CCF Transactions on High Performance Computing (2024) 6:179–191
https://doi.org/10.1007/s42514-023-00172-w

REGULAR PAPER

OneGraph: a cross-architecture framework for large-scale graph computing on GPUs based on oneAPI

Shiyang Li[1] · Jingyu Zhu[1] · Jiaxun Han[1] · Yuting Peng[1] · Zhuoran Wang[1] · Xiaoli Gong[1] · Gang Wang[1] · Jin Zhang[1] · Xuqiang Wang[2]

Received: 5 May 2023 / Accepted: 10 October 2023 / Published online: 9 November 2023
© China Computer Federation (CCF) 2023

Abstract

The explosive growth of graph data sets has led to an increase in the computing power and storage resources required for graph computing. To handle large-scale graph processing, heterogeneous platforms have become necessary to provide sufficient computing power and storage. The most popular scheme for this is the CPU-GPU architecture. However, the steep learning curve and complex concurrency control for heterogeneous platforms pose a challenge for developers. Additionally, GPUs from different vendors have varying software stacks, making cross-platform porting and verification challenging. Recently, Intel proposed a unified programming model to manage multiple heterogeneous devices at the same time, named oneAPI. It provides a more friendly programming model for simple C++ developers and a convenient concurrency control scheme, allowing managing different vendors of devices at the same time. Hence there is an opportunity to utilize oneAPI to design a general cross-architecture framework for large-scale graph computing. In this paper, we propose a large-scale graph computing framework for multiple types of accelerators with Intel oneAPI and we name it as OneGraph. Our approach significantly reduces the data transfer between GPU and CPU and masks the latency by asynchronous transfer, which significantly improves performance. We conducted rigorous performance tests on the framework using four classical graph algorithms. The experiment results show that our approach achieves an average speedup of 3.3x over the state-of-the-art partitioning-based approaches. Moreover, thanks to the cross-architecture model of Intel oneAPI, the framework can be deployed on different GPU platforms without code modification. And our evaluation proves that OneGraph has only less than 1% performance loss compared to the dedicated programming model on GPUs in large-scale graph computing.

Keywords Heterogeneous programming · Graph computing · Out-of-memory process · Cross-architecture portability · OneAPI

1 Introduction

Graph computing has become an increasingly important field in recent years and has applications in many areas such as social networks (Rossi and Ahmed 2015), recommendation systems (Boldi et al. 2004), and biological networks (Kim 2012). As data becomes more complex and larger in scale, graph computing provides a powerful tool for analyzing and processing these data.

Graph processing algorithms involve a large number of computations, iterations, and memory accesses, which can be time-consuming and resource-intensive, particularly for large-scale graph data sets. There are many accelerating devices to deal with large-scale graph computing, including multi-core platforms, GPU platforms, and FPGAs, each with its own strengths. Facing the massive computation in large-scale graph computing, CPU along systems become overstretched due to a lack of computing resources. GPUs have become prevalent for accelerating large-scale graph computing in recent years. However, GPU devices generally have very limited memory to process a large-scale graph (Sahu et al. 2017). As a result, the CPU-GPU heterogeneous

Jingyu Zhu, Jiaxun Han, Yuting Peng and Zhuoran Wang contributed equally to this work.

✉ Xiaoli Gong
gongxiaoli@nankai.edu.cn

[1] Colleage of Computer Science, Nankai University, Tianjin 300350, China

[2] State Grid Tianjin Information and Communication Company, Tianjin, China

platforms become necessary to provide the required computing power and storage resources. Nevertheless, different GPU devices from different manufacturers have various graph computing accelerating solutions, they may differ in terms of hardware, software, programming languages, and APIs, among other aspects, making programming on a heterogeneous platform challenging.

Proficiency in a new programming language and model is often required when programming on a heterogeneous platform. For instance, CUDA toolkit (NVIDIA 2022) for NVIDIA GPUs, ROCm (AMD 2022) for AMD GPUs, and OpenCL (Khronos 2011) for Intel GPUs. This creates a steep learning curve for developers who need to become familiar with different programming languages and models for heterogeneous computing on GPUs. For instance, mastering CUDA programs and NVIDIA GPU architectures is widely acknowledged to be challenging. Furthermore, when attempting to port existing code to a different platform, such as migrating a CUDA program to an Intel GPU, the complexity only increases. In such cases, developers have to rewrite the CUDA program using OpenCL, making the process even more demanding. Worse still, porting across architectures often comes with performance fluctuations, traditional cross-architecture programming model like OpenCL usually has significant performance drop compared to dedicated programming model like CUDA.

Besides the burden of learning specific programming models, concurrency control and memory management are also troublesome for developers in large-scale graph computing. Due to limited global memory in GPU, developers must carefully allocate memory resources and deal with large amounts of data transfer. Graph partitioning is a widely adopted technique in prior studies to exploit the parallelism of GPUs for large-scale graph computing, and two primary approaches are typically employed. The first approach involves manual graph partitioning, dividing the graph into multiple subsets that can fit into the GPU memory. A second approach is through the Unified Virtual Memory (UVM) to oversubscribe GPU memory and implicitly transfer data. The graph data is allocated in the main memory and mapped to the GPU memory by the GPU driver. When the GPU requires this data, it will be transferred to GPU memory at a granularity of page(4KB-1 M). UVM can be regarded as an implicit graph partitioning technique where each partition corresponds to a page.

However, manual graph partitioning often leads to redundant data transfer and developers suffer from complex concurrency control when designing their systems. The main memory allocation and graph partitioning are processed on the CPU side and these part codes are usually written in C/C++ source files. Still, heterogeneous kernel codes about the GPU memory management, kernel launch, and computing process are written in another file. Concurrency control and synchronization will be a challenge to developers and significantly influence program performance. UVM can be regarded as an implicit graph partitioning, and it significantly reduces developers' burden because the GPU driver will deal with memory management and concurrency control automatically. However, studies (Kim et al. 2020; Harris 2021) have demonstrated that UVM can result in data thrashing and frequent page faults, which leads to significant overhead.

In summary, existing solutions for large-scale graph processing on heterogeneous platforms suffer from high learning costs, complex concurrency control, and poor portability.

Intel oneAPI (Intel 2023a) is an open, standards-based cross-architecture programming model that enables programmers to develop cross-architecture applications for CPUs, GPUs, and FPGAs with agility and efficiency.

In the oneAPI programming model, researchers can use a single programming language and programming model for multiple heterogeneous hardware platforms including different models of GPUs and FPGAs, so that they can focus on applying their ideas to next-generation innovations without having to rewrite their own software for new or next-generation hardware platforms. This will help reduce the time and effort required for developing and optimizing code for heterogeneous platforms, and enable developers to focus on algorithm design and performance optimization, rather than worrying about the underlying hardware and programming languages. oneAPI also provides efficient performance analysis and tuning tools for different hardware, and the performance analysis results in this paper were obtained using VTune Analyzer from oneAPI product.

In this paper, we propose *OneGraph*, a general cross-architecture framework for large-scale graph computing, implemented with oneAPI. We follow the state-of-the-art out-of-GPU-memory graph processing system (Tang 2021) to design our system. The details about the scheme will be illustrated in Sect. 3. We conducted rigorous experiments to evaluate *OneGraph* and verified its across-architecture portability. Results show that *OneGraph* achieves significant speedup over CPU algorithms and other large-scale graph processing schemes. Moreover, *OneGraph* can be ported to different GPU platforms without any code modification, we have successfully run it on NVIDIA GPU, Intel GPU, and AMD GPU. *OneGraph* also achieves a graceful balance between portability and performance. Our evaluation proves that *OneGraph* only has less than 1% performance loss with a smaller code size compared to the dedicated programming model in large-scale graph computing on CPU-GPU heterogeneous platforms.

In summary, we have made the following contributions.[1]

- We propose a cross-architecture large-scale graph computing framework and implement its prototype *OneGraph* with oneAPI.
- We discussed oneAPI's optimization on concurrency control and system synchronization in heterogeneous programming for large-scale graph computing.
- As a cross-architecture framework, *OneGraph* achieves a graceful balance between portability and performance. We verified it on three kinds of GPUs from different manufacturers. The results show that *OneGraph* achieves an average 3.3x speedup over the SOTA graph-partitioning approach on CPU-GPU platforms. The performance loss compared to the dedicated programming model CUDA is less than 1% on NVIDIA GPU.

The rest of this paper is organized as follows. Section 2 provides some background about large-scale graph computing on heterogeneous platforms and opportunities brought by oneAPI. Section 3 illustrates our system design and optimization. Section 4 presents our experiments, evaluations, and analysis. Section 5 concludes this work.

2 Background and related works

2.1 Heterogeneous platforms and programming model

As graph data size has increased dramatically in recent years (Sahu et al. 2017), heterogeneous platforms, such as CPU-GPU and FPGAs have been prevalent for large-scale graph processing because they can provide enough computing power and storage at the same time.

In the past heterogeneous programming work, if you wanted to port works to a new platform, you had to use a new specific language or a different programming model for rewriting software to run on the target hardware, which undoubtedly increases the time cost of cross-architecture development and verification, and also to some extent hinders the joint use of different hardware platforms, limiting the innovation of researchers.

Recently, Intel has proposed a cross-architecture programming model oneAPI (Intel 2023a) to address this problem. The oneAPI program is written in Data Parallel C++ (DPC++). It utilizes a similar structure to modern C++, following the SYCL (Khronos 2020) standards for data parallelism and heterogeneous programming. DPC++ is a single-source language where host code and heterogeneous kernel code can be mixed in the same source file. The C++/SYCL program is called on the host and offloads the task to the heterogeneous platform. It shields the differences between different hardware platforms and provides a unified scheme to call GPUs or FPGAs from different vendors.

Listing 1 shows a simple vector add code written in DPC++ and Listing 2 shows its CUDA version. Compared to CUDA, DPC++ follows the modern C++ style, which is more friendly to developers. The code size of DPC++ is also significantly reduced. When we need to port this code to a new platform, the only thing that needs to be done is to change the device selector in DPC++ (the first line in Listing 1).

Listing 1 VectorAdd in DPC++

```
//select a heterogeneous platform
auto selector = default_selector;
queue q(selector);//assign a queue to selected
    device
... /*malloc memory for arrays*/
void VectorAdd(queue &q, int *a, int *b, int *
    sum, size_t size){
    q.submmit([&](handler& h) {
        h.parallel_for (size, [=](id<1> tid){
            sum[tid] = a[tid] + b[tid];
        }
    })}).wait();//sync CPU and GPU
}
```

Listing 2 VectorAdd in CUDA

```
//choose an NVIDIA GPU according to your
    requirements
cudaDeviceProp devicePropDefined;
memset(&devicePropDefined, 0, sizeof(
    cudaDeviceProp));
cudaChooseDevice(&devicedChoosed, &
    devicePropDefined);
... /*malloc memory for arrays*/
VectorAdd_CUDA(int *a, int *b, int *sum,
    size_t size);
//below is written in another .cu file
"extern C"
__global__ void VectorAdd<<<blocks,threads
    >>>(int *a, int *b, int *sum, size_t size){
    int i = blockDim.x * blockIdx.x +
        threadIdx.x;
    if (i < size){
        sum[i] = a[i] + b[i];
    }
}
void VectorAdd_CUDA(int *a, int *b, int *sum
    , size_t size){
    threads = get_threads_per_block();
    blocks = get_blocks_per_grid();
    VectorAdd<<<blocks,threads>>>(a, b,
        sum, size);
    cudaDeviceSynchronize();//sync CPU and
        GPU
}
```

[1] This work is a redesign and optimization based on our previous work, which was demonstrated at the 50th International Conference on Parallel Processing (ICPP 2021) as "Ascetic: Enhancing Cross-Iterations Data Efficiency in Out-of-Memory Graph Processing on GPUs".

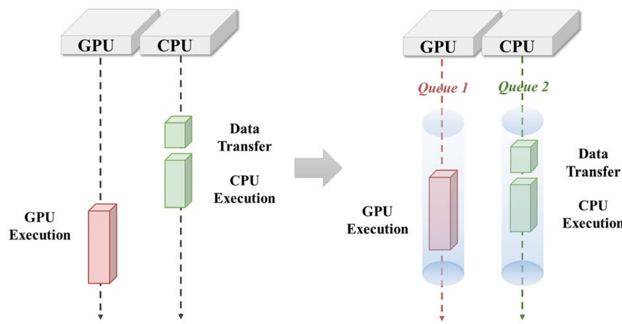


Fig. 1 Concurrency control with task queue in oneAPI

2.2 Concurrency control of oneAPI

As we discussed in Sect. 1, complex concurrency control has caused many troubles for developers and has raised the barrier of heterogeneous programming. oneAPI DPC++ program uses a *queue* mechanism following SYCL standard, where a single queue can be assigned to a single device and will submit its task code to the runtime driver of the corresponding device. Tasks in the same queue are executed sequentially, following the order in which they are added to the queue. Meanwhile, if there are no dependencies or hardware conflicts among tasks in different queues, those tasks can be executed in parallel. Therefore, to achieve the concurrence execution of CPU and GPU, we only need to allocate tasks in 2 separate queues—one for the GPU tasks and another for the CPU tasks.

Figure 1 depicts this scheme with an example. In this example, the execution model coordinates the execution of task kernels and data management between CPU and GPU via queues. Instead of processing data serially, we assign different tasks into two separate queues, which makes the CPU and GPU execute tasks in parallel when data is prepared.

2.3 Out-of-memory graph processing on GPUs

Graph partitioning is a straightforward solution for out-of-memory graph processing. A whole graph is divided into several subgraphs that can fit in GPU memory. The system processes and transfers on-demand subgraphs required by GPU in turn. Many prior works use this scheme, such as GraphReduce (Sengupta et al. 2015) and Graphie (Han et al. 2017). However, the sparse and irregular access pattern of graph traversal leads to lots of redundant data transfer.

Unified Virtual Memory (UVM) was introduced by NVIDIA in Pascal architecture (NVIDIA 2006). With UVM, developers can use a single, virtual address space to manage memory on both CPU and GPU. UVM enables applications to implicitly move data between CPU and GPU memory with page migration managed by the GPU driver.

When processing out-of-memory graphs with UVM, the page scheme could be considered a partitioning-based scheme whereby each partition is a page. However, handling frequent page faults leads to bad performance (Kim et al. 2020) and we find that the graph data is always evicted before reuse due to long reuse distance, making the situation worse.

SubWay (Sabet et al. 2020) is another solution proposed to minimize data transfer between CPU and GPU by selecting and reorganizing on-demand data to be transferred. It is considered the best graph-partitioning-based scheme so far. In *SubWay*, the vertices are maintained in both main memory and GPU memory, and the edges are only kept in main memory. Before each iteration, the required data will be selected and reorganized. By accurately selecting required data for the current iteration during pre-processing, the amount of data transfer between CPU and GPU is greatly reduced, and redundant data transfer is eliminated.

Ascetic (Tang 2021) is the state-of-the-art out-of-GPU-memory graph processing framework. It partitions the GPU memory into two regions, *Static Region* and *On-demand Region*. *Static Region* stores some reusable data and *On-demand Region* requires other graph data on-demand. The computing of *Static Region* and the data transfer of *On-demand Region* is overlapped by concurrent execution of CPU and GPU.

However, all the above works only work on NVIDIA GPUs. And they do not support managing multiple devices at the same time.

2.4 Related works

Out-of-memory graph processing on GPUs has been a highly active research area in recent years due to the growing demand for efficient processing of large-scale graphs. Various frameworks and systems have been developed to tackle the challenges of processing graphs that do not fit into a single GPU memory.

One of the earliest and most well-known frameworks for out-of-memory graph processing on GPUs is Gunrock (Wang 2016), which uses graph partitioning to address the memory constraint. Similarly, approaches such as CuSha (Khorasani et al. 2014), MapGraph (Malewicz 2011), and CuGraph (Jiang et al. 2018) have been developed. However, the limited GPU memory continues to pose a challenge. There are three primary challenges in this area. First, graph partitioning can incur significant overhead due to CPU-GPU synchronization. Second, graph partitioning-based schemes involve numerous redundant data transfer. Third, Unified Virtual Memory (UVM) does not perform well due to data thrashing and page fault latency.

To overcome these limitations, recent works have been proposed. Pegasus (Dong 2021) is a distributed-memory GPU cluster-based graph processing framework for large-scale graphs. SubWay (Sabet et al. 2020) is a recently proposed fine-grained memory management

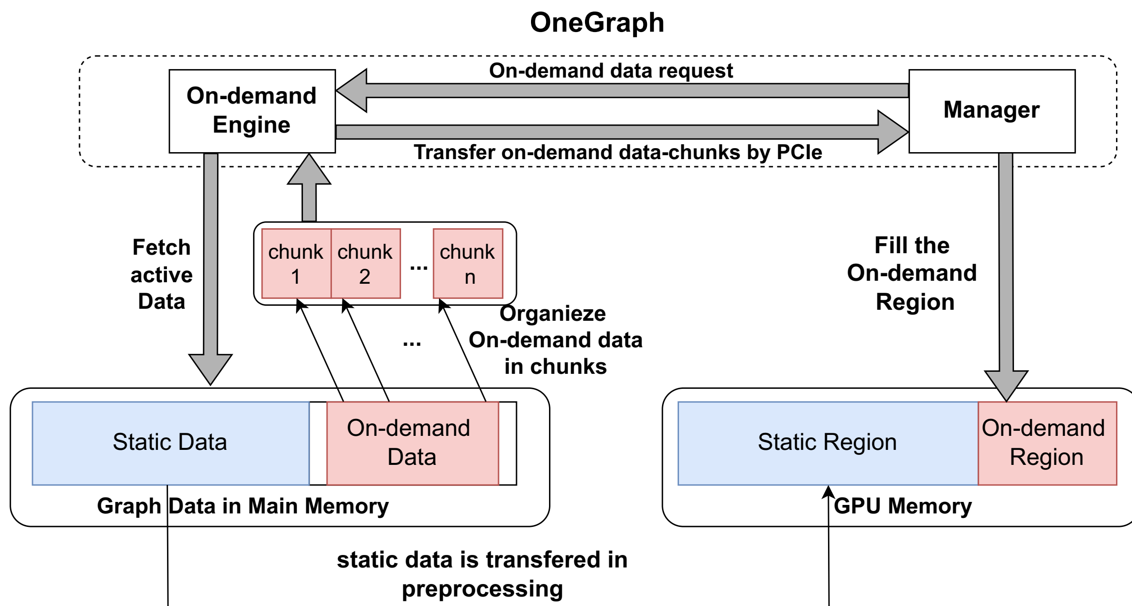


Fig. 2 System overview of *OneGraph*

graph processing system, which reduces redundant transfer by accurately organizing subgraphs and accelerates the organization process with GPUs. Ascetic (Tang 2021) is the state-of-the-art out-of-GPU-memory graph computing framework. It utilizes part of GPU memory as a data cache and handles cache misses in the rest of GPU memory, improving data efficiency with data reuse.

3 System design and implementation

Based on oneAPI, we propose a general cross-architecture large-scale graph processing system *OneGraph*. We follow the scheme of *Ascetic* Tang (2021), which accelerates large-scale graph computing on GPUs across three aspects. First, fully utilizing the GPU memory; Second, attempting to exploit the data reusability in large-scale graph computing to improve data efficiency, reduce data transfer, and eliminate redundant transfer; Third, overlapping CPU and GPU tasks as much as possible to obtain higher parallelism and minimize the busy waiting time.

The memory management approach partitions the GPU memory into two regions, namely the *Static Region* and the *On-demand Region*. The *Static Region* is responsible for caching reusable data, while the *On-demand Region* loads other required data on demand. In this section, we will describe the system design and its implementation with oneAPI in detail.

3.1 System overview

As shown in Fig. 2, the GPU memory is divided into *Static Region* and *On-demand Region* following the design strategy of *Ascetic* Tang (2021). The *Static Region* stores part of data that can be reused across iterations, called *Static Data*. While the *On-demand Region* is in charge of storing data required in the current iteration but not present in the *Static Region*, called *On-demand Data*. In the whole workflow, two controllers are set up. One is the *GPU Manager* which sends requests to the CPU for data to be stored in *On-demand Region*, and the other is the *On-demand Engine* on the CPU side which sorts out the requested data from the raw data in a fine-grained manner and transfers it to the *On-demand Region* within the GPU memory.

To locate the *On-demand data*, we set two bitmaps on the GPU side, namely *Active Bitmap* and *Static Bitmap*. The *Active Bitmap* marks the data required in the current iteration, while the *Static Bitmap* keeps track of the data already prefetched into the *Static Region* of GPU. In each iteration, the data to be accessed is marked as active and set to 1 in the *Active Bitmap*. Before the computing task begins, *GPU Manager* processes the two bitmaps by `AND` operation to locate *Static Data* which can be processed immediately. Simultaneously, *GPU Manager* processes the two bitmaps by `XOR` operation to identify the *On-demand Data* which is left in the main memory. Data in this part is organized in a similar way in SubWay Sabet et al.

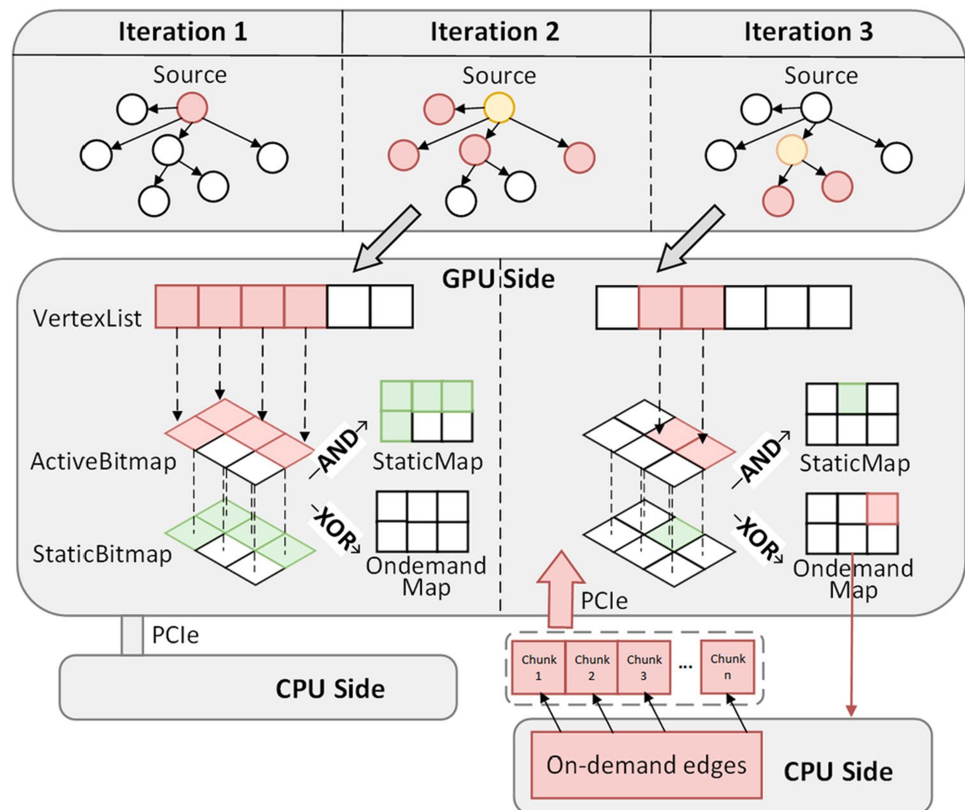


Fig. 3 A BFS workflow of OneGraph. Pink vertexes are active in the current iteration. If its adjacent edges are stored in Static Region, the corresponding index on StaticBitmap will be marked as 1 (green items)

(2020), GPU has to wait for the CPU to process the required data and transfer it to the *On-demand Region*. Furthermore, as the *On-demand data* cannot be stored into the *On-demand Region* all at once, the CPU divides the data into several chunks, resulting in multiple data transmissions.

In this explanation, we'll take the example of BFS (Breadth-First Search) to illustrate the workflow of OneGraph. This is depicted in Fig. 3. The first step involves the GPU manager partitioning the GPU memory based on the size of the loaded graph data, details about how to partition is illustrated in Sect. 3.3. The Static Data is then loaded into the Static Region, and the *Static BitMap* is initialized accurately. Once the iterations begin, the index of the source vertex is marked as 1 on the *Active Bitmap*. Then, parallel `AND` and `XOR` between *Static BitMmap* and *Active Bitmap* and generate *StaticMap* and *On-demand Map* separately. All available adjacent vertexes of the source vertex in *Static Region* are marked as 1 on *StaticMap*, otherwise marked as 1 on *On-demand Map*. GPU begins processing those available *Static Data*, simultaneously sending requests for *On-demand data* to *On-demand Engine*. The CPU manages the organization of this data. After the computing of *Static Data* is completed, *On-demand Engine* fills the *On-demand Region* in GPU memory and GPU begins processing *On-demand Data*. During the processing of graph data, all of the visited vertexes will be marked to 0 and their adjacent vertexes will be marked to 1 on the *Active BitMap*. This prepares the *Active BitMap* for the next iteration. The above processing is repeated until no more vertexes are marked as 1 on the *Active BitMap*.

3.2 Concurrent execution of CPU and GPU

In the given process, specific steps in Fig. 2 can be performed concurrently, particularly between data transmission and computing. For instance, once the GPU manager has finished executing the *AND* operation of two bitmaps, it can immediately begin computing the data in the *Static Region*. The processing of this portion will overlap the *On-demand* Engine's collection and transmission of data in the main memory. This approach reduces the busy waiting time between CPU and GPU, overlapping the latency caused by data transmission.

The programming model of oneAPI makes it simpler to achieve such parallelism with the device queue scheme mentioned in Sect. 2. Therefore, to achieve the overlapping of computing and data transmission, we only need to put these tasks in 2 separate queues - one for the computing task in the *Static Region* and the other for the data transmission in the *On-demand Region*. This approach enables the system to hide data transmission latency and improve overall system performance.

Figure 4 depicts the overlapping effect in each iteration. In *OneGraph*, all the tasks on the CPU side are put into

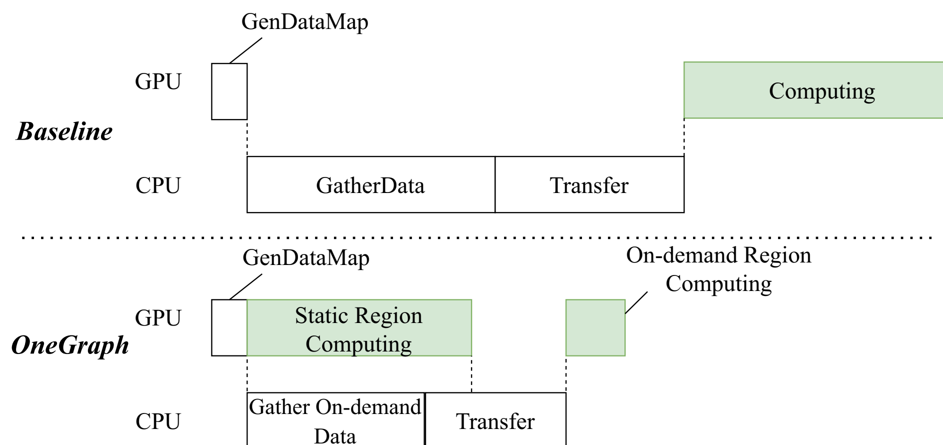


Fig. 4 Computing and data transfer overlapping in OneGraph

one task queue, which is assigned to the CPU. Tasks on the GPU side are put into another queue that is assigned to the GPU device. Due to there being no data dependence and hardware conflict between *Static Region Computing* and CPU tasks, the *Static Region Computing* and on-demand data location and transfer will be processed in parallel. In order to ensure correctness, a system-wide synchronization must be performed prior to *On-demand Region Computing*. Listing 3 shows the code of this part and Listing 4 shows its CUDA version. Concurrency execution in the DPC++ program has been simplified, allowing programmers to control only when the `wait()` interface for a `queue` is used. In Listing 3, no `wait()` is demonstrated for the *StaticRegion Computing* and *On-demand data location* before their completion, enabling these tasks to execute concurrently and the on-demand data locating and transfer latency will be overlapped with *StaticRegion Computing*. Conversely, executing the same scheme in CUDA necessitates specific APIs, such as `cudaMemcpyAsync`, and managing different `cudastreams` while launching kernels.

Listing 3 overlapping code in OneGraph

```
auto dev_1 = gpu_selector();
auto dev_2 = cpu_selector();
sycl::queue gpu(dev_1), cpu(dev_2);
 ...
gpu.submit([&](handler& h){
        //StaticRegion computing code on GPU
});
 ... //on−demand data location code on CPU
cpu.memcpy(dst,src,size).wait();
gpu.wait();//system−wide synchronization
 ... //On−demand Region computing
```

Listing 4 overlapping code in cuda

```
cudaChooseDevice(&devicedChoosed, &
     devicePropDefined);
cudaStream_t streamStatic, streamDynamic;
dim3 grid = get_blocks_per_grid();
dim3 block = get_threads_per_blocks();
 ...
//the statement of kernel is written in another
     file
SaticComputingKernel<<<grid,block,0,
     streamStatic>>>(args...);
 ... //on−demand data location code on CPU
cudaMemcpyAsync(dst,src,size,
     cudaMemcpyDeviceToHost, streamDynamic
     );
cudaDeviceSynchronize();
 ... //On−demand Region computing
```

3.3 GPU memory partition

In *OneGraph*, the data in the *Static Region* is reusable across iterations, and the computing of these data can overlap the latency of data transfer. We attempt to maximize *Static Region*'s size. However, a small *On-demand Region* can lead to dividing the *On-demand Data* into more and smaller chunks, resulting in more frequent data transfer between the CPU and GPU. This situation is similar to the graph partitioning approach discussed in Sect. 2.1, leading to data thrashing and performance degradation. Moreover, transmitting too little data during each transmission will waste the PCIe bandwidth. Hence, it is critical to set a proper ratio between the two regions for better performance.

In the *OneGraph*, we just follow the memory partitioning approach in *Ascetic* Tang (2021), determining the ratio between the *Static Region* and the *On-demand Region* using empirical values. Assuming that the proportion of required edge data in a single iteration is K, set M as the size of GPU memory, and M_{Static} is the size of *Static Region*. If the size of the dataset is D, the average data size that needs to be loaded into the *On-demand Region* for each iteration is

Table 1 GPU platform configuration

	NVIDIA GPU	Intel GPU
Model	A100 PCIe	Data Center GPU Flex 170
Cores	6912	512
Memory	80GB HBM2	16GB GDDR6
Driver	Driver 515.65.01 and CUDA 11.6.2	intel-i915

$(D - M_{Static}) \times K$. Set R is the proportion of *Static Region*, that is $R = M_{Static}/M$, in order to maximize the size of the *Static Region* while meeting the requirements, we should ensure that:

$$R = \left(1 - K \times \frac{D}{M}\right)/(1 - K) \tag{1}$$

Some studies (Sahu et al. 2017; Tang 2021) have demonstrated that the average proportion of edge data required per iteration is about 10%. Hence we choose 10% as the default value of K in experiments.

4 Experiment and evaluation

4.1 Experiment settings

Our local experiment platform has 96 Intel(R) Xeon(R) Gold 5318Y CPUs and 256GB DRAM memory, running on Linux Ubuntu 22.04.1. The configuration of the NVIDIA GPU and Intel GPU we used are shown in Table 1.

As shown in Table 2, we use four real-world large-scale graphs and two synthesized graphs in experiments. The GS and FK are directed and others are undirected. Their size shown in Table 2 is their size in CSR format without edge weight, and when the edge weights are necessary, it will nearly double the CSR file size. For instance, the size of FK with edge weight in CSR is 20 G. We use RMAT (Chakrabarti et al. 2004), a widely used graph generator, to generate the synthesized graphs. We limit the available GPU memory to 12 G for our applications in experiments to simulate the out-of-memory situation. It is worth mentioning that in this scenario, all the data sets, regardless of whether they have edge weight or not, are out-of-memory. This is because the frameworks used in experiments will introduce some additional memory overhead. Four classical graph processing algorithms are used to evaluate our scheme, BFS, CC, SSSP, and PR. We use the empirical value to determine the size of *Static Region* according to *Ascetic* (Tang 2021).

For comparison, we implement a UVM-based scheme in our framework with Intel oneAPI, using the Unified Shared Memory API *malloc_shared* in Intel oneAPI to allocate the *EdgeList* and keep other data in GPU memory. We also reproduce a prototype of *SubWay* with oneAPI for comparison, which gathers requested data in a fine-grained manner and sends them to GPU for processing. *SubWay* and *OneGraph* both use muti-thread to accelerate the data organization on the CPU side. Each application is run 10 times and we take the arithmetic mean in our evaluation.

4.2 Performance analysis and evaluation

Table 3 shows three memory management approaches' performance on the datasets in Table 2. We evaluate them with their speedup over the common CPU serial approach. The results show that the UVM-based scheme could reach an average 11.65x speedup while the *SubWay* only has 9.23x. *OneGrpah* can achieve a 32.62x speedup on average and 127.53x in the best case.

OneGrpah achieves the best performance among them. Our analysis of the reasons for the poor performance of *SubWay* revealed that the bottleneck lies primarily in the GPU busy waiting for the CPU to organize and transfer data. Although we have accelerated the data organization process with multi-thread and GPU, the final results show that the GPU is still idle for an average of 50% of the GPU time in *SubWay*.

Table 4 demonstrates the average data transfer amount of *OneGrpah* is only 12.74% of that of USM. Compared to USM, *OneGrpah* reduces data transfer by almost 90%, and compared to *SubWay*, *OneGrpah* reduces data transfer by almost four times. This illustrates that *OneGrpah* greatly improves data efficiency while fully utilizing GPU memory resources. In some experiments, *SubWay* may transfer more data than USM, which is due to the communication costs when using GPU to accelerate subgraph generation,

Table 2 Datasets used in experiments

Abbr	Name	Vertices	Edges	Size
GS	Gsh-host-2015(d) (Low 2010)	65.47M	1.68B	14 G
FK	Friendster-konect(d) (Ganguly 2020)	65.18M	2.41B	11 G
UK	Uk-2007-04(u) (Low 2010)	101.92M	3.53B	15 G
FS	Friendster-snap(u) (Leskovec and Krevl 2014)	124.83M	3.61B	15 G
RMAT1	RMAT-rand(u)	5.25M	1.96B	16 G
RMAT2	RMAT-rand(u)	106.67M	3.72B	15 G

Table 3 Performance results

		Serial	USM	SubWay	OneGraph
SSSP	FS	105.91s	1.04x	3.75x	**10.11x**
	GS	340.97s	2.12x	3.27x	**10.72x**
	FK	151.46s	2.08x	3.43x	**20.56x**
	UK	236.69s	0.51x	1.06x	**2.34x**
	RMAT1	117.63s	1.96x	3.97x	**14.15x**
	RMAT2	507.40s	1.91x	7.78x	**29.33x**
BFS	FS	48.78s	4.36x	3.26x	**15.36x**
	GS	66.94s	35.72x	26.90x	**110.83x**
	FK	11.31s	21.07x	16.27x	**110.90x**
	UK	133.14s	6.27x	5.71x	**28.54x**
	RMAT1	8.98s	10.57x	11.21x	**39.44x**
	RMAT2	284.32s	54.28x	48.47x	**127.53x**
CC	FS	92.65s	6.48x	2.88x	**17.01x**
	GS	41.82s	1.93x	1.05x	**12.53x**
	FK	126.66s	8.35x	3.52x	**28.68x**
	UK	170.19s	2.41x	1.53x	**4.94x**
	RMAT1	63.58s	2.68x	1.84x	**9.97x**
	RMAT2	192.01s	7.48x	4.28x	**11.64x**
PR	FS	604.25s	17.16x	13.41x	**28.38x**
	GS	568.10s	2.01x	**6.94x**	3.31x
	FK	560.50s	31.16x	34.89x	**71.57x**
	UK	879.70s	0.90x	5.45x	**7.36x**
	RMAT1	455.00s	15.84x	10.90x	**38.29x**
	RMAT2	906.50s	41.41x	1.89x	**29.32x**
Average			11.65x	9.23x	**32.62x**

The values of USM, SubWay and OneGraph are represented by the speedup of Serial. The best among the three methods is in bold

Table 4 Data transfer results

		USM(GB)	SubWay (%)	OneGraph (%)
SSSP	FS	33.03	265.49	**33.64**
	GS	418.93	126.26	**3.29**
	FK	404.84	15.32	**2.82**
	UK	3545.39	29.18	**0.43**
	RMAT1	2781.65	1.47	**0.26**
	RMAT2	421.05	21.90	**5.82**
BFS	FS	22.66	60.08	**12.88**
	GS	82.31	21.00	**3.02**
	FK	106.81	12.01	**2.01**
	UK	199.56	18.60	**4.56**
	RMAT1	147.69	6.05	**2.24**
	RMAT2	108.06	15.08	**8.85**
CC	FS	14.12	158.25	**19.63**
	GS	123.59	46.51	**13.53**
	FK	82.05	44.49	**23.62**
	UK	334.10	28.85	**22.21**
	RMAT1	157.22	24.99	**15.29**
	RMAT2	228.67	17.94	**14.63**
PR	FS	348.41	38.50	**30.25**
	GS	4697.27	8.23	**3.23**
	FK	688.21	17.19	**7.37**
	UK	13951.46	3.35	**2.53**
	RMAT1	170.59	67.09	**20.22**
	RMAT2	255.51	1247.07	**39.34**
Average			90.04	**12.74**

The values of SubWay and OneGraph are represented by the percentage of USM. The best among the three methods is in bold

which incurs a significant overhead when the communication between CPU and GPU is too frequent.

4.3 Breakdown of the optimization

For a deep understanding of the performance optimization of *OneGraph*, we use "static savings" to represent the performance benefits of caching data in *Static Region*, and "overlapping savings" to represent the performance benefits of the overlapping strategy. We use SubWay's performance as a baseline, as shown in Fig. 5. The results show that, compared to *SubWay*, the *Static Region* can provide an average performance improvement of 30%, while the overlapping strategy can provide an average performance improvement of 32%. It is worth noting that, for the BFS, there is actually no fine-grained data reuse since visited nodes will not be visited again. However, the *Static Region* prefetching policy can still provide an average performance improvement of 20% for the BFS. This is because, after some data is prefetched into the GPU's memory, the GPU can directly access it without waiting for the CPU to organize the data.

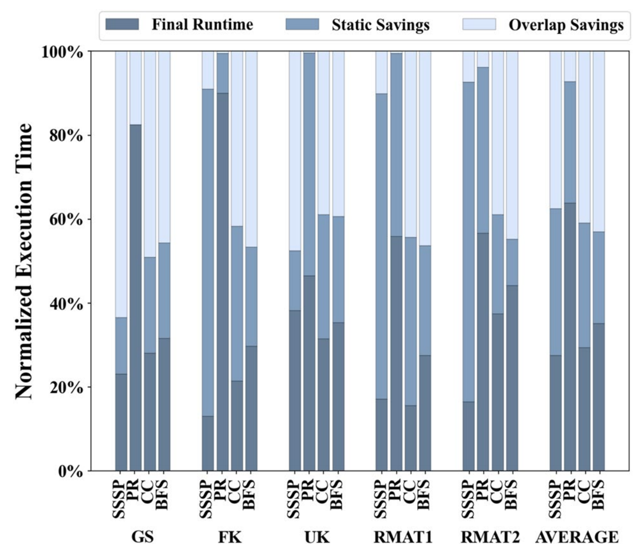


Fig. 5 Breakdown of the optimization benefits

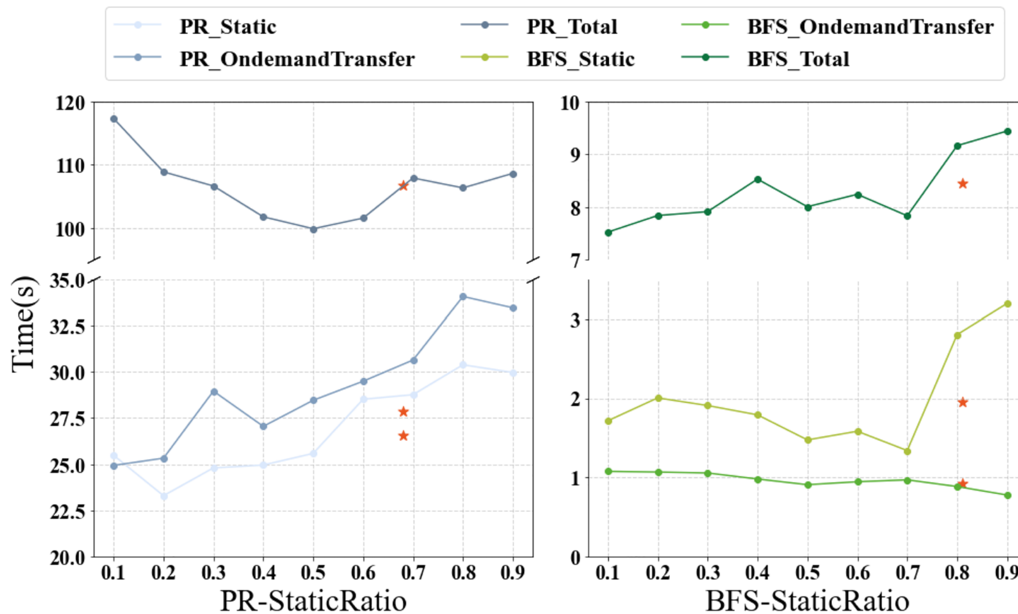


Fig. 6 Impact of static region ratio on performance(PR/BFS) using UK dataset. The red star is the result of chosen ratio based on empirical value

The bitmaps in *OneGraph* are not free. We have evaluated the overhead during the look-up of *StaticMap* and *OndeamandMap*. The results show that the look-up of bitmaps takes 31% of GPU time in *OneGraph*. According to our measurement, *SubWay* spends 50% of GPU time to locate data on the CPU side and USM spends 61% of GPU time to handle page faults, which is consistent with other research (Kim et al. 2020). In contrast, *OneGraph* significantly minimizes the overhead associated with locating on-demand data. Bitmaps also reduce available GPU memory for storing graph data. According to our scheme, bitmaps and relative auxiliary arrays will occupy 2.2 GB for GSH and FK, and 4.1 GB for FK and UK. Despite the limited GPU memory for processing all edges, sacrificing some memory to achieve faster data locating and processing is a justifiable trade-off. However, when the dataset is so large that bitmaps exceed GPU memory, *OneGraph* becomes infeasible.

We also test the impact of a different ratio between *Static Region* and *On-demand Region* on the UK dataset with PR and BFS. As shown in Fig. 6, with a larger ratio of *Static Region* increases, *OneGraph* spends more time in *Static Data* processing, leading to fewer *On-demand Data* transmissions and more overlapping between transfer and computing. The results demonstrate that the chosen ratio based on our empirical value can achieve relatively good performance, although not always optimal performance. This impact varies with the different topologies of different graph data sets.

We also use RMAT (Chakrabarti et al. 2004) to generate some larger datasets(e.g.20G-50 G) and compare the performance of USM, *SubWay*, and *OneGraph*. This time we limited available GPU memory to the application to 15 G-20 G. Results show that *OneGrpah* achieves an average of 1.64x speedup over USM and 2.75x speedup over *SubWay* on these larger datasets. Moreover, with a larger dataset and limited PCIe bandwidth, *SubWay* and *OneGraph* have to spend more time in data transfer, however, *OneGraph* always transfers fewer data because the date reuse in *Static Region*.

In general, our experiments prove that the empirical value of the ratio between two regions is reasonable. And whether different ratios or a larger data set, *OneGraph* always shows the best performance.

4.4 Graceful balance between portability and performance

oneAPI is a cross-architecture programming model and Intel has also proposed a series of tools to support convenience cross-architecture portability and migration. For NVIDIA GPUs and AMD GPUs, besides Intel oneAPI Toolkit, a plugin for compiler (CodePlay 2023a, b) is necessary. The plugin can be used along with the existing oneAPI Toolkits that includes the oneAPI DPC++/C++ Compiler to build your SYCL code and run it on compatible NVIDIA or AMD GPUs. Moreover, the DPC++ Compatibility Tool(dpct) in oneAPI toolkit could automatically migrate a CUDA project to a DPC++/SYCL project (Intel 2023b).

We have tested the performance of *OneGrpah* on different hardware platforms. Without any code modifications, *OneGrpah* was successfully compiled and able to run on

Table 5 The performance of OneGraph and SubWay on Intel Flex170 GPU

	SubWay	OneGraph	SpeedUp
BFS	26.53s	7.98s	3.32x
PR	83.89s	64.43s	1.30x
CC	48.15s	17.93s	2.69x
SSSP	293.4s	56.59s	5.18x
AVERAGE	100.97s	36.73s	2.75x

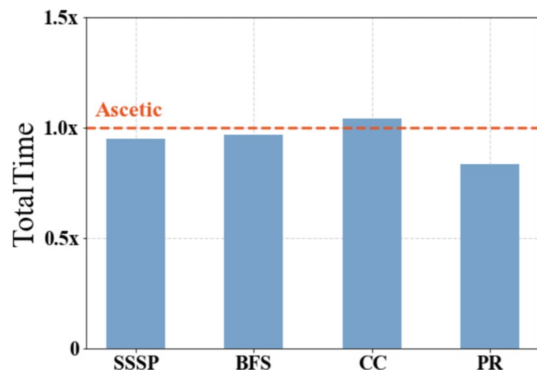


Fig. 7 Performance comparison with CUDA program

NVIDIA A100 GPU, Intel Datacenter Flex170 GPU, and AMD Radeon integrated GPU. We conducted experiments using datasets in Table 2 on the Intel GPU in the Intel data center to evaluate *SubWay* and *OneGrpah*. The results, summarized in Table 5, indicate that *OneGraph* achieves an average speedup of 2.75x over *SubWay*. Our experiments also reveal that PR performs better on the Intel Flex170 GPU than on the NVIDIA A100 GPU. Due to the AMD GPU we have is not a data-center-level product, we did not conduct any further evaluations on it.

When considering the trade-offs between convenient portability and performance, it is widely acknowledged that achieving both can be challenging. oneAPI, as a general cross-architectural programming model, usually being considered to have a significant performance loss compared to a dedicated programming model like CUDA. We have reproduced *Ascetic* according to its open-source release and compared the performance of *OneGraph* with it. Figure 7 shows the performance comparison between *OneGraph* and *Ascetic*. As shown in Fig. 7, the results demonstrate that the performance loss of *OneGraph* is less than 1% for CC. Furthermore, for BFS, SSSP, and PR, *OneGraph* outperforms *Ascetic*. *This finding suggests that OneGraph strikes a graceful balance, providing both good portability and excellent performance for large-scale graph computing on CPU-GPU heterogeneous platforms*.

5 Conclusion

Efficiently utilizing heterogeneous platforms to accelerate large-scale graph computing applications is a significant challenge. The steep learning curve, complex concurrency control schemes, and bad portability of dedicated programming models have been troublesome for heterogeneous developers. In this paper, we utilize the opportunity provided by Intel oneAPI to design a cross-architecture framework for large-scale graph computing on heterogeneous platforms with GPU and CPU. It follows the design of the state-of-the-art large-scale graph computing framework. We implement a prototype of the framework *OneGraph* using Intel oneAPI and conduct rigorous performance tests on four classical graph algorithms. The results show that *OneGraph* outperforms the graph-partitioning scheme and UVM scheme in large-scale graph computing. Moreover, it can be easily ported to different hardware platforms without code modification. The performance loss is only less than 1% in large-scale graph computing and the code size is also significantly reduced compared to other dedicated programming models.

Acknowledgements This work was supported in part by the Key Research and Development Program of Guangdong, China (2021B0101310002), Natural Science Foundation of China (62172239), and Intel Corporation.

Data availability The data that support the findings of this study are available on request from the corresponding author upon reasonable request.

Code availability The source code of *OneGraph* is available at https://github.com/NKU-EmbeddedSystem/OneGraph

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

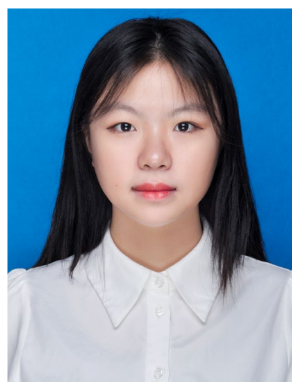
AMD: ROCm (2022). https://www.amd.com/zh-hans/graphics/servers-solutions-rocm-ml. Accessed: April 7, 2023
Boldi, P., Codenotti, B., Santini, M., Vigna, S.: UbiCrawler: a scalable fully distributed web crawler. Softw. Pract. Exp. **34**, 711–726 (2004)
Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining, pp. 442–446. SIAM (2004)
CodePlay: OneAPI for AMD GPUS. https://developer.codeplay.com/products/oneapi/amd/home/ (2023a). Accessed 14 Apr 2023
CodePlay: OneAPI for NVIDIA GPUS. https://developer.codeplay.com/products/oneapi/nvidia/home/ (2023b). Accessed 14 Apr 2023
Dong, Y., et al.: PEGASUS: pre-training graph neural networks by contrastive decoding of graph random walks, pp. 6996–7008 (2021)
Ganguly, D., Zhang, Z., Yang, J., Melhem, R.: Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription, pp. 451–461 (2020)

Han, W., Mawhirter, D., Wu, B., Buland, M.: Graphie: large-scale asynchronous graph traversals on just a GPU, pp. 233–245 (2017)
Harris, M. Unified memory for CUDA beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (2021). Accessed 31 Dec 2020
Intel: Intel oneAPI. https://www.intel.com/content/www/us/en/software/oneapi.html (2023a). Accessed Mar 2023
Intel: Migrate CUDA applications to oneAPI cross-architecture programming model based on SYCL. https://www.intel.com/content/www/us/en/developer/articles/technical/migrate-cuda-applications-to-oneapi-based-on-sycl.html (2023b). Accessed 14 Apr 2023
Jiang, C., Chou, J., Zhou, T.: cuGraph: a GPU-accelerated graph analytics library, pp. 1–7. IEEE (2018)
Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: CuSha: vertex-centric graph processing on GPUS, pp. 239–252 (2014)
Khronos: OpenCL. https://www.khronos.org/opencl/ (2011). Accessed 7 Apr 2023
Khronos: SYCL 2020 provisional specification. Tech. Rep., Khronos Group (2020)
Kim, W.: Prediction of essential proteins using topological properties in GO-pruned PPI network based on machine learning methods. Tsinghua Sci. Technol. **17**, 645–658 (2012)
Kim, H., Sim, J., Gera, P., Hadidi, R., Kim, H.: Batch-aware unified memory management in GPUS for irregular workloads. In: ASPLOS’20, pp. 1357–1370. Association for Computing Machinery, NY, USA, New York (2020)
Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)
Low, Y., et al.: GraphLab: a new framework for parallel machine learning. In: UAI’10, pp. 340–349. AUAI Press, Arlington, Virginia, USA (2010)
Malewicz, G., et al.: Pregel: a system for large-scale graph processing, pp, 135–146. ACM (2011)
NVIDIA: CUDA toolkit. https://developer.nvidia.com/cuda-toolkit (2022). Accessed 7 Apr 2023
NVIDIA: NVIDIA Tesla P100-the most advanced datacenter accelerator ever built featuring pascal GP100. https://www.nvidia.cn/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-teslap100-techoverview.pdf (2006). Accessed 26 Nov 2022
Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI’15, pp. 4292–4293 (2015)
Sabet, A.H.N., Zhao, Z., Gupta, R.: Subway: minimizing data transfer during out-of-GPU-memory graph processing. In: EuroSys’20. Association for Computing Machinery, NY, USA, New York (2020)
Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing. Proc. VLDB Endow. **11**, 420–431 (2017)
Sengupta, D., Song, S.L., Agarwal, K., Schwan, K.: GraphReduce: processing large-scale graphs on accelerator-based systems. In: SC’15, Association for Computing Machinery, NY, USA, New York (2015)
Tang, R., et al.: Ascetic: enhancing cross-iterations data efficiency in out-of-memory graph processing on GPUS. In: ICPP 2021. Association for Computing Machinery, NY, USA, New York (2021)
Wang, Y., et al.: Gunrock: a high-performance graph processing library on the GPU, pp. 1–12 (2016)

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Shiyang Li received the bachelor’s degree from Nankai University, in 2022. He is currently working toward the graduate degree with Nankai University. His research interests include parallel processing and operating system.

Jingyu Zhu is currently working toward the graduate degree with Nankai University.



Jiaxun Han is currently working toward the graduate degree with Nankai University.

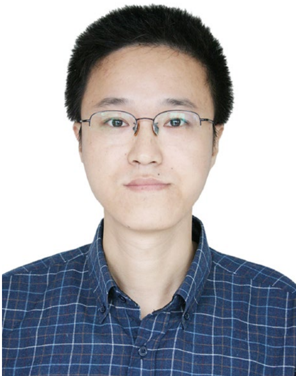
Yuting Peng is currently working toward the graduate degree with Nankai University.

Zhuoran Wang is currently working toward the graduate degree with Nankai University.

Jin Zhang received the PhD degree from Nankai University. He is a professor with Nankai University. His main research interests include mobile cloud computing.

Xiaoli Gong was born in 1983. He received the bachelor's and doctor degrees from Nankai University, Tianjin, China in 2005 and 2011 respectively. He is currently an associate professor and the vice chair with the Institute of Systems and Networks at Nankai University. His research interests include system virtualization, parallel computing, embedded system optimization, and trustable computing.

Xuqiang Wang was born in 1989. He is an associate senior engineer in the Information and Communication Company of State Grid Tianjin Electric Power Company. His research interests include application technology of AI in power grid and big data mining.



Gang Wang was born in 1974. He received the PhD degree from Nankai University. He is currently a professor. His research interests include massive data storage, search engine, parallel computing, and bioinformatics.