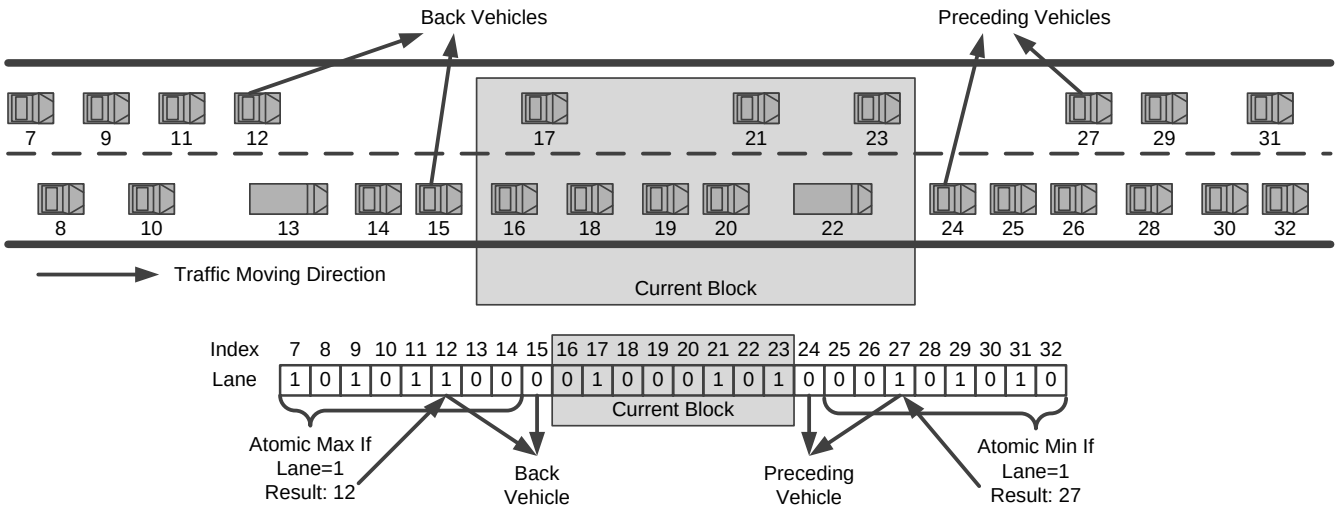


Figure 4: Locating two preceding and two back neighbors for a block.

sions drawn from this paper still apply.

This implementation of traffic simulation on GPUs uses one work-item for each vehicle and applies transit functions in parallel in each time-step. A work-group composed of several work-items processes a block of vehicles. Without loss of generality, all the figures in this paper are drawn assuming the size of the work-group, i.e. the size of a vehicle block, is 8 if not stated. The state data of the vehicles are maintained in the global memory. Three major steps in the simulation iteration are i) to locate the neighbors, ii) to update vehicle states, and iii) to sort the states array globally according to X positions.

3.1 Locating Neighbor and Update States

The first step of the implementation is to locate the three neighbors for the vehicles. The search of neighbors may cross the boundary of vehicle blocks, therefore, the core algorithm of locating neighbors is composed of two stages both of which have a BSP structure : i) locating neighbor vehicles for the blocks and ii) locating neighbors for individual vehicle within the block.

The first stage locates the preceding two neighbors and the back two neighbors from *each lane* for the block of vehicles processed by the work-group. An example for this stage in Figure 4 is that vehicle 12, 15, 24, 27 are the four neighbor vehicles for the current block 16 - 23. Although only one back neighbor is needed for an individual vehicle, this stage locates two back block neighbors since the last vehicle in the block can be in either lane. The following describes the algorithm for locating two *preceding* neighbors for the block. The same procedure can apply to locating two *back* neighbors with the opposite operations, e.g., using minimum instead of maximum operations, and minus instead of plus. The index of one preceding neighbor (vehicle 24) can be computed directly as the maximum vehicle index (vehicle 23) in the block plus one, since no matter which lane this vehicle is in, it must precede the head vehicle either before or after a possible lane-change. To locate the other preceding neighbor, the algorithm searches forward iteratively from the preceding neighbor that has been identified. In each iteration, the work group examines a block of vehicles in parallel with each work-item checking one vehicle. For example, in the first iteration, vehicles 25-32 in Figure 4 are examined by the work group. The search stops when a vehicle with the lane number different from the identified preceding neighbor is found. The minimum index is selected if multiple vehicles are found, which is implemented by applying the atomic minimum operation on the indices of those vehicles. In Figure 4, among the vehicles examined in the first searching iteration, vehicles 27, 29 and 31 have the different lane number from the identified preceding neighbor 24. The minimum of the three indices is 27, which is the other preceding vehicle for the block 16-23. The number of searching iterations is limited to a few blocks, since vehicles that are too far away have little effect on IDM, and therefore can be ignored.

The second stage locates the three neighbors for the individual vehicle in the current block respectively. These neighbors are the two preceding neighbors in each lane and one back neighbor in the lane after a possible lane-change. For instance, vehicle 19 has vehicles 20 and 21 as its two preceding neighbors and vehicle 17 as its back neighbor after a possible lane-change. At the beginning of this stage, the states of current block of vehicles are loaded into local memory with the two preceding and two back block neighbors, so that all following communication between vehicles in the block are conducted through the local memory for fast access. For example, the current block of vehicles 16-23 in Figure 4 along with four neighbors 12, 15, 24, 27 are loaded into local memory as shown in Figure 5. The following describes the procedure for locating the two *preceding* neighbors for the individual vehicle in the block. The same procedure can apply to locating the *back* neighbor by performing the computation in an opposite direction. The algorithm is composed of three steps as shown in Figure 5 and described as follows.

Step 1: Compute lane dynamic. Lane dynamic is defined as the indices of all the vehicles whose lane numbers are different from the previous vehicles. For example, if vehicle 17-21 have lane numbers (1,0,0,0,1), the lane dynamic is (18,21) since vehicles 18 and 21 have different lane numbers from vehicle 17 and 20. To compute lane dynamic, an

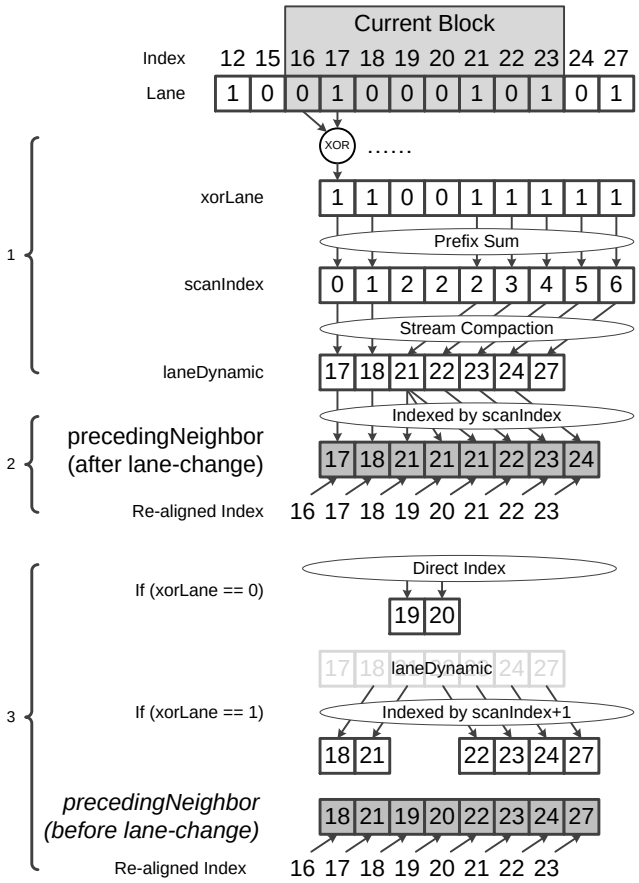


Figure 5: The three steps for locating two preceding neighbors for individual vehicle.

exclusive-or operation is applied to the lane numbers of two consecutive indexed vehicles, followed by the prefix sum and stream compaction [5] operations as shown in Figure 5.

Step 2: Compute the preceding neighbor after a possible lane-change. The basic idea is to find the next nearest vehicle in the block that has the different lane number from the current vehicle. This can be done by indexing the lane dynamic array with the scan index generated by the prefix sum as shown in Figure 5. The reason is that the scan index records the position in the lane dynamic array of the next vehicle whose xorLane is '1' (i.e. whose lane number is different from the previous vehicle). Indexing the lane dynamic array by the scan index can generate an array composed of these vehicles. When properly re-aligned with the indices of the current block, they exactly point to the preceding neighbors to be located in this step. Take an individual vehicle as an example, the scan index is 2 for vehicle 19, giving index 2 of the lane dynamic as the next vehicle with the different lane, which is 21. After a re-alignment with the current block indices, this number is aligned with 18, meaning that vehicle 18 has 21 as its preceding neighbor after a possible lane-change.

Step 3: Compute the preceding neighbor before a possible lane-change. There are two cases for this neighbor. In the first case, the current vehicle and the next nearest vehicle share the same lane number, so that the result of *exclusive-or* operation is 0 for these two vehicles and the neighbor can be located directly. For example, vehicles 18 and 19 in Figure 5 have the same lane number 0, which generates 0 as their *exclusive-or* result, therefore, 19 is the preceding neighbor of 18. In the second case, the next nearest vehicle and the current vehicle do not have consecutive indices, e.g. vehicle 20 and 21. The algorithm then increases the scan index by one and uses it to index the lane dynamic array. The lane number of the result vehicle is the same as the current vehicle as it changes twice (e.g. 0 changes to 1 and then back to 0). In Figure 5, the algorithm first checks the result of *exclusive-or* operations, *xorLane*, and then combines the two cases together to generate the final result.

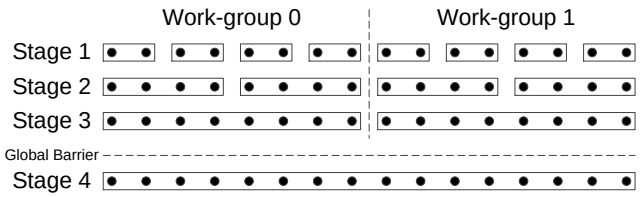


Figure 6: Bitonic sort algorithm skeleton.

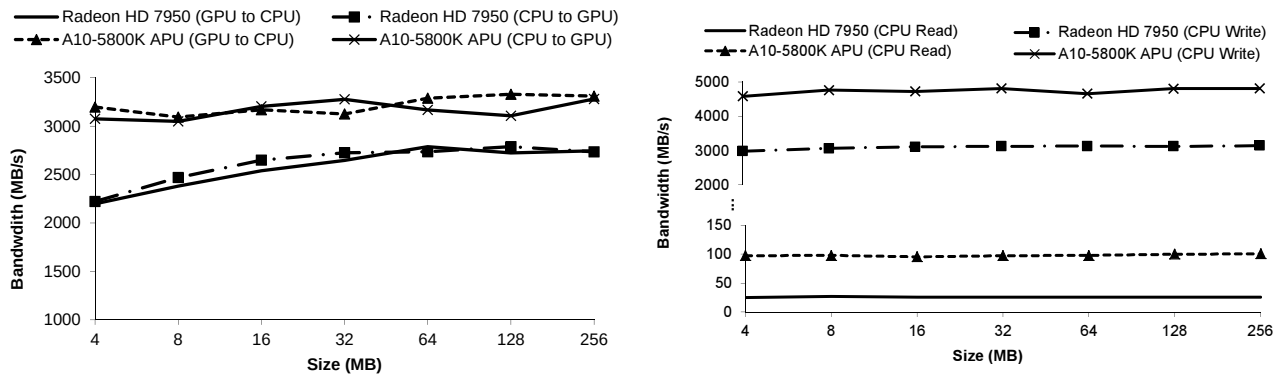
With all the three neighbors located, each work-item fetches the state data for current vehicle and the neighbors, computes the new acceleration, velocity and X position according to IDM, makes lane-change decision according to MOBIL and stores the newly updated states back to the global memory. This procedure is completely parallel since each vehicle can update its states independently. Ping-pong buffers are used to avoid any global memory race condition in loading the state data. When implementing the neighbor locating and states update, no cross-work-group dependency exists, requiring only one GPU kernel referred as *traffic states update kernel* hereafter.

3.2 Sorting States Data

The previous algorithm of locating neighbors is based on the assumption that all the state data of the vehicles are sorted according to the X position of the vehicle. In order for this assumption to hold, sorting is necessary at the end of each time step, simply because the X positions of the vehicles are updated and the relative sequential order can be changed when, for example, the vehicles in one lane pass the vehicles in the other lane.

This implementation uses bitonic sort algorithm [19] for its highly hierarchical structure that can be easily adapted to the framework of vehicle neighbor locating. Figure 6 shows the algorithm skeleton for bitonic sort. The algorithm contains $\log N$ stages (N is the number of the inputs to be sorted), each of which generates a bitonic sequence by comparing and swapping data. In Figure 6, the data in the rectangular boxes are either monotonically non-decreasing or non-increasing. For the beginning $\log M$ stages where M is the work-group size, all the data accessing and swapping are within the work-groups, therefore only work-group barriers are needed. However, all the subsequent stages require data accessing across work-groups, therefore the global barriers are necessary between stages.

The GPU implementation of other parallel sorting algorithm such as merge sort and radix sort have similar structures to the bitonic sort that require both the processing



(a) (b)

Figure 7: Bandwidth comparison of AMD HD7950 GPU and AMD A10-5800K Trinity APU. (a) shows the bandwidth for non-zero copy memory transfer rate and (b) shows the bandwidth for zero copy memory access rate.

within the work-groups and across the work-groups subjected to the global barriers. Therefore, they can also be used in the traffic simulation framework. We select bitonic sort for its best performance out of the experiments on several different sorting algorithms in the traffic simulation. The sorting stage is referred to the *sorting kernel* hereafter.

4. OPTIMIZATION FOR HETEROGENEOUS ARCHITECTURE

This section introduces an optimization of the traffic simulation implementation on the heterogeneous architecture featuring both the GPU and the CPU. We discuss the motivation and identify a part of the problem whose workload can be reduced and moved to the CPU.

4.1 Motivation

In traditional GPGPU programming, GPUs generally take the role of computation accelerators while CPUs prepare and transfer the data to the GPUs. Therefore, the CPUs are idle most of the time. One alternative would be to move some of the workload to the CPU. However, the overhead of memory access between the CPU and the GPU can impose significant impact on the overall performance.

The integrated architecture such as AMD APUs put the CPU and the GPU on the same die with the on-chip memory bus that can provide better support for zero-copy memory transfer and higher memory bandwidth. Figure 7 compares the CPU-GPU memory bandwidth of the discrete GPU which uses the PCIe bus and that of APU which uses the on-chip memory bus. The figure shows the comparison for both the memory access with non-zero copy and zero-copy. The zero-copy CPU read operations are very slow since the accesses are uncached and only a single outstanding read is supported. For both the zero-copy and non-zero copy cases, the integrated architecture outperforms the discrete platform with 1.2x-3.9x higher bandwidth, which enables more closely workload sharing between the CPU and the GPU with less overhead in data access. For the traffic simulation implementation, we adapt the algorithm to utilize both the CPU and the GPU and investigate the performance on different heterogeneous architectures.

4.2 Local Sort and Merge

For all three steps of the traffic simulation implementation on GPUs described in Section 3, the global sorting part consumes most of the execution time according to the performance break down in the experiment section (Section 5). We design an optimized algorithm to reduce the sorting workload and thereby the overall implementation workload. This optimization requires computation on CPU and the data transfer between GPU and CPU. The performance can vary depending on different heterogeneous architectures, especially taking advantage of the high bandwidth in the integrated architecture.

In the bitonic sort algorithm, if only the first $\log M$ stages are performed, the data are sorted within a work group. This is defined as *local sort*. After the local sort, the boundary data across the work group might be out-of-order, which requires merge between neighbor work-groups. In bitonic sort, the merge process is performed through the subsequent stages. However, in the traffic simulation, the vehicles can only have very small changes in their x positions in each time step, which would only require minimal effort to merge. Figure 8 shows two cases of traffic updates in one time step requiring or not requiring merge after local sort. In this example, the work-group size is set to 4. At time step N, work-group 0 updates vehicle 0-3 and work-group 1 updates vehicle 4-7. Then after one time step, vehicles are reordered in two cases. For case 1, reordering occurs only within the work-group, so that only local sort is needed. For case 2, vehicle 2 is moving fast enough to pass vehicle 4 in work-group 1. Therefore, after a local sort, vehicles can still cross the boundary of work-group 0 and work-group 1 so that the merge is necessary.

In the presence of both case 1 and case 2, we propose an optimization for the sorting algorithm in the original traffic simulation implementation, which is composed of two steps. In the first step, the first $\log M$ stages of bitonic sort are preserved to perform the local sort. Since the local sort is done within a work-group, it can be included in the original *traffic states update kernel*. In the second step, the CPU will check the boundary data of each work-group to perform necessary merge between work-groups. The algorithm for CPU merge is shown in Figure 9a. The inputs to the algorithm are all the vehicle states that have been local sorted by each work

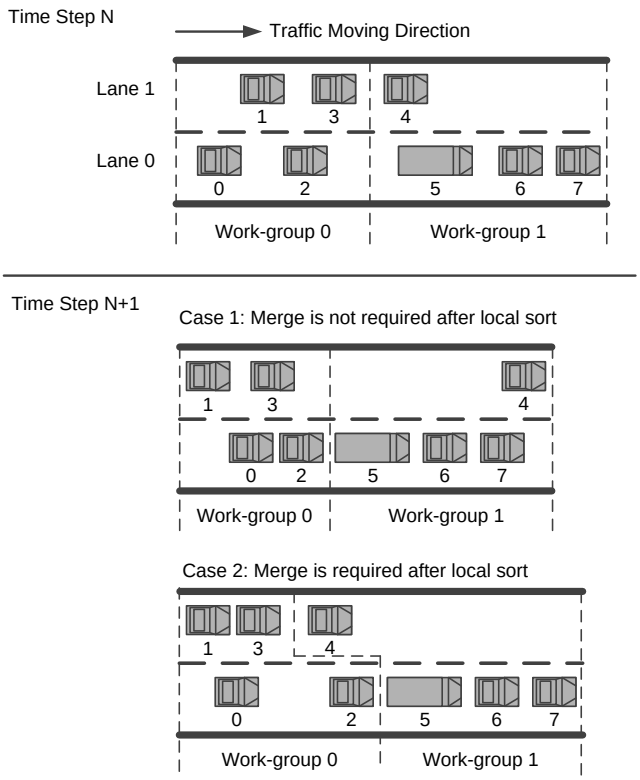


Figure 8: Two cases of traffic states update that require or do not require merge after local sort.

group. These data reside in GPU global memory. The CPU will access these values by direct zero-copy reading/writing. To determine if the merge is necessary, the CPU reads out the X positions of both the first and the last vehicles in each work-group and compares their values. This procedure is performed in the increasing order of the work-group indices. If there exists vehicles that cross the boundaries, e.g. the last vehicle in the current work-group has greater X position than the first vehicle in the next work-group as shown in Figure 9a, the CPU will do a merge across the work-group. Any traditional CPU-based merge algorithm can be used in this procedure. We specifically use the in-place merge algorithm introduced by Katajainen et al [12] for its fast computation performance and low space complexity. Parallel merge methods can also be utilized on multi-core CPUs. However, the details of the merge algorithm are out of the scope of this paper.

Sometimes the merge can involve several consecutive work-groups as shown in Figure 9b. At the beginning of time step N, vehicles in work-group 0 all lag behind work-group 1. After one time step, vehicle 3 in work-group 0 can pass all the vehicles in work-group 1 and some of the vehicles in work-group 2. In this case, the merge involves work-group 0, 1 and 2. However, there is a limit for the number of work-groups that can be involved in a single merge. The reason is that within a time step which is usually from less than one second to several seconds in the traffic simulation, vehicles in one lane can only pass a limited number of vehicles in the other lane. This limit can be achieved when vehicles in one lane stop moving due to some congestion while vehicles

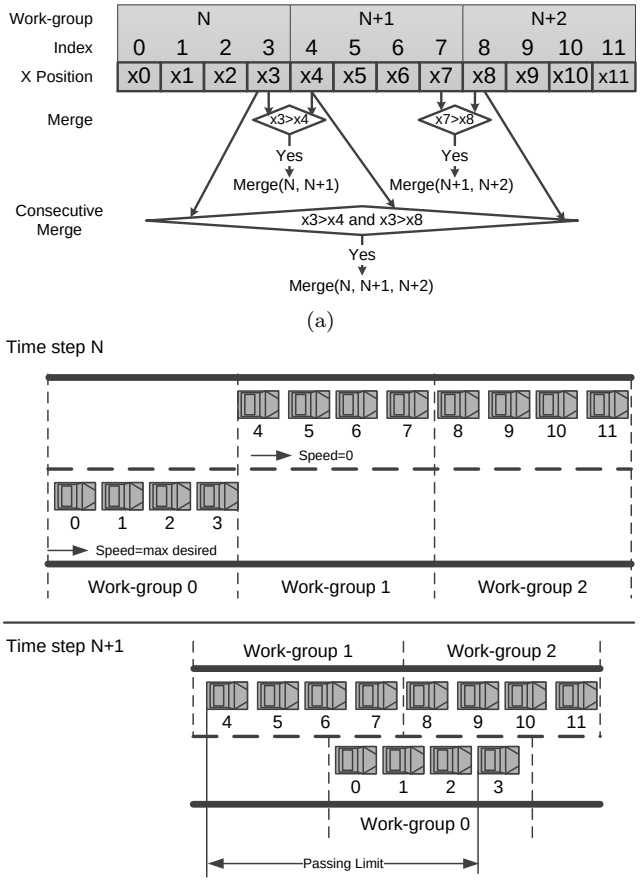


Figure 9: Proposed optimization for sort using merge on CPU. (a) shows the merge algorithm and (b) shows the maximum number of work-groups to be merged is imposed by the passing limit in traffic moving.

in the other lane move at the highest desired speed. In the example, since vehicles in lane 0 have 0 velocity and vehicle in lane 1 move at the maximum desired speed, the maximum possible number of work-groups that require merge is achieved, which is 3. In real simulations, this number can be computed by dividing the max distance one vehicle can travel in a time step by the minimum possible length of a work-group of vehicles. The imposed maximum work-group number to be merged can limit the CPU workload.

The proposed optimization can also apply to other agent-base model implementations [22, 16, 8] which have sorting as a stage to re-order the agent states. It has the following three features or benefits:

- It reduces the workload of the global sorting and thereby the overall system workload.
- The merge algorithm requires serial structure, which can have the CPU as its more natural venue on the heterogeneous architecture.
- In the merge operations, the communication between the CPU and GPU is through direct access from the CPU to the GPU memory, which can generate different performance on the discrete platform with the PCIe

	Integrated	Discrete
Device		
Platform (AMD)	A10-5800K APU	Radeon HD7950
GPU Arch.	Northern Island	Southern Island
Stream Procs	384	1792
Compute Units	6	28
Device Memory	512 MB	3072 MB
Local Memory	32KB	64KB
Peak Clock Freq.	800MHz	850MHz
Host		
Processor	A10-5800K APU	Intel i7-920
Clock Freq.	3.8GHz	2.66GHz
System Memory	8GB	8GB

Table 2: Experimental environment.

Parameter	Value
Everyday Acceleration a	car:0.5, truck:0.4, m/s^2
Desired Velocity v_0	$80km/h$
Acceleration Exponent δ	4
Safety Headway Time T	$1.5s$
Comfortable Deceleration b	$3m/s^2$
Minimum Gap s_0	$3m$
Politeness Factor p	0.2
Max Safe Deceleration b_{save}	$12m/s^2$
Lane-change Threshold a_{thr}	car:0.3, truck:0.2
Car Length	car:$6m$, truck:$10m$
Simulation Time Step	$0.5s$

Table 3: Traffic simulation parameters used in the experiments.

bus and on the integrated platform with the on-chip memory bus.

5. EXPERIMENTS AND EVALUATION

We run the traffic simulation both on the discrete and the integrated heterogeneous architectures. The experimental environment is specified in Table 2. The parameters for the traffic simulation are shown in Table 3. The work-group size is 64 for the simulation, which is the optimized result from multiple experiments of different kernel configuration on both the discrete and the integrated GPUs. For OpenCL implementation running on CPUs, the performance does not vary when the work-group size exceeds 16. Therefore, the work-group size is also set to 64 to ease performance comparisons. We set the input size from 128 to 2M vehicles, which might be considerably large for real traffic, however, we claim that the experiments and conclusions in this section can apply to other large-scale agent-based model such as cell-level biological simulation where 2M agents are common and reasonable.

5.1 Evaluation of the GPU Implementation

We first evaluate the performance of the original GPU implementation of the traffic simulation on the discrete GPU HD7950, the GPU part of A10-5800K APU and the Intel CPU (as the compute device in OpenCL) by comparing the speedup over the serial implementation [23] on Intel CPU. We measure the execution time of one time-step simulation averaged from 10000 iterations and compute the speedup over the serial CPU implementation which is shown in Figure 10a.

The OpenCL implementation on the Intel CPU achieve similar performance compared to the serial implementation for input size 128 - 1K. When the input size increases, the OpenCL implementation can have more benefit from its parallel algorithm design and achieves more speedup from 1.25x to 1.98x.

Similarly, for small input size, the GPU part of APU and the discrete GPU performs worse than the serial implementation. More than 1x speedup can be observed on both devices for input size larger than 2K. This speedup increases steadily from 1.29x to 33.9x on the discrete GPU with the simulation input size. In comparison, the implementation on the APU achieves 1.41x-2.72x speedup over the serial implementation.

The speedup over the serial implementation demonstrates that the parallel algorithm designed for agent-based is efficient on both the multi-core CPU and the GPU. The difference between the speedup on the APU and the discrete GPU shows that the latter has much stronger computation power than the GPU part of the APU due to larger number of computation units, advance in architecture and higher clock speed.

To further analyze the performance result, we break down the execution time for the *traffic states update kernel* and the *sorting kernel* in Figure 10b and Figure 10c, normalized towards the total execution time for each simulation size on each device. Both on the discrete GPU and the GPU part of the APU, the *sorting kernel* takes more than 80% of the total execution time when the input size exceeds 16K, which justifies the rationale of the proposed optimization in Section 3.1. It can also be noted that the percentage of the execution time consumed by sorting is increasing since more stages and global barriers are required for larger input size.

We also evaluate the performance of other sorting algorithms when used in this problem. For example, the traditional odd-even sorting algorithm [15] is a good candidate on the almost sorted vehicle states. We observe that for input size which is smaller than 64K vehicles, the odd-even sorting can be very efficient that takes less than 65% of the total execution time. However, the performance decreases significantly for larger input size due to more iterations required to converge and increased overhead on global synchronization for each iteration. When the number of vehicles exceeds 256K, the sorting takes more than 99% of the total execution time. The comparison of multiple sorting algorithms shows that bitonic sort is the most efficient for this problem, which is also evidenced by optimal sorting algorithms on GPGPU architectures [20].

5.2 Evaluation of the Optimization on Heterogeneous Architectures

As discussed in Section 4.2, merge is a necessary step in the proposed optimization when there exists vehicles that cross the work-group boundary after the local sort. To evaluate how often the merge is required, we compute the *merge rate* which is defined as the number of work-groups that require merge divided by the number of total work-groups. Figure 11 shows the merge rate for different simulation size. The low merge rate (less than 25%) for all input sizes indicates that most of the time, performing only the local sort will preserve the ordering of the states data, so that the merge effort can be minimized to reduce the overall imple-

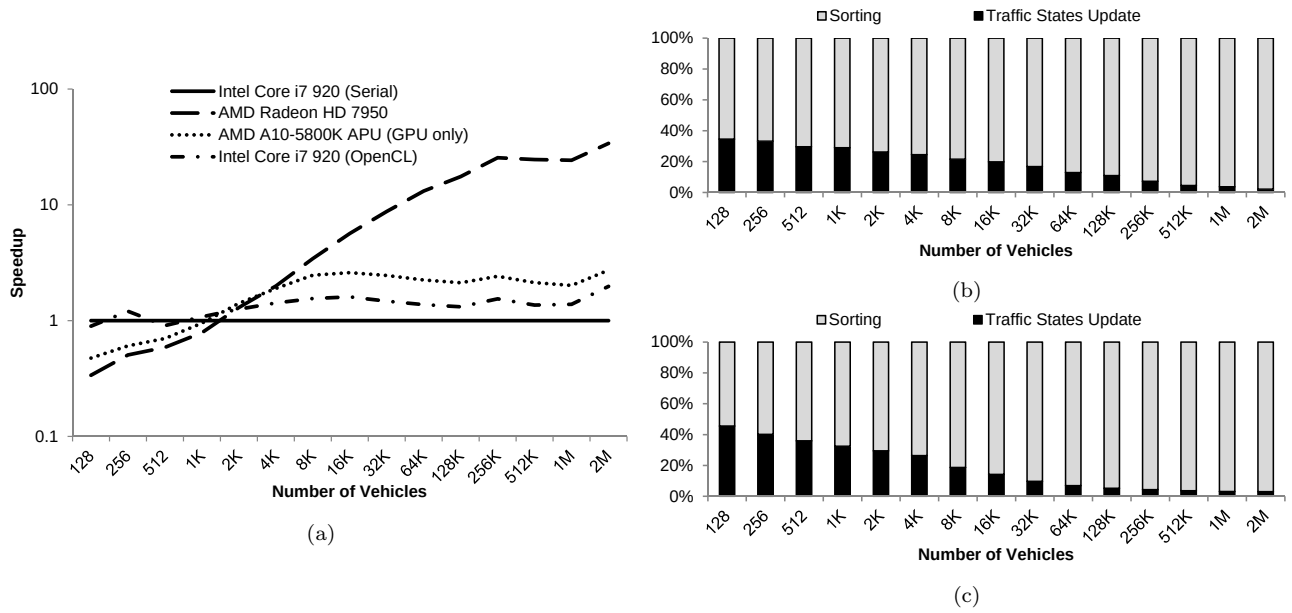


Figure 10: Performance of GPU implementation of traffic simulation. (a) shows the speedup of the OpenCL implementation on HD7950, A10-5800K and Intel Core i7 920 over the serial implementation on Intel Core i7 920, (b) and (c) shows the execution time breakdown for traffic states update and sorting normalized to the total execution time on HD7950 and A10-5800K respectively.

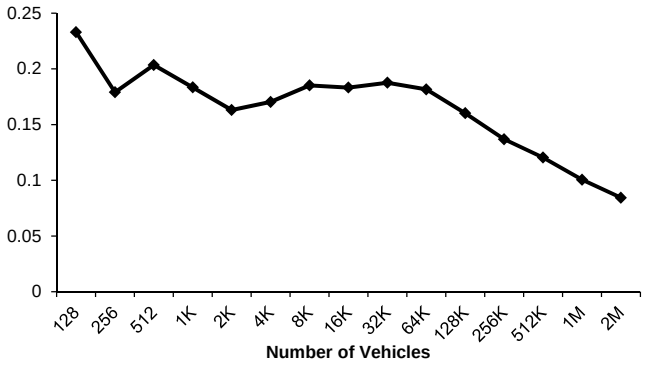


Figure 11: Merge rate for different simulation size.

mentation workload.

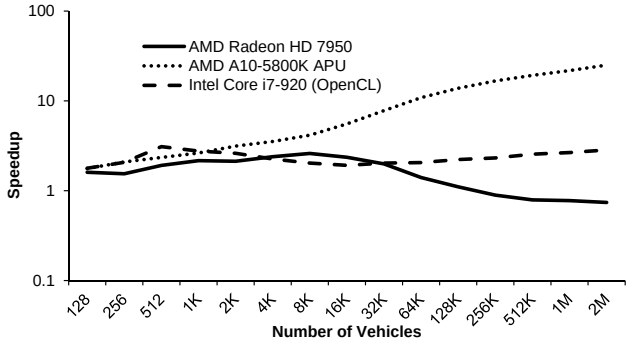
We apply the proposed optimization both on the discrete and the integrated heterogeneous platforms. On both platforms, the GPU performs the traffic states update and local sort while the CPU performs the merge. The CPU accesses the traffic states data directly through zero-copy memory space mapping from the GPU to the CPU. Therefore, only one copy of the states data resides in the system. The memory mapping and data access for zero copy are through the PCIe bus on the discrete platform and the on-chip memory bus on the integrated platform. As a comparison, we also evaluate the optimization on the OpenCL implementation on the Intel CPU. In this case, CPU performs both the states update/local sort and the merge operations by directly accessing the data in the system memory.

Figure 12a shows the speedup over the original OpenCL implementation on the three platforms respectively and Figure 12b compares the speedup over the baseline GPU implementation on discrete GPU HD7950.

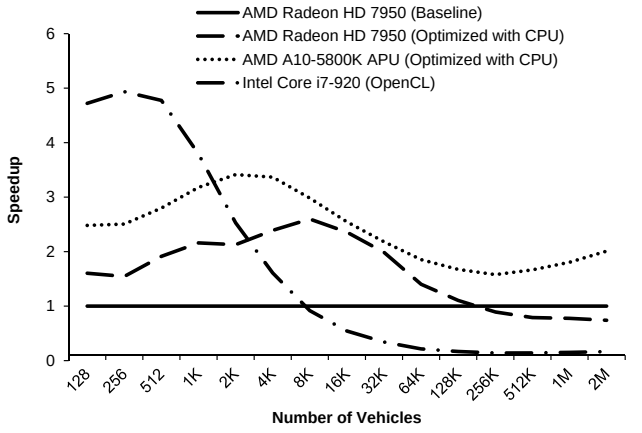
The APU achieves most significant speedup from the optimization which is up to 25x over the original implementation and 1.57x to 3.41x over the baseline implementation on the discrete GPU. In the contrast, the discrete GPU achieves 1.1x to 2.6x speedup for input size smaller than 128K but performs even worse than the baseline implementation for input size larger than 128K (0.8x-0.9x). If we compare the optimization result on the APU and the discrete GPU, the former achieves overall performance 1.08x to 2.71x over the latter. The speedup on both the APU and the discrete GPU is a result of reduced workload in the sorting. However, the larger input size requires more data accesses through the memory bus. The scattered memory access pattern in the CPU merge algorithm can even increase the memory bus pressure. On the discrete GPU, the memory access overhead negates the benefit of reduced workload for large input and results in the overall system performance decrease. Compared to the PCIe bus that connects the discrete GPU to the CPU, the APU has higher memory bandwidth that can help reduce the overhead of memory access which results in performance increase.

The optimization of the OpenCL implementation on the Intel CPU introduces very little data access overhead, so that we can see the speedup over the original implementation when replacing the workload of global sort with local sort and CPU merge. However, since the global sort does not take as much time as in the GPUs, the optimization only achieves 1.8x-3.0x speedup. When compared to baseline implementation on HD7950, the multi-core CPU achieves 1.6x-4.9x speedup for input size smaller than 4K when the computation capability on GPU is not saturated but performs worse thereafter.

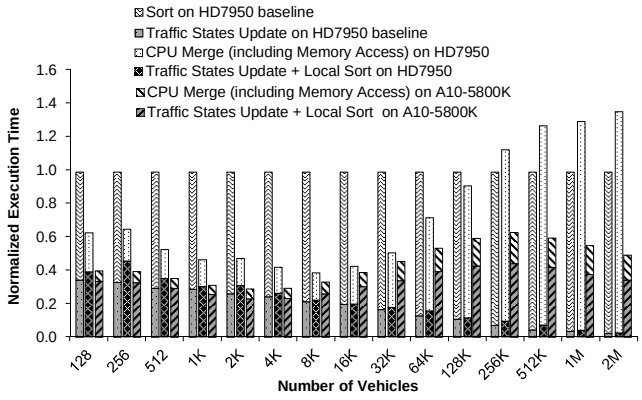
Figure 12c demonstrates the memory access overhead by



(a)



(b)



(c)

Figure 12: Performance of the optimization for the traffic simulation. (a) shows the individual speedup over the original OpenCL implementation, (b) shows the speedup over the baseline GPU implementation on HD7950 and (c) shows the execution time breakdown normalized to the baseline total execution time on HD7950.

showing the breakdown of the *traffic states update/local sort kernel* time and the CPU merge time. It also shows the breakdown of the baseline discrete GPU execution time as a comparison. Note that with the zero-copy technique, the memory access time composes the majority of the CPU merge time since the merge algorithm is memory bound and CPU performs the merge by reading and swapping the data in the GPU memory directly.

On the discrete GPU, the new *traffic state update kernel* after optimization includes the local sort portion, which results in slight execution time increase as shown in Figure 12c. The global sort in baseline implementation is replaced with the CPU merge, which can be efficient for small input size but consumes a large amount of time when the input size increases and thereby hurts the overall system performance.

In contrast, the implementation on the APU consumes much more time on the computation part including the traffic states update and the local sort due to the reduced computational capability. For input size larger than 8K, the computation part on APU takes 1.2x to 14.6x more time than the discrete GPU. However the time spent on CPU merge is much less (only 11% to 45% of the time spent on CPU merge on the discrete GPU or 9% to 20% of the time spent on global sort in discrete GPU baseline). When combining the computation part and the memory access time, the APU can still achieve better overall performance.

5.3 Discussion

The optimization for integrated architectures can also be performed using sorting algorithms other then bitonic sort as long as they can benefit from splitting workload between the CPU and GPU. For example, the earlier iterations in the odd-even sort requires heavy work on comparing and swapping data, for which the parallel execution on the GPU can be highly efficient. However, the later stages have unbalanced workload for sorted and unsorted blocks, therefore, running them on the CPU would be a better choice to reduce the overhead of global synchronization between each stage. As a practical matter, the choice of sorting algorithm is an engineering decision depending on the problem, but the optimization proposed in this paper is generically applicable to different sorting algorithms.

While we perform the experiments specifically for the traffic simulation problem, the optimizations and results can be generalized for other agent-based model applications because of their similarity in neighbor locating and communication patterns. For a general agent-based model simulation framework [21, 16, 8], restructuring such as sorting is required in each time step to re-order the states data, which usually contains serialized work that can be implemented on the CPU and take advantage of the integrated architectures.

The experiments reveal that when the algorithm is properly designed, the APU can achieve the balance between the computation power and the host-device memory bandwidth, which provides an opportunity for mapping current GPGPU applications to the integrated heterogeneous architecture to utilize both the GPU and the CPU while still preserving low memory transfer overhead. The overall system performance on the APU can be even better than on the discrete GPU.

For large-scale parallel programs such as data-warehousing applications, the overhead of PCIe can eat up all the benefits from optimizing the parallel algorithms on the GPU [25]. This paper provides the insight that the integrated architecture can help these large-scale application to avoid the slow PCIe bus and achieve better performance.

6. RELATED WORK

Recently agent-based model simulations have been implemented on the GPU architectures with the support of the GPGPU programming model. The work of Richmond et

al. [21] and Lysenko et al. [16] are among the earlier researchers that implement the agent-based model on GPUs by addressing problems such as mapping agents, developing partitioning schemes to allow agent communications, solving agent collision problems and dealing with agent death and reproduction. Richmond et al. also advance their work to a general GPGPU agent-based model simulation frame work FLAME [22]. While all the above works report significant performance improvement over CPU implementations, they hardly propose any optimizations in neighbor communications which rely on the restructuring methods like sorting. Erra et al. [8] propose a more efficient algorithm on GPUs to perform the nearest-neighbor search, which comprises hashing mapping, sorting and reordering of the agent states. Aaby et al. [1] identify the optimization that reduces the global synchronization and communications between neighbor blocks of agents by introducing overlap in the processing blocks. They claim the optimization is able to hide the latency caused by the global synchronization. The various optimizations enable large-scale agent-based model simulation on a single GPU as well as the extension to multi-GPU clusters. The optimization proposed in this paper also targets more efficient neighbor communications, with more emphasis on the utilization of the memory hierarchy of the state-of-the-art heterogeneous architecture.

As a recent trend in heterogeneous computing, researchers are spending more effort on the evaluation of the integrated GPU-CPU architectures such as the AMD Fusion APU. These evaluation works are represented by research on the performance measurement for mapping various GPGPU applications onto integrated architecture where the new on-chip memory bus alleviates the communication bottleneck of data transfer between the CPU and the GPU. Doerksen et al. [7] study the design and implementation of two problems: 0-1 knapsack and Gaussian Elimination on an APU, claiming that most of the workload has been focused on the GPU while the CPU portion only checks the termination conditions and performs synchronizations. Their research is augmented by Daga et al. [6], who study and evaluate a more complete set of applications on the APU. The study analyzes the detailed effect of the memory bandwidth of the discrete GPU and the APU fusion architecture and concludes that the APU may have more benefit on the overall performance when there are large amounts of data to be transferred. While the above works are valuable and significant in current research on the integrated architecture, it should be noted that the applications used by these works are originally designed for GPU architectures. To utilize the advantage provided by the APU, the algorithms should be redesigned to extract more computational power from the CPU part and utilize the higher memory bandwidth. This paper demonstrates this point by designing an optimization that can utilize both the GPU and the CPU part with reasonable amount of data access between them.

7. CONCLUSION

This work addresses the problem of implementing and optimizing agent-based models on heterogeneous architectures. Specifically, we illustrate the GPU implementation details for traffic simulation using agent-based model, including the neighbor locating and states update algorithms. To tackle the problem of inefficient sorting in the agent states updating, we propose an optimization that combines low-workload local sort with necessary boundary merge. This optimization utilizes the computation capability of both the CPU and the GPU part of the heterogeneous architectures and requires memory access from the host to the device. The experiments show that the implementation on the APU performs 1.08x to 2.71x faster than the discrete GPU, which validates the importance of fast CPU-GPU communication.

The conclusions and insights from this paper are important to a variety of problems since it provides a way for mapping the traditional GPGPU applications to integrated heterogeneous architectures by redesigning the algorithm in favor of a more closely coupled CPU-GPU memory hierarchy. The generality of the methodology for accelerating these applications deserves further research.

8. REFERENCES

[1] B. Aaby, K. Perumalla, and S. Seal. Efficient simulation of agent-based models on multi-gpu and multi-core clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 29. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[2] AMD. Amd fusion family of apus: Enabling a superior, immersive pc experience. March 2010.

[3] AMD. Amd accelerated parallel processing opencl programming guide. December 2012.

[4] AnandTech. Amd's graphics core next preview: Architected for compute. 2011.

[5] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166. ACM, 2009.

[6] M. Daga, A. Aji, and W. Feng. On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, pages 141–149. IEEE, 2011.

[7] M. Doerksen, S. Solomon, and P. Thulasiraman. Designing apu oriented scientific computing applications in opencl. In *International Conference on High Performance Computing and Communications (HPCC)*, pages 587–592. IEEE, 2011.

[8] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *International Workshop on High Performance Computational Systems Biology, 2009. HIBI'09*, pages 51–58. IEEE, 2009.

[9] N. Ferrando, M. Gosalvez, J. Cerda, R. Gadea, and K. Sato. Octree-based, gpu implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, 182(3):628–640, 2011.

[10] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Supercomputing*, 2012.

[11] Intel. 2nd generation intel core i5 processor. 2011.

[12] J. Katajainen, T. Pasanen, and J. Teuhola. Practical

in-place mergesort. *Nordic Journal of Computing*, 3(1):27–40, 1996.

[13] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, September-October 2011.

[14] Khronos Group. *The OpenCL Specification, version 1.2.19*, November 2010.

[15] S. Lakshmivarahan, S. Dhall, and L. Miller. Parallel sorting algorithms. *Advances in Computers*, 23:295–354, 1984.

[16] M. Lysenko and R. D'Souza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[17] M. Niazi and A. Hussain. Agent-based computing from multi-agent systems to agent-based models: a visual survey. *Scientometrics*, 89(2):479–499, 2011.

[18] NVIDIA. Nvidias next generation cuda compute architecture: Fermi. 2009.

[19] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place sorting with cuda based on bitonic sort. *Parallel Processing and Applied Mathematics*, pages 403–410, 2010.

[20] M. Pharr and R. Fernando. Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation. 2005.

[21] P. Richmond and D. Romano. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings of International Workshop on Super Visualisation (IWSV08)*, 2008.

[22] P. Richmond, D. Walker, S. Coakley, and D. Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[23] M. Treiber and A. Kesting. An open-source microscopic traffic simulator. *Intelligent Transportation Systems Magazine*, 2(3):6–13, Fall 2010.

[24] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[25] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 '12, pages 107–118, 2012.