Enabling Latency-Aware Data Initialization for Integrated CPU/GPU Heterogeneous Platform

Zhendong Wang, *Graduate Student Member, IEEE*, Zihang Jiang, Zhen Wang, Xulong Tang, Cong Liu, *Member, IEEE*, Shouyi Yin, *Member, IEEE*, and Yang Hu

***Abstract*—Nowadays, driven by the needs of autonomous driving and edge intelligence, integrated CPU/GPU heterogeneous platform has gained significant attention from both academia and industry. As the representative series, NVIDIA Jetson family perform well in terms of computation capability, power consumption, and mobile size. Even so, the integrated heterogeneous platform only contains one limited physical memory, which is shared by the CPU and GPU cores and can be the performance bottleneck of the mobile/edge applications. On the other hand, with the unified memory (UM) model introduced in GPU programming, not only the memory allocation is significantly reduced, which mitigates the memory bottleneck of the integrated platforms but also the memory management and programming are simplified. However, as a programming legacy, the UM model still follows the conventional copy-then-execute model, initializing data on the CPU side after allocating memory. This legacy programming mode not only causes significant initialization latency but also slows the execution of the following kernel. In this article, we propose a framework to enable the latency-aware data initialization on the integrated heterogeneous platform. The framework not only includes three data initialization modes, the CPU initialization, GPU initialization, and hybrid initialization, but also utilizes an affinity estimation model to wisely decide the best initialization mode for an application such that the initialization latency performance of the application can be optimized. We evaluate our design on NVIDIA TX2 and AGX platforms. The results demonstrate that the framework can accurately select a data initialization mode for a given application to significantly reduce the initialization latency. We envision this latency-aware data initialization framework being adopted in a full-version of autonomous solution (e.g., Autoware) in the future.**

***Index Terms*—Affinity estimation, data initialization, heterogeneous platforms, integrated GPU (iGPU), latency, unified memory (UM).**

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by NSF under Grant CNS 1527727, Grant CNS 1750263 (CAREER), Grant CCF 1822985, and Grant CCF 1943490 (CAREER). This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Yang Hu.)*

Zhendong Wang, Zhen Wang, Cong Liu, and Yang Hu are with the University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: zhendong.wang@utdallas.edu; zhen.wang2@utdallas.edu; cong@utdallas.edu; yang.hu4@utdallas.edu).

Zihang Jiang and Shouyi Yin are with Tsinghua University, Beijing, China 100084 (e-mail: jiangzh15@tsinghua.org.cn; yinsy@tsinghua.edu.cn).

Xulong Tang is with the University of Pittsburgh, Pittsburgh, PA 15213 USA (e-mail: tax6@pitt.edu).

Digital Object Identifier 10.1109/TCAD.2020.3013047

I. Introduction

THE EMERGENCE of heterogeneous System-on-a-Chip (SoC) has pushed the computing platforms of many edge-intelligence applications, such as driving automation system, drone, and robots, on the verge of a major design shift from high performance, energy-consuming discrete CPU and GPU equipped platform, to effective, energy-efficient integrated CPU and GPU platform. As the major game player, NVIDIA has developed a series of heterogeneous SoC platforms, such as Jetson SoCs [1] and Drive SoCs [2], for autonomous embedded systems and driving automation systems, respectively. The integrated CPU and GPU heterogeneous platform typically shares the physical memory between the CPU and GPU cores. For example, NVIDIA TX2 possesses an on-chip memory of 8 GB, which is physically connected by CPU and GPU cores.

Exploiting integrated GPU (iGPU) heterogeneous platform for autonomous and intelligent workloads processing can bring better size, weight, and power (SWaP) tradeoff compared to traditional discrete GPU (dGPU)-based solutions. Moreover, the lower price makes the iGPU platform an excellent choice for embedded autonomous systems. Despite the advantage of the iGPU-enabled heterogeneous platform, building future embedded autonomous systems based on this hardware is further stymied by the unprecedented challenges that are specifically imposed by the *stringent latency requirements* and *memory footprint* of autonomous systems, and the *intrinsic hardware restrictions* of iGPU-enabled heterogeneous SoC platforms.

First, modern autonomous systems, such as drones and driving automation systems need *a vast amount of perception data* for decision-making guidance [3]. An autonomous vehicle is typically equipped with 8 to 12 surrounding cameras to provide 360-degree visibility around the vehicle, with a single two-megapixel camera (24 bits per pixel) operating at 30 frames/s generates 1440 MB of data every second. If a poorly managed memory allocation method such as the traditional copy-then-execute model is used, where memory is allocated on both host and device side, such a large amount of raw data can easily exhaust the memory of iGPU-based heterogeneous platform.

Second, modern autonomous systems consider the *processing latency* as one of the most important tenets for safety and functionality. The latency from image capture to recognition completion is critical since the response time of the control operations depends on it. Failing to execute actuation in time may cause catastrophic consequences, such as financial

0278-0070 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

TABLE I
MATRIX OPERATION SCALE (M.O.S.) OF TYPICAL DNN MODELS APPLIED IN AUTONOMOUS DRIVING, INCLUDING MATRIX MULTIPLICATION AND ADDITION

DNN model	YOLO2 [4]	YOLO3 [5]	SSD [6]	DAVE-2 [7]
M.O.S.	49K	81K	10K	250K

repercussions or even loss of human lives. However, modern driving autonomous systems adopt a variety of DNN models to achieve complete functionalities, while the DNN functions typically involve large-scale matrix operations. Table I shows the matrix operations scale of several representative DNNs models adopted by the mainstream autonomous driving solutions.

To meet these rigorous requirements on memory footprint and processing latency for the iGPU-based heterogeneous computing platforms, tapping into the emerging unified memory (UM) model is considered as a promising solution. Recent NVIDIA GPU architectures enable UM [8] to ease the explicit programming efforts of handling data movement and address translation. The UM address space is shared among CPU nodes and GPU nodes in the system. By uniforming the address space, the programmer can minimize the efforts of explicitly managing the data movement between CPU and GPU. The UM is particularly beneficial for iGPU platform to process autonomous workloads since it replaces the explicit data copy in the traditional copy-then-execute model with implicit data initialization and addresses translation, which can save memory footprint.

However, we observe that the current UM model does not further reduce the processing latency on iGPU-based heterogeneous platforms, such as NVIDIA Jetson TX2. Our characterization shows that the inherited routine of initializing data on the CPU side provided by the UM model not only results in significant latency due to the limited computation capability of CPU but also slows the following GPU kernel execution. Especially, we observe that the existing CPU-based matrix initialization can introduce striking latency for GPU tasks with large-scale matrix operations. Here, initialization refers to the process of filling out the input buffer. Intuitively, for some simple applications, the input variables can be directly assigned with constant/specific values (e.g., being copied from another constant buffer) or the input variables can load values from the existing file (e.g., read input feature map for the CNN model). For example, regarding the intelligent and autonomous workloads, the object-detection CNN models are widely adopted and the model can either load image data from system memory or fetch data from the sensor memory for realtime detection.

Based on these findings, to reduce the data initialization latency as well as the entire application's response time, we design a framework that enables the latency-aware data initialization for iGPU-based heterogeneous platforms. More specifically, the framework provides two new data initialization modes, the GPU-side initialization (GPU-Init) and the hybrid initialization (Hybrid-Init, i.e., the initialization is cooperatively implemented by both CPU side and GPU side) on top of the existing CPU-side initialization (CPU-Init). Furthermore, we develop an affinity estimation model, which can estimate the affinity to a certain initialization mode for an application based on the application's characteristics (e.g., the size and the diversity of the initialized data) and the platform's features (e.g., the computation capability), to help the framework wisely choose the corresponding initialization mode such that the initialization latency performance of the application can be optimized. Finally, we extensively evaluate our latency aware data initialization framework on the NVIDIA Jetson TX2 and Xavier AGX platforms using Rodinia benchmarks [9] and a popular object detection CNN model. The evaluation demonstrates the accuracy of the affinity estimation model and the efficacy of the framework in deciding an initialization mode and optimizing the initialization latency performance for an application. In summary, this article makes the following contributions.

1) We conduct a comprehensive characterization of the current UM model on emerging integrated CPU/GPU platform, the NVIDIA TX2. Our characterization results show that the existing CPU-side data initialization mechanism can cause significant processing latency for the overall application.
2) For an application, we develop a latency-aware data initialization framework that includes three initialization options, the CPU-Init, GPU-Init, and Hybrid-Init.
3) The framework can utilize an affinity estimation model that incorporates the workloads and system features (e.g., the input data size, initialization diversity, and platform capacity) to wisely decide the initialization mode to optimize the initialization latency performance of the application.

II. BACKGROUND

A. Memory Management Methods and Unified Memory

There exists various methods to manage the memory between the host CPU and the device GPU in CUDA programming. Conventionally, the memory management follows a *copy-then-execute* model. That is to say, the host CPU has to calls specific API [i.e., *malloc*() and *cudaMalloc* in CUDA programming] to allocate memory space on both CPU side and GPU side for the data. After allocation, the host CPU initializes data (e.g., assign values to the variables) on its own memory, and then copy the initialized data from host memory to the device memory before a GPU kernel can execute. Once the kernel execution completes, the results data are copied back from the device memory to the host memory. In this process, a comparable data-size space is created on both host and device memory regions, which requires $2\times$ memory space to be allocated to the same data. Also, the data transfer between the host memory and device memory typically contributes to the response time of the application.

Later, the UM model is developed, which not only frees the developers from this complicated data copy process but also significantly reduces the allocated memory space for an application. Typically, the UM can provide an illusion of

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

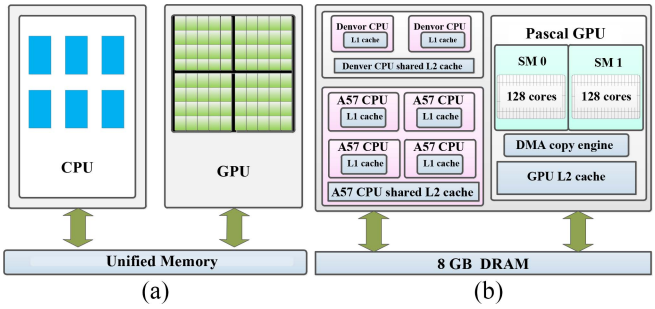


Fig. 1. Schemes of (a) UM programming model and (b) Parker-based TX2 architecture.

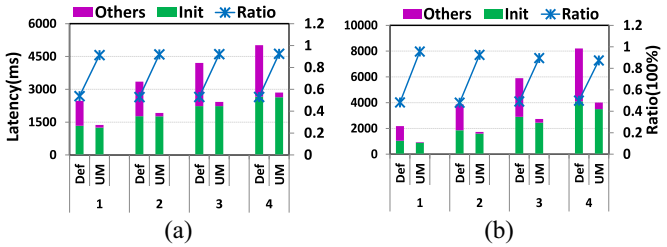


Fig. 2. Average CPU-Init latency under Def. and UM. models on TX2 as well as the corresponding initialization ratio in the overall application response time. (a) MatAdd. (b) MatMul.

TABLE II
LONGEST CPU-INIT LATENCY AS WELL AS THE INITIALIZATION RATIO UNDER DEF. AND UM. MODELS. EACH CELL (*a, b, c*) INDICATES THE INIT., OTHERS, AND RATIO, RESPECTIVELY

Longest	MatAdd				MatMul			
	1	2	3	4	1	2	3	4
Def	(1421, 1250, 0.53)	(1894,1655, 0.53)	(2335,2071, 0.53)	(2910,2431, 0.54)	(1102,1183, 0.48)	(1951,2107, 0.48)	(3045,3150, 0.49)	(4306,431, 0.50)
UM	(1335,120, 0.91)	(1886,162, 0.92)	(2405,200, 0.92)	(2804,224, 0.93)	(938,47, 0.95)	(1676,136, 0.92)	(2574,301, 0.89)	(3679,530, 0.87)

CPU–GPU unified virtual memory to avoid explicit data copy and ease memory management. The programmer only allocates memory once, which can be accessed by either CPU or GPU through a shared pointer. As a result, the memory footprint is reduced. The API *cudaMallocManaged*() corresponds to UM allocation in CUDA programming. As a matter of fact, NVIDIA's UM management model has been introduced since CUDA 6 [10]. The architecture is shown in Fig. 1(a). After CUDA 8 being available, the advanced Pascal architecture can further exploit the benefits of the UM model by introducing an on-demand page faulting and migration mechanism to support concurrent access from code running on either CPUs or GPUs [11]. Specifically, when the API *cudaMallocManaged*() is used to allocate data space in the memory, there is no physical memory space allocated on either host or device. Instead, the memory will be allocated and migrated on-demand when the processor encounters a page fault during memory access. As the on-demand memory allocated, new page table entries (PTEs) are created in the GPU page table and these entries are validated upon completion of migration [12]. However, on-demand paging is not supported on the integrated Jetson GPU [13] as of now. On iGPU platform, there typically exists the data page mapping process. Specifically, after the CPU calles the API *cudaMallocManaged*() to allocate the memory space for the data, the data populates in the CPU side and the page tables covering the data are set valid. Then if the GPU tries to access the data, the page tables on CPU side are set invalid first and then remapped to the GPU side. After the page tables are set valid and the data populates in GPU side, the GPU can access the data smoothly [14], [15]. We will discuss the detail of the UM memory management in Section III-B.

Besides, another memory management method, the host-pinned memory [i.e., the API *cudaHostAlloc*() in CUDA programming] also provides the benefit of avoiding explicit data copy and reducing memory footprint [8]. However, the method bypasses the CPU and GPU side caches to directly access the memory, causing significantly latency. We will not discuss this method in our work.

B. Integrated CPU/GPU Platforms

NVIDIA proposes its Jetson line of iGPU platforms targeting autonomous systems and embedded intelligence and Jetson iGPU platforms include a series of Tegra SoCs, such as Parker and Xavier. Typically, the iGPU platform shares a physical memory between the CPU and GPU cores.

NVIDIA Parker SoC: Fig. 1(b) illustrates the NVIDIA Parker SoC embedded platforms, such as Jetson TX2 and NVIDIA Drive PX2 [16], which are widely practiced in many autonomous drone or vehicle applications (e.g., used in Volvo XC90 and Tesla Model S [17]). NVIDIA Parker SoC contains a "big.LITTLE" CPU cluster composed of two Denver2 and four ARM Cortex A57 cores. The chip has an integrated twocore Pascal GPU with 256 CUDA cores connected via an internal bus. The integrated TX2 platform shares up to 8 GB of LPDDR4 memory with up to 50 GB/s of bandwidth in typical applications.

NVIDIA Xavier SoC: NVIDIA Xavier is the latest SoC from NVIDIA used in Xavier AGX and Drive AGX Pegasus platforms. Xavier has an 8-core "Caramel" CPU (based on ARM V8) and an 8-core Volta GPU with 512 CUDA cores, and 16 GBs of shared main memory. We include both NVIDIA Parker SoC and AGX Xavier SoC in our evaluation. Note that even though our motivation and evaluation are based on NVIDIA Parker and Xavier, our methodology is applicable to all the integrated CPU/GPU architectures that utilize a UM model.

Due to the limited physical memory on these iGPU platforms, UM model can be utilized to reduce the memory footprint. Even though the UM model is adopted, which allows the CPU and GPU to share one pointer to access the allocated data region and avoids the explicit data copy between the CPU and GPU, the data initialization process is still implemented on CPU side by default, which is a programming legacy from the conventional copy-then-execution model. As we demonstrate later, the CPU-initialization on CPU side can contribute to significant latency to the overall application latency. It is actually unnecessary to implement the data initialization on CPU, especially with the support of the UM model on the iGPU platform.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

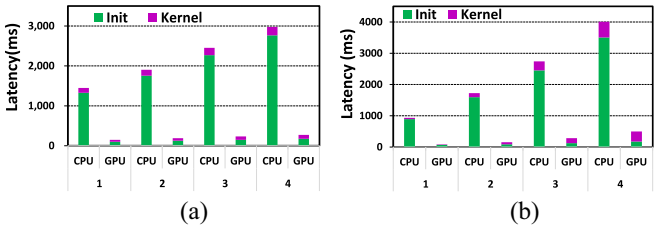


Fig. 3. Average initialization and kernel execution time in the cases of CPU-side and GPU-Init under UM. (a) MatAdd. (b) MatMul.

III. Characterization and Motivation

In this section, we mainly characterize the significant latency caused by the conventional CPU-side data initialization as well as other hidden latency in the UM model, which can strongly motivate our design of the latency-aware data initialization framework.

A. Data Initialization Latency

As we stated above, an application typically initializes data on CPU by default. Intuitively, data initialization is the process of assigning specific values to the defined variables. For example, assigning variable a with constant or randomly generated values [i.e., variable a = rand()]. Conventionally, it's an action of CPU filling the input buffer. To explore the latency introduced by CPU-Init, we breakdown the execution time of two representative micro-benchmarks, Matrix Add (*MatAdd*) and Matrix Multiplication (*MatMul*), for both copy-and-execute model (Def.) and UM model (UM), and analyze the latency of the initialization part and other part in detail. We implement the experiments on NVIDIA TX2 platform and use *gettimeofday*() to extract the time of each phase. Specifically, in the Def. model, we divide the entire benchmark response time into two parts: 1) data initialization time (init.) and 2) other times (others) that covers the data migration between CPU and GPU as well as kernel execution on GPU. In the UM model, since there is no data copy on iGPU platform, the entire benchmark time includes initialization time (init.) and other time (others) that only covers kernel execution on GPU.

The result is shown in Fig. 2, where the x-axis indicates the increasing input data size. The 1 to 4 corresponds to the matrix size equaling 9000*9000, 12000*12000, ... and 18000*18000, respectively. We run 10 times and report the average latency value. We also report the longest-observed values in Table II. More specifically, the bars in Fig. 2(a) and (b) show the latency of each part in the two models for benchmarks *MatAdd* and *MatMul*, respectively, which corresponds to the y-axis on the left. For *MatAdd*, we can observe that the data initialization latency is nontrivial in the Def. model, which can compete with the latency of other parts. With the data size increases, the data initialization latency increases drastically. Intuitively, the CPU has to process a larger amount of data, thus leading to greater latency. In the UM model, the latency of data initialization is also significant, which even dominates the entire benchmark response time due to the fact that the kernel execution latency of the two benchmarks are reduced. For *MatMul*, in either Def. model or UM model, the data initialization latency almost follows the same pattern.

TABLE III
Initialization and Kernel Execution Time in the Cases of CPU-Side and GPU-Init. Each Cell (*a, b*) Indicates the Init. and Kernel Latency, Respectively

Longest	MatAdd				MatMul			
	1	2	3	4	1	2	3	4
CPU	(1393,126)	(1844,153)	(2384,192)	(2906,218)	(937,42)	(1676,135)	(2575.301)	(3679,530)
GPU	(104,52)	(132,67)	(156,89)	(183,102)	(72,18)	(91,72)	(132,168)	(186,337)

Besides, we calculate the ratio of the initialization-latency in the entire benchmark response time for both two models, as indicated by the dots in Fig. 2(a) and (b), corresponds to the y-axis on the right. For both benchmarks, the ratio of initialization latency in the runtime has been around 50% in Def. model, and even been around 90% in the UM model. This indicates that the CPU-Init latency is exacerbated in UM model.

Observation 1: As the programming legacy of the copy-then-execute model on the dGPU platform, data initialization is typically implemented on CPU side. Obviously, the CPU-side data initialization process can result in significant execution latency and prolong the entire workload response time. In the UM model, the data initialization latency dominates the overall benchmark response time. In fact, UM model on the iGPU platform does not require explicit copy of the initialized data from CPU side as the Def. model does, the initialization may not be necessarily performed on CPU side, which provides opportunities to reduce the initialization latency as well as overall benchmark response time.

B. Kernel Latency

As we stated above, in the UM model, GPU can utilize the pointers shared with CPU to access the data in the allocated memory region. Also, it is GPU kernels instead of CPU ones that access the data after initialization in most cases. Therefore, it is possible to perform data initialization on the GPU instead of the CPU. Furthermore, GPU side initialization is feasible because data initialization is typically well-structured and paralleled. Initializing data on GPU may benefit significantly due to GPU's acceleration in parallel computing. Therefore, we propose to initialize the data on GPU side using an initialization kernel under the UM model and compare the latency of each part with the latency of the conventional CPU-Init case. Similarly, we utilize the two benchmarks and implement the GPU-Init on TX2, as show in Fig. 3. The x-axis indicates the increasing data size, which corresponds to the size in Fig. 2, and y-axis indicates the specific latency. We run 10 times and report the average latency value. We also report the longest-observed values in Table III. Compared to the conventional CPU-Init case, in the GPU-Init case, the initialization latency is reduced up to 93.7% for the benchmark *MatAdd* and 95.0% for the benchmark *MatMul*, respectively. Furthermore, the kernel execution time also is surprisingly reduced. We deep dive this and find that the reduction of kernel execution time is benefited from the migration of the page mapping. More specifically, Fig. 4 shows the specific process. The figure on the top demonstrates the conventional CPU-Init case. Since

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

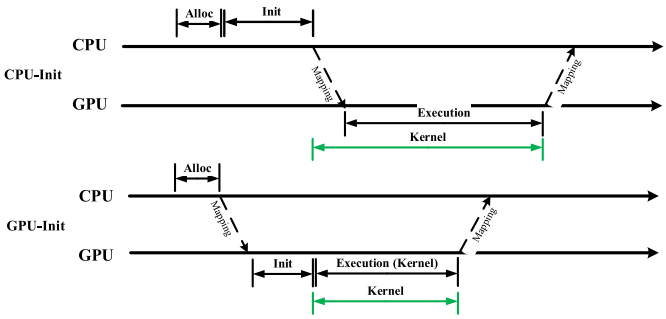


Fig. 4. Process of CPU initialization and GPU initialization process under UM model on iGPU platform.

the data is initialized on the CPU side, the pages covering the data has to been mapped from the CPU side to the GPU side before the kernel executes [18]. Therefore, the kernel time here includes both the mapping latency and the execution latency. Also, based on our characterization, the mapping latency even is larger than the kernel execution latency. In comparison, the figure on the bottom demonstrates the proposed GPU-Init. Since the initialization is implemented in the init kernel on GPU side, the page covering the data has been mapped to the GPU before the following kernel executes. As a result, the kernel time only includes execution time. Overall, the entire benchmark response time is reduced as well. For the benchmark *MatAdd*, the latency is reduced up to 91.0%, and for the benchmark *MatAdd* the latency is reduced up to 90.7%.

Observation 2: Under the UM model, if the GPU-Init is introduced, the initialization latency is significantly reduced compared to the conventional CPU-Init case. Furthermore, the iGPU platform inherently introduces extra latency on kernel launch due to the address translation and page mapping. The GPU-Init can premap the page covering the data before the real kernel execution, thus benefiting the kernel execution latency performance as well as the entire application's response time.

IV. Design

In this section, we propose a data initialization framework for an integrated CPU/GPU platform, which can choose the best data initialization mode for an application with the goal of optimizing the initialization latency performance of the application. Our design consists of three key components. First, we develop three different data initialization modes, namely, the CPU initialization mode (CPU-Init), GPU initialization mode (GPU-Init), and GPU-CPU hybrid initialization mode (Hybrid-Init), and then discuss the execution details and advantages of each mode. Second, we explore the workload and system features that can impact the latency performance of these initialization modes. Third, we establish an affinity estimation model to help framework wisely choose the best initialization mode for an application.

A. Data Initialization Modes

In the conventional copy-then-execute model, two separate data-size space is allocated on CPU and GPU side. The data initialization has to be implemented on CPU side before the data is copied from CPU to GPU. Under an UM model, although the data is not required to be explicitly copied from CPU to GPU, the data initialization is still implemented on the CPU side by default, which is a programming legacy. We consider this mode as the *CPU-Init* mode, as shown by *CPU_Init_Mode* in Algorithm 1.

Algorithm 1 Pseudocode the Three Initialization Modes

CPU_Init_Mode

```
float* data;
cudaMallocManaged(&data, size); {Allocate memory in UM model}
cpu ini(data,···); {Initialize data on CPU side}
kernel exe<<<Grid, Block>>>(data,···); {Specific kernel execution on GPU}
cudaFree(data); {Free memory in UM model}
```

GPU_Init_Mode

```
float* data;
cudaMallaocManaged(&data, size); {Allocate memory in UM model}
gpu ini<<<Grid, Block>>>(data,···); {Initialize data on GPU side}
kernel exe<<<Grid, Block>>>(data,···); {Kernel Execution on GPU}
cudaFree(data); {Free memory in UM model}
```

Hybrid_Init_Mode

```
float* data;
cudaStreamCreate(&s1); {Creates the CUDA stream s1}
cudaMallaocManaged(&data, size); {Allocate memory in UM model}
cpu ini(data,···); {Initialize partial data on CPU side}
gpu ini<<<Grid, Block, s1 >>>(data + offset,···); {Initialize partial data on GPU side}
cudaStreamSynchronize();
kernel<<<Grid, Block>>>(data,···) {Kernel execution on GPU}
cudaFree(data) {Free memory in UM model}
```

As a matter of fact, the CPU and GPU cores share one physical memory on the integrated CPU/GPU platform. Also, with the support of UM, only one data-size region is allocated in the memory space. Both CPU and GPU utilize one shared pointer to access the region. Since it is unnecessary to implement the initialization process on the CPU side in this situation, we propose to initialize the data on the GPU side, which is named as *GPU-Init* mode. Specifically, we implement a kernel to achieve the initialization process on GPU, as shown by *GPU_Init_Mode* in Algorithm 1.

Apart from the CPU-Init mode and GPU-Init mode, with the support of CUDA concurrent streams and multithreads [19], we propose to implement the data initialization by CPU and GPU cooperatively, which we consider as the *Hybrid-Init* mode, as shown by *Hybrid_Init_Mode* in Algorithm 1.

Generally, in three initialization modes, the API *cudaMallocManaged*() is called by the CPU to allocate memory for the initialized data. Then, the pointer *data* can be shared by the GPU to directly access the allocated data region. More specifically, in CPU-Init mode, a subfunction, *cpu_init*() is implemented to initialize the data. In GPU-Init mode, a kernel, *gpu_init*() is called by the CPU to initialize the data. In Hybrid-Init mode, both the subfunction *cpu_init*() and the kernel *gpu_init*() can execute concurrently to cooperatively initialize partial data. We will demonstrate later that the cooperative CPU-initialization and GPU-initialization in the Hybrid-Init mode can significantly outperform the other two initialization modes in some cases.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

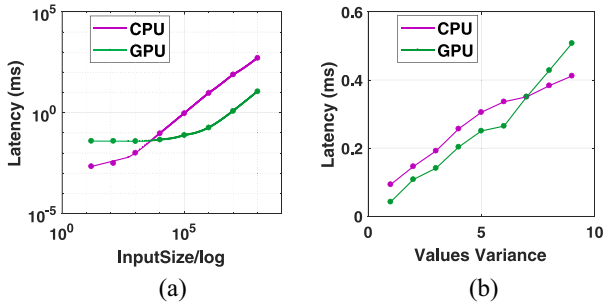


Fig. 5. Impact of (a) input data size and (b) initialization diversity on initialization latency performance (average latency).

On the other hand, we observe that the specific features and requirements of different applications can impact the execution time of different initialization modes. In most cases, data initialization is a process of assigning specific values to input variables. For some large-size workloads, the initialization process is highly paralleled and structured, thus GPU-Init mode can benefit this kind of initialization considering that GPU is good at highly paralleled operations. In contrast, some initialized data is in small scale and less structured, and CPU-Init mode can be chosen due to that CPU is better at dealing with complex and divergent operations. Besides, some initialization cases are much more complicated. Some partial initialization may be less paralleled and unstructured while the other part is highly structured. Hybrid-Init mode is thus suitable for this kind of initialization. Therefore, we explore the workloads characteristics and system features that can impact the latency performance of the different initialization modes in Section IV-B.

B. Impacting Factors

Regarding the initialization process, many factors, such as the initialization scale, the structure of the initialized data, etc., can influence the initialization latency performance. Also, when the initialization process is implemented on different platforms, it may perform differently. Therefore, it is important to understand how these different workloads and system features impact the latency performance of the initialization process before an appropriate initialization mode can be chosen. Here, we mainly consider four factors, the *input data size*, the *initialization diversity*, the *platform capability*, and *CPU–GPU assignment ratio*.

Input Data Size: Intuitively, initialization latency varies as the data size changes. Therefore, we first explore how the input data size impacts initialization latency performance. We implement the CPU-Init and GPU-Init modes on TX2 to initialize a matrix with different input data sizes, as shown in Fig. 5(a), where x-axis represents the increasing data size for the input matrix and y-axis represents data initialization latency. We run 10 times and report the average latency value. Also, we report the longest-observed values as below, CPU:0.002, 0.003, 0.01, 0.101, 0.100, 9.646, 50.603, 536.012, GPU:0.043, 0.043, 0.041, 0.049, 0.083, 0.187, 1.253, 11.691. The values correspond to the eight points of the two lines in the figure. Obviously, if the input size is large enough (e.g., over 4096), GPU-Init outperforms CPU-Init. Typically, GPU has a large number of parallel cores and can be scaled up well to handle the increasing amount of data initialization requests while CPU has limited cores and can be easily saturated by the increasing requests. On the other hand, if the input size is small (e.g., below 4096), CPU-Init surprisingly has better performance than GPU-Init. That is because GPU-Init mode has higher inherent overheads. For example, GPU-Init has to be launched from the host CPU and then scheduled and dispatched to the GPU processing cores to execute, in which kernel launching, scheduling, and dispatching lead to unavoidable latency for GPU-Init. When the input data size is small, GPU computation units cannot be fully utilized and thus, the benefits of GPU-Init from parallel cores will be offset and CPU-Init becomes a better choice. Besides, we conduct the same test on AGX platform and observes that the CPU-Init mode and GPU-Init mode follows the same track. That is, if the input size is small, the CPU-Init mode outperforms GPUInit mode, if the input size is relatively large, GPU-Init mode performs better. The threshold lies in the input size equaling about 14208.

Initialization Diversity: As we discussed above, some initialized data is highly paralleled and structured and while some others are less paralleled and even unstructured. Therefore, we consider an indicator here, initialization diversity, to measure how these difference impacts the initialization latency performance. More specifically, initialization diversity is defined as the values variance of the input data. We consider a common case here: 1) assigning value a to all variables and 2) assigning value a to some variables but value b to the other variables. The two initialization cases show different initialization diversity and thus may have different initialization latency. Intuitively, low initialization diversity indicates the variables to be initialized have less variance, are well paralleled and easy to be assigned with structured values. In contrast, the variables with high initialization diversity have complicated structures and requires more operations to be initialized.

To clearly demonstrate the influence of data diversity/variance on the initialization latency performance, we conduct experiments using both CPU-Init mode and GPU-Init mode to initialize a matrix with different data diversity/variance. The input data size is set 10 240 and the result is shown in Fig. 5(b), where the x-axis represents the initialization diversity/values variance of the input matrix and the y-axis represents data initialization latency. For example, $x = 6$ indicates that there are 6 different values being distributed in the matrix. In this case, the CPU-Init mode and GPU-Init mode causes about 0.34 and 0.28 ms latency, respectively. We run 10 times and report the average latency value. Also, we report the longest-observed values as below, CPU:0.098, 0.153, 0.201, 0.269, 0.320, 0.353, 0.368, 0.403, 0.433, GPU:0.044, 0.113, 0.148, 0.213, 0.263, 0.278, 0.368, 0.450, 0.533. The values correspond to the nine points of the two lines in the figure. Although the initialization latency of both CPUInit and GPU-Init increases with the initialization diversity increasing, we can find that GPU-Init mode performs

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

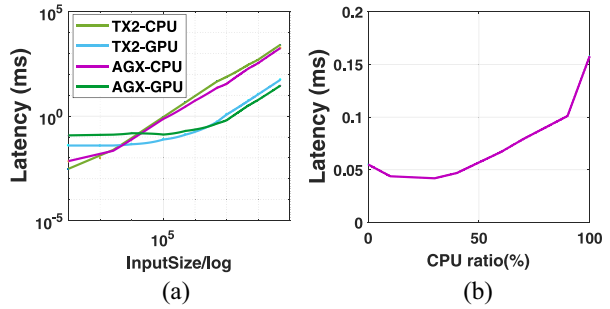


Fig. 6. Impact of (a) platform capability and (b) CPU–GPU assignment ratio on the average initialization latency performance.

well in initializing low-diversity data while CPU-Init mode outperforms GPU-Init mode in initializing data with high diversity. That is because GPU is throughput-oriented in the design and powerful in dealing with highly structured data, which can initialize the low-diversity data efficiently. The increasing diversity leads to the increasing divergence, which can degrade the GPU computation performance. On the contrary, CPU is latency-oriented and equipped with powerful branch prediction and locality capability, which can deal with high-diversity data more efficiently than GPU.

Platform Capability: Nvidia has issued different generations of Jetson series iGPU platforms. Here, we mainly investigate two types of iGPU platforms: 1) the Parker SoC-based TX2 and 2) the Xavier SoC-based AGX. The two platforms have different micro-architectures (i.e., TX2 is Pascalbased while AGX is the Volta-based), and thus, have different computing capability defined by Nvidia (i.e., TX2 is 6.2 and AGX is 7.2). Also, under UM model, on-demand paging is supported by Volta architecture. To further explore how platforms influence the initialization performance, we conduct GPU-Init mode on TX2 and AGX, respectively. Fig. 6(a) shows the results, where the x-axis represents input data size and the y-axis represents the initialization mode latency. We observe that the latency of GPU-Init on the two platforms are not exactly the same even though the latencies of both platforms increase with the data size increasing. When the data size is small, GPU-Init mode on TX2 has lower initialization latency than GPU-Init mode on AGX. However, GPU-Init mode on AGX outperforms the mode on TX2 when the data size reaches a relatively large threshold. Basically, the different microarchitectures as well as the different supporting mechanisms for the UM of the two platforms can lead to different initialization performance. Although AGX is the latest iGPU platform, it counter-intuitively shows worse performance than TX2 under the small data size, this should be attributed to the extra overhead of Volta architecture. Apart from the GPU-Init mode, we compare the CPU-Init mode on both platforms as well, as shown in Fig. 6(a). We observe that the CPU-Init mode latency on two platforms almost follow the same track. The latency performance on the two platforms are very close with each other.

CPU–GPU Assignment Ratio: As we discussed above, the CPU–GPU Hybrid-Init mode may bring performance benefits in the case of initializing partially structured data. However, it is nontrivial to effectively divide the initialization work between CPU and GPU such that the latency performance of the initialization process can be optimized. Here, we define the CPU–GPU assignment ratio to describe how the initialization work is divided between CPU and GPU. The ratio indicates how much data is assigned to be initialized on CPU side in the entire initialization work. For example, 0% represents the complete GPU-Init mode while 100% represents the complete CPU-Init mode, and 30% indicates that 30% of data is initialized on CPU side and the remaining 70% data is initialized on GPU side. Then, we implement the Hybrid-Init mode on a matrix with the input size of 10 240 on TX2 platform to preliminarily explore how the CPU–GPU assignment ratio impacts the initialization latency performance. The result is shown in Fig. 6(b), where the x-axis indicates the specific CPU–GPU assignment ratio and the y-axis indicates the corresponding initialization latency. Considering that the CPU part initialization and GPU part initialization can be implemented in parallel, the latency only indicates the maximal initialization latency between the two parts (i.e., the maximal value between the CPU-Init and GPU-Init latency).

We observe that there exists a minimum point (i.e., 30%) in the curve, indicating an optimal ratio between CPU and GPU assignment. When CPU–GPU assignment ratio is less than 30%, the initialization latency decreases as the ratio increases. However, when the ratio is larger than 30%, the latency increases drastically with the growing CPU ratio and even reaches the maximum as the ratio equals 100% (i.e., a full CPU-Init mode). This is because the GPU-Init part dominates the initialization work when the input data size is small in the beginning (i.e., before 30%). Thus, the decreasing GPU-Init ratio results in the decreasing initialization latency. However, as Fig. 5(a) shows that the CPU-Init mode causes significantly larger initialization latency than GPU-Init mode under this input data size, therefore, with CPU-Init ratio increasing, the CPU-Init part dominates the initialization work and contributes to the increasing initialization latency eventually.

Furthermore, we explore how the optimal ratio in the CPU–GPU assignment changes with the varying input data sizes, as is shown in Fig. 7, where the x-axis indicates the increasing ratio of CPU-Init part and the y-axis indicates the initialization latency. We can observe that as the input data size decreases from 20480 to 2048, the optimal ratio has an increasing range from 0% to 100%. When the input size reduces to 2048, the Hybrid-Init mode with any CPU–GPU assignment ratio (i.e., the CPU-Init mode if the ratio $= \%100$) outperform GPU-Init mode in terms of initialization latency. That is because when input data size decreases, CPUInit mode gradually causes lower latency than GPU-Init mode. Thus, the increasing ratio of CPU-Init part results in increasing initialization performance and the latency is even the minimal if the CPU-Init part ratio is 100%.

Summary: Although we mainly utilize the example of initializing a matrix to explore how the workloads and system features impact the initialization mode performance, the results envision to generalize in other types of initialization cases in a specific application program. In practice, under these factors synergistic impact, it may not be easily solved by

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

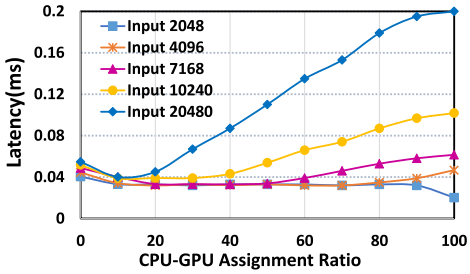


Fig. 7. Optimal CPU–GPU assignment ratio varies under different input data size with diversity = 2 in Hybrid-Init mode. The x-axis indicates the ratio of data assigned to CPU side to be initialized.

simple trialand-error solutions to select a best-matching mode to optimize the initialization performance for an application.

C. Automatic Mode Selection

Affinity Score Computation: From Section IV-B, we find that several workloads and system features can impact the latency performance of different initialization modes. It is nontrivial to decide which initialization mode is the best for an application to optimize the latency performance. Therefore, we develop an indicator, *Affinity_Score*, as shown in (1), which integrates all these impacting factors together to estimate the affinity to a certain initialization mode for an application

$$\text{Affinity_Score} = \frac{\lambda * \text{Input Size}}{\text{Initialization Diversity}} * \text{Platform Capability}. \quad (1)$$

In (1), the platform capability represents the computing capability of the platform. We adopt the official statistics provided by NVIDIA that the Jetson TX2 and AGX is 6.2 and 7.2, respectively, [20]. The parameter λ depends on specific applications. By utilizing the affinity estimation model, the initialization framework can holistically consider initialized data characteristics and workload features as well as platform capability to wisely choose the best-matching initialization mode based on the *affinity_score* to optimize the initialization performance of the application.

We classify the *affinity_score* into three bins as shown in Table IV. *Low* range indicates that the initialized data is less structured (e.g., in small input size with high diversity) for the chosen platform if all impacting factors being taken into account, while *High* range indicates that initialized data is more paralleled and structured (e.g., in large input size with low diversity) for the chosen platform to deal with. Basically, if the *affinity_score* is in *Low* range, CPU-Init mode is preferred while if the *affinity_score* is in *High* range, GPU-Init mode can be more suitable. Regarding the *Medium* range score, the Hybrid-Init mode can be considered to initialize data.

For example, for breadth-first search (BFS) benchmark, the input is 36k in size and 4 in diversity, if it's running on TX2 platform, we can calculate that its *affinity_score* is 55800 with λ set 1. We can find that the *affinity_score* falls into *Medium* range, which means Hybrid-Init mode should be chosen. According to our characterizations, the CPU-Init mode, Hybrid-Init mode, and GPU-Init mode cause 2.3, 1.08, and 1.8 ms latency, respectively. We can observe that the Hybrid-Init mode outperforms the other two initialization modes. Similarly, if the same data is initialized on AGX, we can calculate the *affinity_score* is 64800, indicating the GPU-Init mode should be chosen. Based on our characterizations, the CPUInit mode, Hybrid-Init mode, and GPU-Init mode cause 1.8, 1.17, and 0.8 ms latency, respectively, which supports our affinity estimation. Besides, for an application, if its estimated affinity score is equal to 1024 or 61440, the CPU-Init or GPU-Init mode is chosen instead of the Hybrid-Init mode as the CPU-Init/GPU-Init mode is comparative to the Hybrid-Init mode in performance but relatively simpler in execution.

TABLE IV
CLASSIFICATIONS OF ESTIMATED AFFINITY SCORE INTO BINS

Classification Bins	Low	Medium	High
Range	$\leq$ 1024	1024-61440	$\geq$ 61440

Execution Time Prediction: As we stated above, if CPU-Init mode or GPU-Init mode is chosen, the system can directly implement the mode as in Algorithm 1 to optimize the latency performance of the initialization process. However, if HybridInit mode is chosen, it concerns how to effectively divide the initialization work between CPU and GPU and find the optimal CPU–GPU assignment ratio such that the initialization performance can be optimized in the Hybrid-Init mode.

Here, instead of trying different the CPU–GPU assignment ratio, we profiling-based model by testing a large amount of data to predict the execution time of the CPU-Init and GPU-Init mode for an application with the given input data size and diversity on a platform. The prediction model is a key supplement of the affinity estimation model in our initialization-latency-aware framework, which can help system automatically determine an optimal assignment ratio for a Hybrid-mode-affiliated application. Specifically, (2) and (3) show the predicted GPU-Init execution time and CPUInit execution time, respectively. The N indicates the input size/diversity and the others (e.g., a, b, c, d, e, f) indicate profiled parameters inclusive of platform capability effect. The maximal error rate is about 12%. As CPU initialization part and GPU initialization part are executed in parallel, the execution time of Hybrid-Init is the maximum value between them. Theoretically, we can predict the optimal CPU–GPU assignment ratio by making the latencies of CPU initialization part and GPU initialization part equalled by leveraging our time prediction model as in (2) and (3). In practice, the predicted value may not be exactly the same to the absolute optimal ratio due to the resources contention, etc. However, as Fig. 7 shows that within an optimal ratio range, the latency performance of the Hybrid-Init mode does not make a significant difference. More importantly, the fact is that the Hybrid-Init mode with the optimized assignment ratio can significantly outperform the other two initialization modes

$$T = ae^{bN+c} + dN^2 + eN + f \quad (2)$$

$$T = aN^b + f\,. \quad (3)$$

Summary: The overall workflow of the latency-aware data initialization framework is shown in Fig. 8. For a given CUDA program, the system first considers the input data size,

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

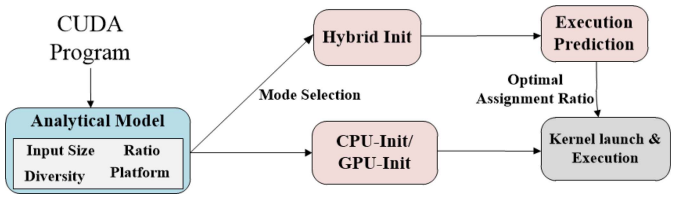


Fig. 8. Workflow of the latency-aware data initialization framework.

TABLE V
INITIALIZATION MODE SELECTION FOR BFS

Test	Input	Platform	Diversity	Predicted	CPU(ms)	Hybrid(ms)	GPU(ms)
1	0.5k	TX2	4	CPU-Init	**0.018**	0.022	0.082
2	1k	TX2	8	CPU-Init	**0.028**	0.038	0.094
3	1k	AGX	4	CPU-Init	**0.058**	0.068	0.226
4	16k	TX2	4	Hybrid-Init	0.347	**0.121**	0.200
5	16k	TX2	8	Hybrid-Init	0.625	**0.281**	0.510
6	16k	AGX	8	Hybrid-Init	0.207	**0.159**	0.388

the diversity of the initialized data and the specific implementation platform capability and utilize (1) to estimate the affinity score of the application and then decide whether CPU-Init/GPU-Init mode or Hybrid mode is potentially the best-matching. If the CPUInit or GPU-Init mode is selected, the system launches the corresponding initialization function in the mode as indicated in Algorithm 1 to complete the initialization process. However, if the Hybrid-Init mode is selected, the system will take advantage of execution time prediction model to predict the latency of the CPU-Init mode and GPU-Init mode as well as refer to the optimal range to identify the optimal CPU–GPU assignment ratio to further optimize the initialization latency performance of the CUDA application.

Besides, multiple applications may arrive dynamically, our initialization framework can address this situation as well. As our system design targets the autonomous workloads such as the object detection tasks in driving automation system, such tasks may possess some traceable input data features. For example, for the image-detection applications, the input feature map is typically among the cases of 256*256, 512*512, or 1024*1024 regarding the resolution and the data diversity is limited up to 256 regarding the gray-scale values. Basically, our initialization framework can presuppose these input settings to quickly identify the input data features information once the application arrives. As a supplementary, we can offline profile more typical applications in advance, or utilize such tools as *imresize*(), *imadjust*() in MATLAB to adjust the input size and data diversity to a range to speedup the process of identifying the application's input features information, though some overhead may be caused. This way, the system can greatly reduce the affinity estimation time and choose the initialization modes more quickly, which benefits the system to deal with the incoming applications dynamically. As a matter of fact, this is a tradeoff behind the performance and accuracy in practice.

V. EVALUATIONS

A. Test Hardware Platforms and Workloads

The evaluation of the proposed affinity estimation model performance and the efficacy of the framework is implemented on both NVIDIA Parker and Xavier platforms. We use Jetson TX2 and AGX as the representatives for Parker and Xavier SoCs. These platforms target autonomous machines application and are widely deployed in many drone [21] and autonomous driving scenarios.

Concerning the workloads, as the initialized data can be directly copied from another buffer (e.g., assign constant to the data) or be loaded from a file (e.g., read values from an existing file). We consider two cases here: the initialized data is 1) directly assigned with random/specific values (e.g., being copied from another constant buffer) or 2) loaded from the input files (e.g., read the input feature maps for CNN model). Therefore, we use both well-established Rodinia benchmarks suite and a CNN instance YOLO to illustrate the two cases. First, we utilize Rodinia benchmark Suite [9] because it includes applications and kernels targeting multicore CPU and GPU platforms and covers a diverse range of application domains. Each program in the Rodinia benchmark exhibits various types of parallelism and data-sharing characteristics as well as synchronization techniques, and correspondingly, the initialized data possesses multiple features. Second, we target our application scenario in autonomous machines (e.g., drone and autonomous driving). Therefore, we select a popular CNN instance in these applications, the object detection function YOLO [5] to evaluate our design in the second case.

Considering the targeted application scenario is typically safety-critical and time-sensitive, we mainly utilize latency as the metric to evaluate the design.

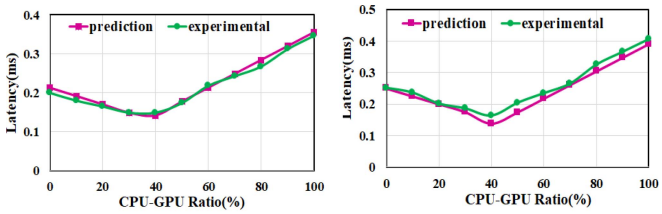
B. Case I: Rodinia Benchmarks

Basically, Rodinia benchmarks can be roughly categorized into two types: 1) memory-intensive and 2) computation-intensive [22]. We select two representative benchmarks from each category to evaluate the affinity estimation model and the initialization system performance. All benchmarks are executed under the UM model. For the given input data size, the diversity of the initialized data as well as the implementation platform (i.e., either TX2 or AGX), we first utilize our affinity estimation model to predict which initialization mode is chosen to optimize the initialization latency performance of the benchmark. Then, we initialize the benchmark input data by utilizing all of the three initialization modes and profile the initialization latency in each mode. By comparing the latency under different initialization modes, we can observe how these workloads and system features impact the selection of initialization mode. Also, by comparing the predicted mode and the optimal mode in real execution, we can validate our proposed affinity estimation model.

1) Memory-Intensive: BFS and Needleman–Wunsch (NW). BFS is a type of graph traversal algorithm and utilizes the BFS principle to traverses all the connected components in a graph. Needleman–Wunsch is a nonlinear global optimization method for DNA sequence alignments.

Initialization Mode Selection: Table V shows the result of the benchmark BFS. The first column in the table indicates

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

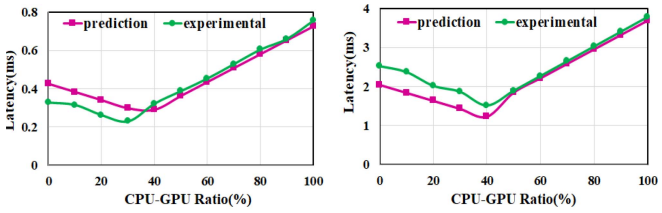


(a) (b)

Fig. 9. Prediction of the optimal CPU–GPU assignment ratio for (a) BFS and (b) NW on TX2.

TABLE VI
INITIALIZATION MODE SELECTION FOR NW

Test	Input	Platform	Diversity	Predicted	CPU(ms)	Hybrid(ms)	GPU(ms)
1	16	TX2	1	CPU-Init	**0.008**	0.015	0.074
2	16	AGX	1	CPU-Init	**0.024**	0.104	0.208
3	224	TX2	1	Hybrid-Init	0.408	**0.142**	0.192
4	224	TX2	2	Hybrid-Init	0.734	**0.294**	0.480
5	640	TX2	1	GPU-Init	4.112	1.234	**0.866**
6	640	AGX	1	GPU-Init	2.589	0.776	**0.576**



(a) (b)

Fig. 10. Prediction of the optimal CPU–GPU assignment ratio for (a) Gaussian and (b) HotSpot on TX2.

the test number and each row indicates a test case. Column 2 to column 4 indicate the input data size, platform, and initialized diversity, respectively. We utilize a tuple (*input-size, platform, diversity*) here to indicate the specific input features information. Column 5 indicates the optimal initialization mode selected by our analytical model. The last three columns indicate the measured latency in the experiments by applying CPU-Init, Hybrid-Init, and GPU-Init mode in BFS, respectively. For example, test1 indicates, that to initialize the data with (0.5k, TX2, 4) parameters, the predicted mode is CPU-Init. In the experiments, the initialization latency of the three modes is 0.018, 0.022, and 0.082 ms, respectively. Obviously, the CPU-Init mode causes the minimal latency (i.e., indicated by the *Bold*), and thus, is the best mode to optimize the initialization latency performance. By comparing the predicted mode with the measured latency of the three initialization modes in real execution, we can verify whether our analytical model can choose the best initialization mode for an application. In the first three test cases, we change the input data size, the initialization diversity, and platform capability, and we observe that our affinity estimation can accurately predict the optimal mode. Then, we implement test cases 5 and 6 with different input parameters, the results prove the accuracy of our model as well. Table VI shows the result of the benchmark NW.

Optimal CPU–GPU Assignment Ratio: As we discussed above, if we choose Hybrid-Init mode to initialize data, it is necessary to find the optimal CPU–GPU assignment ratio such that the initialization latency can be optimized. Therefore, we select a Hybrid-Init case of BFS on TX2 to verify whether our execution time prediction model can identify the optimal CPU–GPU assignment ratio for a given Hybrid-Init case. Fig. 9(a) shows the result, where the x-axis indicates the increasing CPU assignment ratio in the Hybrid-Init mode applied to the BFS benchmark, and the y-axis indicates the initialization latency. The two curves indicate the predicted execution time and experimental execution time for the given CPU–GPU assignment ratio. The bottom point on the curve indicates the optimal ratio identified by either the prediction model or experiment result. We can observe that our prediction model identifies the optimal ratio correctly for the benchmark BFS. In the test case, the CPU-Init mode causes larger latency than GPU-Init mode, therefore, in the beginning, the CPU assignment ratio (i.e., the ratio is less than 40%), the GPUInit part dominates the initialization latency. With the GPUInit part ratio decreasing, the initialization latency decreases as well. With the CPU ratio increases (larger than 40%), the CPUInit part dominates the initialization process and contributes to the increasing initialization latency. Fig. 9(b) shows the results of the optimal CPU assignment ratio on NW. It follows the same track as BFS does.

TABLE VII
INITIALIZATION MODE SELECTION FOR GAUSSIAN

Test	Input	Platform	Diversity	Predicted	CPU(ms)	hybrid(ms)	GPU(ms)
1	16	TX2	2	CPU-Init	**0.006**	0.013	0.064
2	16	TX2	4	CPU-Init	**0.010**	0.014	0.096
3	16	AGX	2	CPU-Init	**0.014**	0.023	0.229
4	256	TX2	2	Hybrid	0.470	**0.235**	0.433
5	256	AGX	2	Hybrid	0.381	**0.215**	0.430
6	1024	AGX	2	GPU-Init	3.803	0.951	**0.774**

TABLE VIII
INITIALIZATION MODE SELECTION FOR HOTSPOT

Test	Input	Platform	Diversity	Predicted	CPU(ms)	Hybrid(ms)	GPU(ms)
1	16	TX2	2	CPU-Init	**0.008**	0.131	0.438
2	16	AGX	2	CPU-Init	**0.014**	0.045	0.453
3	128	TX2	2	Hybrid-Init	0.544	**0.299**	0.659
4	128	TX2	4	Hybrid-Init	0.979	**0.621**	1.648
5	512	TX2	2	GPU-Init	2.970	1.485	**1.154**
6	512	TX2	4	GPU-Init	4.455	2.005	**1.731**

2) Computation-Intensive: Gaussian Elimination and HotSpot. Gaussian computes result row by row, solving for all of the variables in a linear system. HotSpot is a widely used tool to estimate processor temperature based on an architectural floor plan and simulated power measurements. Tables VII and VIII shows the initialization selection result on the benchmarks, Gaussian and HotSpot, respectively. We change the input parameters and then compare the predicted mode with the experimental results. We can observe that our affinity estimation accurately predict the best initialization mode for the two benchmarks.

Fig. 10(a) and (b) show the results of how we apply the execution time prediction model to two benchmarks to find the optimal CPU–GPU assignment ratio in the Hybrid-Init mode. Basically, the execution prediction model works well on the two benchmarks. However, as Fig. 10(a) demonstrates,

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

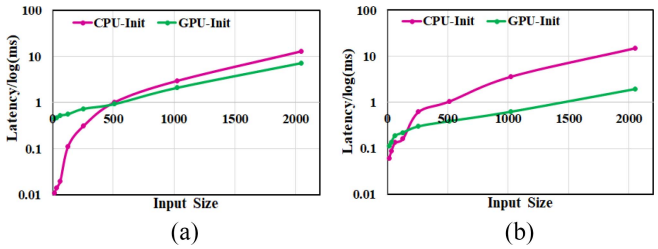


Fig. 11. Initialization latency performance of YOLO in CPU-Init and GPU-Init mode. (a) TX2. (b) AGX.

the experimental result indicates that the best ratio for the Gaussian case is about 40% while the model predicts that the ratio lies in 30%. Considering that CPU-Init mode causes significantly higher latency than GPU-init mode in the case, the best ratio exists within a very narrow range. Although the prediction model may not perfectly identify the best ratio, it identifies a ratio that approaches the optimal one very much.

C. CNN Model and Loading Input Feature Map

As we discussed that some workloads may directly load data from a file to initialize the input data, we consider evaluating the data loading process and measuring its latency performance under different initialization modes in this situation, such as a CNN model reading the input feature map. As a matter of fact, CNN models are widely deployed in the intelligent and autonomous machines to achieve such functions as object detection, tracking, localization, etc. Also, these CNN models need to load the input feature map at a batch and then implement the inference, for example, an object detection model loads a frame of data each time and implements detection function on the frame of data. Therefore, the overall object detection function is achieved in the premise of data loading. The loading process may significantly impact the overall application's performance and the optimization of the loading process under different initialization modes may benefit the overall application performance.

To straightforwardly show how the loading process contributes to the latency performance of the overall application, we demonstrate a binarized YOLO2 model [23] here. As the quantized CNN model consumes smaller memory footprint, but can provide more fast and efficient detection, the model is widely applied in the iGPU platform and edge computation. Therefore, we characterize the loading latency and the inference latency of the YOLO model using the feature maps with resolution 512*512 on TX2 platform and observe that the loading (i.e., CPU initialization mode by default) causes about 22.7 ms latency while the inference process only causes 7.1 ms latency. The loading process contributes about 76.1% latency to the overall YOLO2 performance. Obviously, if the loading latency can be optimized, the overall application performance can be further improved. On the other hand, as the binarized YOLO instance cannot provide high detection accuracy, we mainly test the loading latency as well as the performance of the overall application (i.e., the loading plus inference), which we consider a dummy YOLO instance.

Then, we comprehensively characterize the latency of the data loading of the YOLO instance under different initialization modes. More specifically, we input a gray-scale image and measure the data loading latency under CPU-Init mode and GPU-Init mode, respectively. The result is shown in Fig. 11, where the *x*-axis indicates the increasing input data size, which is derived by multiplying the number of feature maps and the resolution and the *y*-axis indicates the latency performance. Basically, we can observe that on both platforms CPU-Init mode causes less latency than GPU-Init mode in the beginning if the input size is small. Especially, on TX2, the latency of CPU-Init mode is significantly less than the GPUInit mode latency, while on AGX both latencies are competitive. However, with image size increasing, GPU-Init mode significantly outperforms CPU-Init mode on both platforms. On TX2, the latency of both modes increases quickly, while on AGX only the latency of CPU-Init mode increases drastically. Reasonably, AGX GPU cores adopt the advanced Volta microarchitecture and possess a more powerful capability than the Pascal micro-architecture adopted by TX2, leading to better performance when the input size is large, though the Volta demonstrates some hidden overhead when the input image size is small in the beginning.

On the other hand, we select the images with 512*512 resolution and apply the Hybrid-Init mode to evaluate its initialization latency performance. On TX2, we observe that the CPU-Init, Hybrid-Init, and GPU-Init mode cause the latency of 1.024, 0.450, and 0.926 ms, respectively. On AGX, the latency of the three modes are 1.061, 0.252, and 0.388 ms, respectively. Obviously, the Hybrid-Init mode outperforms other modes in this case. In practice, it is common to input 3channels RGB image feature map into YOLO instance. This way we will consider utilizing CPU-Init mode to initialize one or two channels data and GPU-Init mode to initialize the remaining channel data. This way, the Hybrid-Init mode may significantly benefit the data loading as well as the overall application performance of the YOLO instance.

VI. Related Work

Much work has been dedicated to analyzing the UM model on dGPU platforms. Pichai *et al.* [24], Vesely *et al.* [25] Flores [26], and Nielsen and Hussain [27] investigated unifying CPU–GPU memory space to decrease programming burden and increase system performance. Landaverde *et al.* [22] analyzed UM access pattern in CUDA programming and demonstrates that kernels should operate on the subsets of the output data to improve the application's performance. Meanwhile, some work has been done on UM model performance on the iGPU platform. NVIDIA officially summarizes the main characteristics of conventional and UM models on the Tegra platform, but it does not breakdown the overhead for each method [28]. Dashti and Fedorova [8] compared application performance under conventional and UM models on TK1 platform. Otterness *et al.* [29] compared the popular memory models for supporting real-time computer-vision workloads on TK1 and show that only in some scenarios can UM benefit the application. Li *et al.* [30] measured the

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.

performance loss of UM in CUDA on both integrated and dGPU systems and explore the underlying reasons. Dashti and Fedorova [8] compared the performance of applications that adopt different programming frameworks under the UM mechanism on iGPU system. Bateni *et al.* [15] analyzed the three memory management methods on NVIDIA iGPU platform and design guidelines to co-optimize memory footprint and system performance. However, none of them pay much attention to the data initialization part in an application and considers reducing the latency of the part to significantly benefit the entire application latency performance.

VII. Conclusion

In this article, we develop a latency-aware data initialization framework under UM management method for the integrated CPU/GPU heterogeneous platforms. The framework includes three different data initialization modes and utilizes an affinity estimation model to wisely decide the best-matching initialization mode for an application such that the initialization latency performance of the application can be optimized. Our extensive evaluations on two representative platforms shows the accuracy of the proposed affinity estimation model and efficacy of our initialization-latency-aware framework.

References

[1] Nvidia. *Nvidia Jetson Nano Developer Kit*. [Online]. Available: https://developer.nvidia.com/buy-jetson

[2] Nvidia. *Nvidia Drive—Autonomous Vehicle Development Platforms*. [Online]. Available: https://developer.nvidia.com/drive

[3] *Perception Matters: How Deep Learning Enables Autonomous Vehicles to Understand Their Environment*. [Online]. Available: https://blogs.nvidia.com/blog/2018/08/10/autonomous-vehicles-perception-layer/

[4] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 7263–7271.

[5] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018. [Online]. Available: arXiv:1804.02767.

[6] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 21–37.

[7] M. Bojarski *et al.*, "End to end learning for self-driving cars," 2016. [Online]. Available: arXiv:1604.07316.

[8] M. Dashti and A. Fedorova, "Analyzing memory management methods on integrated CPU-GPU systems," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 59–69, 2017.

[9] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[10] M. Harris. (2013). *Unified Memory in Cuda 6*. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/

[11] M. Harris. (2016). *Cuda 8 Features Revealed*. [Online]. Available: https://devblogs.nvidia.com/cuda8-features-revealed/

[12] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory," 2019.

[13] Trinayan. (2017). *On Demand Paging*. [Online]. Available: https://devtalk.nvidia.com/default/topic/1015688/jetson-tx2/on-demand-paging/

[14] N. Sakharnykh, "Everything you need to know about unified memory," 2018.

[15] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform," 2020. [Online]. Available: arXiv:2003.07945.

[16] Nvidia. (2018). *Hardware for Every Situation*. [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware

[17] Nvidia. *Px2 for Volvo Autonomous Driving*. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-s-deep-learning-car-computerselected-by-volvo-on-journey-toward-a-crash-free-future

[18] M. Harris. (Mar. 2017). *Unified Memory for Cuda Beginners*. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/#footnote

[19] Nvidia. *Cuda Concurrent Streams*. [Online]. Available: https://devblogs.nvidia.com/gpupro-tip-cuda-7-streams-simplify-concurrency/

[20] Nvidia. *Your GPU Compute Capability*. [Online]. Available: https://developer.nvidia.com/cuda-gpus

[21] DJI. (2019). *Tx2 for DJI Drone*. [Online]. Available: https://www.dji.com/newsroom/news/dji-manifold-2-onboard-supercomputer-transforms-drones-intoautonomous-robots

[22] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2014, pp. 1–6.

[23] AlexeyAB. *Light Version of Convolutional Neural Network Yolo v3 & v2 for Objects Detection*. [Online]. Available: https://github.com/AlexeyAB/yolo2 light

[24] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 743–758, 2014.

[25] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2016, pp. 161–171.

[26] V. G. Flores, "Memory hierarchies for future HPC architectures," 2017.

[27] M. J. Nielsen and Z. S. Hussain, "Unified memory computer architecture with dynamic graphics memory allocation," U.S. Patent 6 104 417, Aug. 15, 2000.

[28] Nvidia. (2019). *Cuda for TEGRA*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-for-tegraappnote/index.html

[29] N. Otterness *et al.*, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Proc. IEEE Real Time Embedded Technol. Appl. Symp. (RTAS)*, 2017, pp. 353–364.

[30] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on NVIDIA GPUs," in *Proc. 15th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. (CCGrid)*, 2015, pp. 1092–1098.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on December 10,2020 at 16:47:37 UTC from IEEE Xplore. Restrictions apply.