2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3503-8322-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/HIPC58850.2023.00017

Hybrid CUDA Unified Memory Management in Fully Homomorphic Encryption Workloads

Jake Choi
Dept. of Computer Science and Engineering
Seoul National University
Seoul, South Korea
kidcoder@snu.ac.kr

Jaejin Lee,
Sunchul Jung
CryptoLab
Seoul, South Korea
ljj@cryptolab.co.kr
zotanika@cryptolab.co.kr

Heonyoung Yeom
Dept. of Computer Science and Engineering
Seoul National University
Seoul, South Korea
yeom@snu.ac.kr

***Abstract*—Fully homomorphic encryption (FHE) can utilize GPUs to accelerate arbitrary operations directly on encrypted data without decryption. Functions like bootstrapping which refresh noise accumulated on ciphertexts due to repeated operations require great amounts of GPU memory. Such functions cause out-of-memory (OOM) issues in retail GPUs with less than 8GB of VRAM, causing bootstrap to fail. Utilizing CUDA Unified Memory can alleviate OOM problems by allowing automatic page swapping from host to device memory. However, it usually comes with significant performance overheads. We devise a hybrid memory run-time that distinguishes between objects that are allocated asynchronously, or with managed memory. Our initial implementation was to statically determine in code the type of allocation method each GPU object would use. For a more general solution, we created a run-time scheduler which profiles and automatically determines the method of allocation each GPU object should use, without requiring static changes in library code. We then expanded upon this by creating a dynamic scheduler which forecasts the lifetime of future GPU objects without profiling. Our static method increases bootstrapping performance by ∼31% for large parameter sizes when memory is oversubscribed. Our profiling scheduler improves ResNet performance compared to manual swapping by ∼22%. Finally, our pure dynamic scheduler gives performance that is similar to our static solution, and up to 50% better performance in bootstrapping than the base unified case.**

***Index Terms*—homomorphic encryption, gpu, unified memory, hybrid, memory allocation**

I. Introduction

Fully Homomorphic Encryption (FHE) is a cryptographic scheme that enables computations to be performed directly on encrypted data, without the need for decryption. FHE holds immense potential for preserving privacy and security in various domains, such as cloud computing, machine learning, and data analysis. However, the practical deployment of FHE is impeded by its high computational and memory requirements, which can limit its ease of usage.

Graphics Processing Units (GPUs) have emerged as powerful accelerators for a wide range of computationally intensive tasks, including cryptographic operations. Leveraging the parallel processing capabilities of GPUs can potentially overcome the performance bottlenecks associated with FHE computations, particularly with bootstrapping [1]. However, GPU memory is typically limited in comparison to host DRAM memory, and cannot be expanded easily. This is particularly true for cost-efficient retail GPUs used in most home or workspace environments. Server-based GPUs are quite expensive, and cannot be readily accessed in many environments. Therefore, efficient memory management plays a crucial role in optimizing the execution of FHE algorithms on retail GPUs.

In this paper, we investigate the use of a hybrid memory allocation strategy, combining `cudaMallocAsync` and `cudaMallocManaged`, on the HEaaN library [2]. The `cudaMallocAsync` function allocates device memory asynchronously, which results in performance improvements compared to traditional GPU memory allocation. On the other hand, `cudaMallocManaged` provides a unified memory abstraction that allows for data swapping between the host and the device all managed by the CUDA driver, preventing OOM from occuring. Our research focuses on evaluating the performance characteristics and memory management tradeoffs of utilizing hybrid memory allocation techniques in FHE applications.

By benchmarking various FHE operations using our scheme, we aim to quantify the benefits of the hybrid `cudaMallocAsync` and `cudaMallocManaged` approach in terms of computational speed. Specifically, we assess the performance impacts of increased data movement and synchronization overheads resulting from using managed memory and on improved memory allocation performance by using asynchronous GPU memory allocation.

In summary, this paper aims to explore the advantages of utilizing hybrid memory allocation techniques for FHE on GPUs. We will present experimental results highlighting the performance gains achieved through the hybrid `cudaMallocAsync` and `cudaMallocManaged` approach, without resulting in traditional OOM problems. As far as we know, we have not seen any published work that adopted a method using GPU hybrid memory allocation combining both stream-ordered memory allocation and managed memory.

This work is supported by CryptoLab. This work was also supported in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2003618). Prof. Yeom is the corresponding author of this paper.

2640-0316/23/$31.00 ©2023 IEEE
DOI 10.1109/HiPC58850.2023.00017

II. Background

A. CUDA Unified Memory

NVIDIA's CUDA Unified Memory [3] (UM) is a memory management feature introduced in CUDA 6.0 that simplifies memory management for GPU-accelerated applications. It provides a unified memory address space that can be accessed by both the CPU and GPU, enabling seamless automatic data movement between the host (CPU) and the device (GPU). This eliminates the need for explicit memory transfers and also simplifies the programming model, making it easier to develop and optimize GPU-accelerated applications. Developers allocate memory using `cudaMallocManaged`, which allocates memory regions that can be accessed by both the CPU and GPU using a single pointer. This makes it easier for the programmer to share data between devices.

Under the UM model, data is dynamically migrated between the CPU and GPU on-demand. When a CPU or GPU operation accesses data that resides in the other device's memory space, CUDA automatically handles the migration behind the scenes. This migration can also be performed asynchronously using functions like `cudaMemPrefetchAsync`, allowing the CPU and GPU to overlap computation and data transfers, thereby improving overall application performance.

The advantages of CUDA UM include simplified memory management, reduced code complexity, and improved productivity for GPU programming. While CUDA UM offers convenience and ease of use, it is important to consider its implications for performance. Data movement between the CPU and GPU can incur overhead, especially when frequent migrations are required. Careful consideration of data access patterns, memory usage, and synchronization points is crucial to optimize the performance of applications utilizing CUDA UM.

B. CUDA Stream-ordered Memory Allocation

Released in CUDA 11.2 [4], `cudaMallocAsync` is an extension of the `cudaMalloc` function, which is used to allocate memory on the GPU. The main difference is that `cudaMallocAsync` allows for asynchronous stream-ordered memory allocation, which means that the function call does not block the GPU until the memory allocation is complete. The asynchronous nature of `cudaMallocAsync` can potentially improve overall performance and resource utilization by allowing memory allocations to be performed in a stream-ordered manned. In this manner the GPU can continue processing using the allocated memory, without GPU device-wide synchronization that a `cudaFree` operation will invoke. Previously, custom memory allocators were needed to allocate large chunks of memory upfront in order for memory to be reused without lag. This could potentially cause bugs to occur, and increases complexity of the code. However, these new operations provided by CUDA eliminate that need with a programmer-friendly interface to increase productivity.

The `cudaMallocAsync` function takes as parameters the device pointer, the size of memory to be allocated and a CUDA stream. A CUDA stream is a sequence of commands that are executed in order on the GPU. By associating the memory allocation with a specific stream, developers can control the ordering and synchronization of GPU operations.

It's important to note that `cudaMallocAsync` does not guarantee immediate memory allocation. Instead, it initiates the memory allocation process asynchronously based on stream ordering. Internally, a memory pool is created by the CUDA driver. Memory from the pool is allocated from the OS based on a set threshold that the programmer decides. Synchronization mechanisms such as `cudaStreamSynchronize` or event-based synchronization calls automatically return unused memory back to the OS unless the specific threshold is set to maintain memory in the memory pool. Reusing memory by using `cudaMallocAsync` can result in much faster memory allocation performance.

C. Memory Usage in FHE

In FHE schemes, whether it is TFHE [5], CKKS [2], BGV [6], or BFV [7], memory usage can vary based on several factors:

- Security Parameters: The memory requirements can depend on the specific security parameters chosen for the FHE scheme, such as the modulus size, the ciphertext size, and the level of security desired. Larger security parameters generally result in increased memory usage. Table I shows a brief view of the parameter sizes in the HEaaN library [8]. N denotes the number of slots in a ciphertext, Q is a set composed of $q_1, \ldots, q_n$ prime numbers and $log(Q) = log\Pi q_i$, where $i \in [1, n]$. $|\cdot|$ denotes the number of elements in a set. These parameters all have a security level over 128-bits. The larger the value of N, the more memory the given parameter consumes.

TABLE I
HEaaN Parameter Sizes

Parameter	N	$log_2(Q)$	$\|Q\|$
FVa	2^{17}	2070	40
FVb	2^{17}	2292	40
FVc	2^{17}	2341	40
FGa	2^{16}	1555	28
FGb	2^{16}	1555	30
FTa	2^{15}	777	22
FTb	2^{15}	771	17

- Ciphertext Size: The size of the ciphertexts used in FHE can impact memory usage. Larger ciphertext sizes require more memory to store and manipulate the encrypted data. The choice of ciphertext size depends on the application requirements and the desired level of precision or granularity in computations.
- Encryption Keys: The number of encryption keys used in the FHE scheme can also affect memory usage. Additional keys may be required for bootstrapping or other cryptographic operations. The size of these keys and their

associated metadata can contribute to the overall memory requirements.

- Intermediate Computations: FHE schemes often involve performing multiple homomorphic operations on encrypted data. The memory usage can increase as intermediate computations generate new ciphertexts or accumulate noise. Efficient management of intermediate ciphertexts and noise reduction techniques can help optimize memory usage.
- Bootstrapping: Bootstrapping is a key operation in FHE that refreshes the ciphertexts to reduce accumulated noise while maintaining correctness and security during long computations. Bootstrapping typically requires significant memory resources due to the need to store and manipulate large polynomials representing ciphertexts. Bootstrapping is an operation that is particularly memory-intensive, because of large numbers of keys that are generated.

Each factor is independently related, and thus increasing all factors can greatly increase the memory footprint. In this paper, we primarily focus on the memory usage of the HEaaN library, which is an approximate arithmetic number implementation based on the CKKS scheme.

D. HEaaN Library Operations

As stated before, the HEaaN library is an approximate implementation of the CKKS scheme. We will not go into the mathematics in this paper as other work [9] have already covered the details. Basically, CKKS has three types of data: a message, plaintext and ciphertext. Messages are stored as arrays of complex numbers. Plaintexts are messages converted to NTT [10] polynomials in a process called **encoding** and **decoding** to be readily encrypted. Ciphertexts are **encrypted** plaintexts requiring encryption keys in order to be **decrypted** with the corresponding secret key. **Addition** is relatively straightforward in CKKS, where two ciphertexts are pairwisely added, resulting in an approximately correct result because added errors are insignificant. **Multiplication** between two ciphertexts requires a **rescale** operation to finalize the result, where the ciphertext modulus is reduced to a lesser number, and a number of least significant bits are truncated [11]. This bounds the number of multiplications that can be performed without destroying the message. In order to recover the reduced modulus, **bootstrapping** must be performed. **Rotation** of ciphertexts involves shifting the location of elements in plaintext slots (a space containing an element in the message vector), which allows computation of different extracted elements [12].

Each parameter set determines the security level of the ciphertext, and number of ciphertext slots, and requires different sets of keys of different types, depending on the operations performed on ciphertexts. Therefore the amount of memory usage will largely vary depending on the parameter used. Additionally, faster variants will also require sets of constant values that are required to be transferred to GPU memory in order to perform faster computations. Such constants may also potentially consume more or less GPU memory, depending on the parameter sets used. The experiments in this paper are performed using different parameter sets on the above outlined operations.

III. Implementation

A. Static Hybrid Implementation

The HEaaN library in its default GPU implementation performs all GPU memory allocations using the stream-ordered `cudaMallocAsync`. This causes memory allocations to be limited to the maximum GPU size. We initially implement CUDA UM by replacing all host and GPU memory allocation with `cudaMallocManaged`. This guarantees that OOM won't occur, as long as memory usage is bounded by host DRAM capacity, or even secondary storage capacity if swap memory is used, albeit with performance degradation. The performance degradation that we found from just replacing asynchronous memory allocation with managed memory was significant. After profiling several homomorphic operations using CUDA `nsys` [13], we noticed that there was a difference in both individual kernel latencies, and that the GPU was actually idle a significant portion of the time during operations.

The HEaaN library in its default GPU implementation performs all GPU memory allocations using the stream-ordered `cudaMallocAsync`. This causes memory allocations to be limited to the maximum GPU memory size. We initially implement CUDA UM by replacing all host and GPU memory allocation with `cudaMallocManaged`. This guarantees that OOM won't occur, as long as memory usage is bounded by host DRAM capacity, or even secondary storage capacity if swap memory is used, albeit with performance degradation. The performance degradation that we found from just replacing asynchronous memory allocation with managed memory was significant. After profiling several homomorphic operations using CUDA `nsys` [13], we noticed that there was a difference in both individual kernel latencies, and that the GPU was actually idle a significant portion of the time during operations. Figures 1 and 2 show the multiplication operation profiled on after using only asynchronous memory and managed memory allocation respectively. We notice almost a 5-fold increase in operation latency, from 4.8 ms to 24 ms. This overhead occurs *even if memory is not oversubscribed.* After careful analysis, we realized that certain kernels were running alongside the `cudaFree` CUDA API. Because this causes a device-wide synchronization after each call, we determined that this was causing performance to degrade. Allocation of temporary variables in the middle of code would cause the destructor to be called when the code goes out of scope. This causes `cudaFree` to be implicitly called, and led to the significant delays due to the synchronization. We initially approached the situation by reusing the temporary buffers after each operation to avoid the buffers from being freed. However, this was only a temporary solution, as it led to error prone code dealing with double pointers, and was tedious to implement, with buffers still being allocated in the first iteration of each operation causing delays. Even though each operation slightly

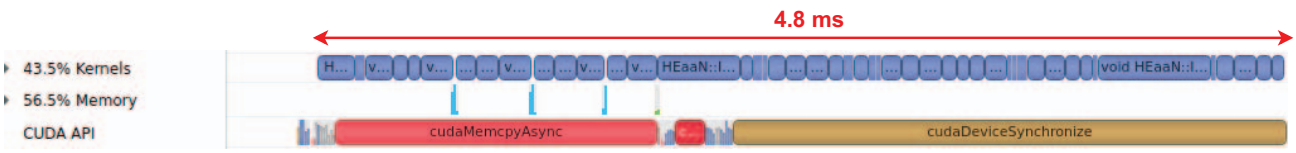

Fig. 1. Multiplication operation profiled on CUDA Nsight with memory allocated exclusively with `cudaMallocAsync` resulting in kernels with almost no idle space in between

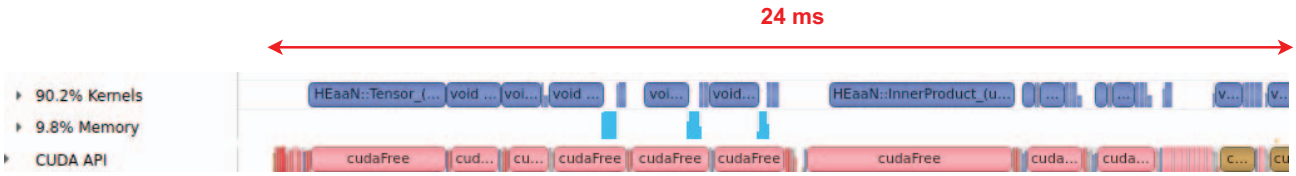

Fig. 2. Multiplication operation profiled on CUDA Nsight with memory allocated exclusively with `cudaMallocManaged` resulting in kernels separated by idle GPU space, and increased individual kernel times

sped up after dealing with temporary buffers in this manner, it was hardly a satisfactory solution.

This led us to think of the hybrid approach, where we would allocate temporary buffers with `cudaMallocAsync` and every other buffer with `cudaMallocManaged`. We added the secondary method of allocation to the general buffer allocator in the HEaaN library, and allocate all temporary buffers with an additional flag denoting that asynchronous allocation would be performed. By doing so, and also similarly applying other techniques for unified memory like prefetching and memory advising (`cudaMemAdvise`) constant values to be read only, or to forcefully reside in GPU memory buffers by setting the **SetPreferredLocation** flag, we were able to decrease operation latency to values that are close to when every buffer is completely allocated asynchronously. In addition to this, we were able to experience the OOM errors that occurred when only asynchronous memory allocation was performed and GPU memory was insufficient to cover all objects. Experimental results of the static hybrid implementation are shown in Figure 3. Parameters in CKKS are organized based on the amount of memory they consume. Refer to Table I for details on the parameters. For bootstrapping in all parameter sizes, memory is oversubscribed because all GPU memory objects do not fit within the bounds of 2GB of VRAM. For other operations, GPU memory is oversubscribed in the FVx parameter and not for the other parameter sizes.

Combining asynchronous allocation with unified memory in a hybrid approach by statically allocating the temporary buffers with `cudaMallocAsync` leads to lower latencies than the baseline 100% unified memory approach. Techniques like memory advise are also statically applied to buffers with constant values required for calculations in the GPU. Latency for the addition operation is small enough to be negligible in both approaches. For other operations like multiplication, rotation, and conjugation, there is a significant range of fluctuation for the baseline unified case in each iteration, while the asynchronously allocated case has a more stable, yet lower latency. Although it is not shown here, the hybrid allocation of temporary buffers and and other memory objects does not reduce latency to the extent of fully asynchronous memory allocations. However by performing this experiment, we are able to see that it is possible to combine the usage of both memory allocation schemes.

B. Hybrid Memory Allocation Ratios

We performed an experiment using hybrid allocation on a test CUDA program in order to determine the optimal ratios of performing asynchronous allocation with unified memory allocation. Figure 4 and 5 show the results of the program. Each sub-graph shows a different oversubscription ratio of how much of the total GPU memory available is allocated using managed memory and asynchronous allocation combined. The maximum amount that asynchronous allocation can use is 100% (technically around 95% because OOM occurs at levels above that due to a fixed amount of memory required by the CUDA driver) so we performed the experiment using eight different ratios: 0%, 25%, 50%, 75%, 80%, 85%, 90%, and 95% asynchronous allocation. For oversubscription values over 100%, the remainder memory would be allocated using `cudaMallocManaged`. We measured, the allocation latency, memory copy latency from host to device, and the kernel latencies doing two different operations: one is simple random addition, and the other is 8-point butterfly [14], which is a computation used in FFT (fast Fourier transformation) algorithms. In order to perform the kernel operations, we allocate pages in 4KB chunks and randomly mix the pointers pointing to each chunk in an array. We then execute either the addition or butterfly kernels.

We noticed that for both Figure 4, higher asynchronous allocation ratios increased the time for all operations. However, increasing oversubscription rates also significantly decreases performance of all operations. The meaning of 100% oversubscription rate is that data covers total available GPU memory once. Rates above 100% mean that data is insufficient

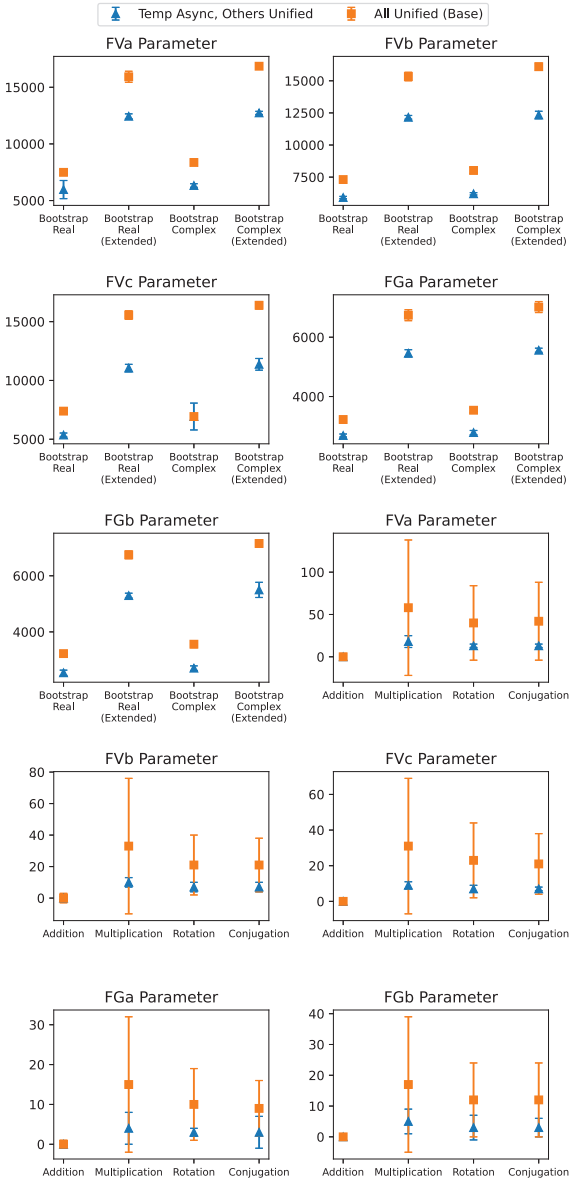

Fig. 3. Comparison of allocating all objects with UM to the hybrid method of allocating temporary buffers with `cudaMallocAsync` statically, and all other objects with UM in a Geforce GTX 1660 Ti with 6GB of VRAM, latency in milliseconds

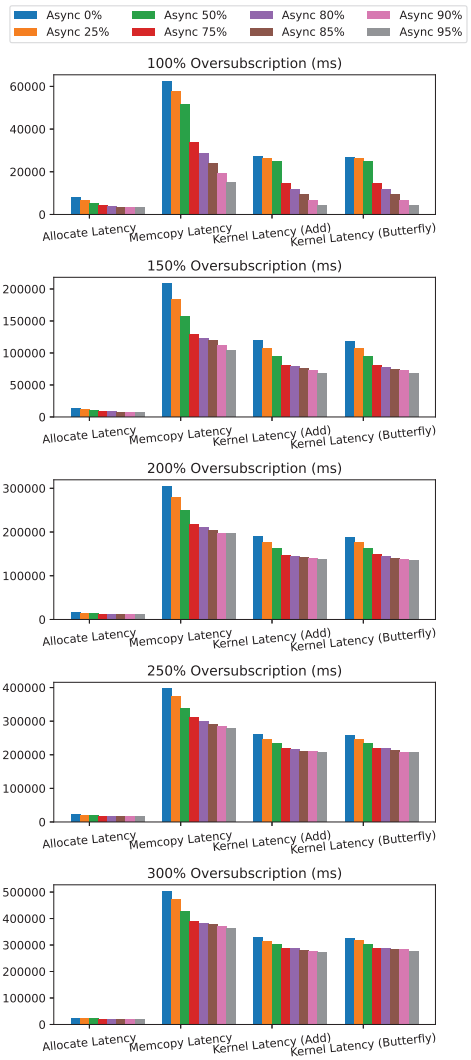

Fig. 4. Sample operational latencies depending on the asynchronously allocated ratio respective of available GPU memory. Tested on RTX 4090 GPU

to be contained fully in the GPU. When oversubscription rate is 200%, meaning that data is two times the size of total GPU memory, we can see that operational latencies are almost one magnitude of order slower than when data can be fully subscribed. This is due to page faults caused by using CUDA Unified Memory. Nevertheless, maintaining a higher asynchronous allocation mode, despite oversubscription ratios gives the greatest performance for all operations. Therefore, we apply this knowledge in the dynamic allocation scheme.

C. Profiling Allocation Scheme

Manually inserting explicit allocation commands like `allocate_async()` can be cumbersome to apply to all GPU objects in frequently maintained code bases like the HEaaN library. Due to the HEaaN library design, the majority of allocated objects except temporary buffers are implicitly allocated using the general memory buffer allocator. This requires the programmer to statically create exceptions for every single GPU buffer in the library code, depending on where the object is automatically freed once it goes out of scope. Not only is it cumbersome and bug-prone to alter the code for all of these cases, but statically modifying the allocation method for each and every object can also cause diverse run-time performances depending on the specific GPU hardware used. As a result, this results in modifying the library

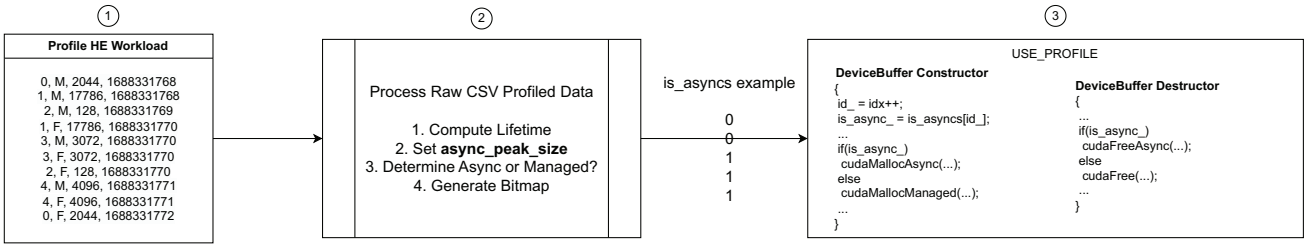

Fig. 5. Profiling Allocation Scheme Flow Diagram

code base in a very invasive manner. During the lifetime of any large-scale user workload running the HEaaN library, there are also a large number of user-determined objects that can be created like additional ciphertexts. This adds disadvantages in the static method because the user who uses the library must be aware of which allocation method they have to use. Therefore, instead of the static method, we prefer a minimally invasive dynamic method where the programmer or user does not have to directly care about which allocation method to use for every single GPU object.

Profiling homomorphic workloads in runtime can give us important details about whether or not an object should be allocated asynchronously. Our idea is based on the fact that **GPU objects with shorter lifetimes should be allocated asynchronously**, because managed memory allocation has the heavy overhead of device synchronization. Figure 5 shows a step-by-step process of this method. In step 1, we run through the entire HE workload pass once with our profiler enabled. In this stage, we collect data such as GPU object ID, whether it is allocated or freed, the size of data, and the time stamp. Once we collect the data, we store it in a CSV file, and run a Python script to process the raw CSV format. In step 2, The Python script calculates the lifetime of each GPU object based on its timestamp and when it is allocated and freed using its ID as the identifier. We also determines a threshold *async_peak_threshold* to determine the maximum amount of data that we choose to allocate asynchronously on the GPU. We organize the GPU data based on the lifetime of all GPU objects in ascending order. Then we choose N, which is the upper limit of the lifetime (in seconds) of GPU objects that we wish to allocate asynchronously. Then, we store the data in order of the GPU id, denoting a 0 for managed memory allocation, and a 1 for asynchronous allocation. Then, in step 3, we use the profiled data to allocate each GPU object accordingly in all future passes. Because the order of the allocation of objects does not change in subsequent passes, we can use this profiling method as an ”oracle” to determine the optimal choice of allocations for all future passes. Performing this method is good for workloads where repeated runs are necessary.

We perform this method using two different workloads. The lighter workload which uses less memory is the bootstrapping operation, which we use in CKKS often to decrease multiplication errors and was described earlier in Section II-D. This workload is called BM-bootstrap [15], and is a custom-made workload that tests all the different CKKS bootstrapping operations across all parameter sizes. The heavier workload is HE-enabled pre-trained ResNet-20 [16], [17] model using MNIST for inferences. Both workloads are large enough in that they do not fit in NVIDIA A40 GPU, which is equipped with 48 GB of VRAM. We created a scatterplot in Figure 7 which shows an entire view and a zoomed in view of the lifetime of all GPU objects. We plot the allocated point in green, and the free point in red, with arrows in opposite directions as markers, as shown in the graph legend. For BM-bootstrap, allocation and free continuously occurs as the program is running until the end. Most objects are freed as soon as they are allocated. However, there are a few objects which are not freed until the end of the program, shown as the sparse red dots at the end of the graph. By zooming in the object IDs until range 1,000 in the second sub-plot, we can see that a few objects are never freed until the end of the program. This is also true in the case for ResNet, however the point in which certain objects are more variable than in the case of bootstrapping. This shows that different workloads have different timings for allocation and free, and different objects have different lifetimes.

Figure 6 shows the lifetime of all objects for the BM-bootstrap and ResNet-20 workloads. The first and third graph show number of occurrences of objects with lifetime below 1 second in log scale, because they constitute the majority of all objects. In both workloads, the majority of GPU objects have a lifetime below 1 second, constituting approximately 92% of all GPU objects for BM-bootstrap. The objects in this range and lower would thus be assigned a value of 1, meaning to be allocated asynchronously (0 to be allocated using managed memory), when we set N=1 as the lifetime threshold. Beyond the first second, the number of occurrences of different lifetimes are quite variable. The second and fourth sub-plots show these objects and their lifetimes. In the case of BM-bootstrap, changing the parameters as the workload is continuously running causes the lifetime of longer objects to change, explaining the intermittent spikes in the first sub-plot.

Figure 8 shows the memory usage of asynchronously allocated and managed memory objects. The threshold value is the N value for the lifetime of objects that we set to be asynchronously allocated. The blue line shows the peak memory usage of asynchronously allocated objects at that point in time

during the execution of the workload. The orange line shows the peak memory usage of managed memory. The red line shows the maximum available GPU memory of the device, which is an A40 GPU having 48GB of total memory available. This shows that until N=256, asynchronous allocations do not consume more memory than physically available, and once lifetime threshold becomes 512, we can see some potential for OOM errors to arise due to memory consuming more than physically available device GPU memory.

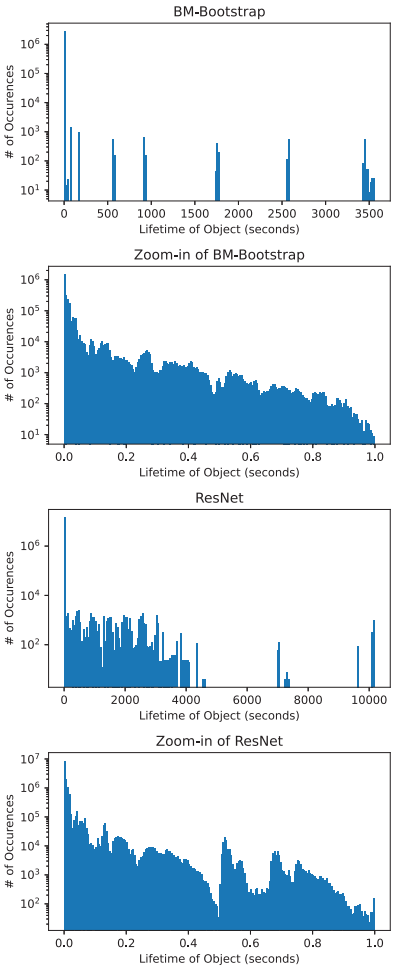

Fig. 6. Mode graph showing the number of occurrences (log scale) of each object depending on their lifetime in seconds

D. Dynamic Allocation Scheme

Profiling requires the workload to be run to completion initially in a single pass to know the lifetime of all the GPU objects. It is not truly dynamic in the sense that optimizations can only occur after the initial pass. If the workload takes a long time, and is not repeated more than once, using the profiling scheme may not provide any benefit at all. We introduce a dynamic scheme in this section, which does not require any prior knowledge of the allocated GPU objects, and

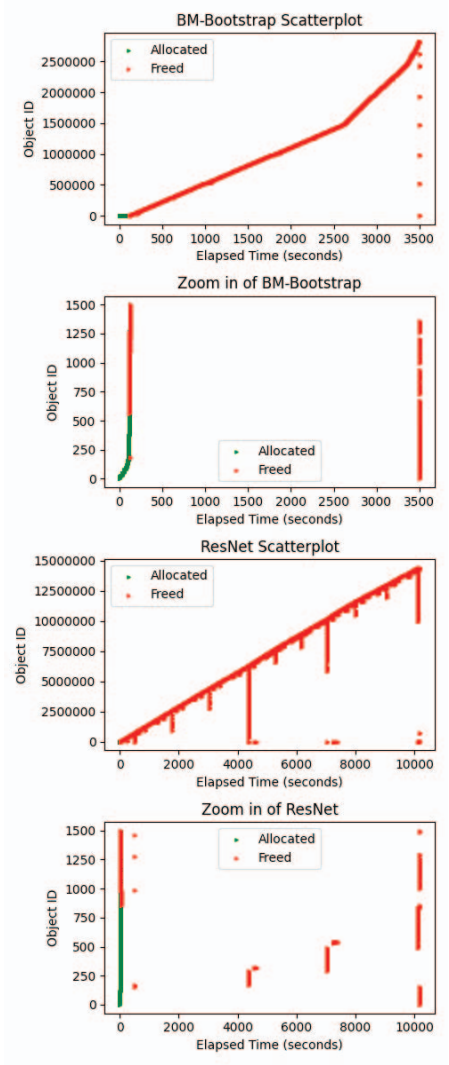

Fig. 7. Allocation and free scatter plot of all GPU objects created during the lifetime of workload

adjusts the allocation methods on-the-fly. We take advantage of the fact that objects with short lifetimes are repeatedly re-allocated as the program progresses. When the objects are re-allocated, we use our decision process to determine whether the object should be allocated asynchronously or under managed memory. Therefore we do not have to perform any modifications of existing pointer allocations by performing pointer switching or memory transfers.

Figure 9 shows an example of how we perform our decision making. Because there is no ”oracle” to give us the deterministic outcome of the lifetime of every single object, we have to use heuristics in this approach. The information given to us during every memory allocation is only the size of the object to be allocated. Therefore we maintain a map where the key is the size of the buffer to be allocated and the value is the number of active allocations that have not been freed

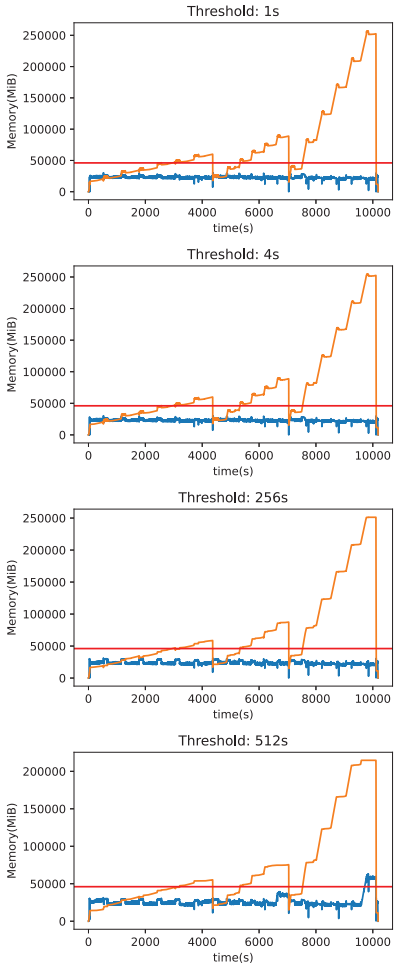

Fig. 8. Peak memory usage of GPU objects depending on time on an A40 GPU equipped with 48 GB VRAM

yet, along with a ten-element active list. Every time an object has been freed, the total number of allocations is reduced by one. Every time an allocation is made, the total number of allocations increases by one. We also maintain an active list, that records the number of active allocations of that particular size currently residing on the GPU. Currently the size of the list is 10. Every time an allocation is made, the total number of allocations is pushed to the end of the list. Once the list is filled up, the first element is then replaced. When the list has not been filled up to ten elements yet, we allocate all objects using managed memory. Once the list is filled up, we measure the range of the queue by getting the maximum value and subtracting it with the minimum value. Then we compare the range with a preset value that we decide, and use it to determine whether or not the object should be asynchronously allocated, as shown in steps 3-5 of Figure 9.

We noticed a trend in that if objects have a short lifetime, there tends to be a repetitive tendency of allocation and frees happening within a short time span. In this case, the range of the list would be small, because active objects would continuously increase and decrease in a similar pattern. Objects that have longer lifetimes would actually increase the number of active objects, and thus increase the range of the active list. Therefore, we use a conditional branch to decide if the range is below a certain X value, to allocate all objects with that size in that manner, until the range of the active list changes again as more objects are added. This heuristic doesn't guarantee that all objects with short life times are asynchronously allocated, like in the previous section, but it does not require a profiling step, and also does not require static modifications to any of the HEaaN library code, in that only the allocator module is changed in an isolated manner.

IV. Evaluation

We evaluated the performance of our three schemes on three different GPUs: One is a server-equipped A40 GPU with 48 GB of RAM, the second is a retail GeForce GTX 1660 Ti with 6GB of RAM, and lastly is the retail GeForce GTX 1050 with 2GB of RAM. We ran different workloads depending on the type of GPU used, and resorted to larger workloads for the more powerful GPU because it had more available VRAM. We run CUDA 12 to ensure that we are compatible with all the outlined features. All workloads are custom workloads built in C++ and interact with the HEaaN library. Our schemes make direct modifications to the HEaaN library, and depending on the scheme used the amount of invasiveness to library code varies. The dynamic scheme is the least invasive in that only modifications to the memory allocator are made, and the static is the most invasive in that actual library code is manually modified in the way that certain temporary objects should be allocated. The profiling allocator uses external resources like files and storage to store the raw CSV data, and Python scripts to process the data into a format that can be used by the allocator in the actual execution phase.

For the A40 GPU, we ran larger workloads which take up more memory like BM-bootstrap and ResNet, utilizing approximately 80GB and 200GB of memory respectively. We tested only the profiling allocation scheme of Section III-C on the A40 GPU because it has the best performance out of the three schemes. BM-bootstrap is a bootstrapping benchmark with different bootstrapping operations ordered after each other. After each operation, the parameters and corresponding keys required are switched to go to to the next operation. ResNet is a pre-trained homomorphic version of the DL ResNet model, and we test the inference operation in the homomorphic state. Figure 10 shows the performance of each bootstrapping operation, and the peak asynchronous memory threshold (ratio of asynchronous allocations) that we set in the load run to allocate GPU objects. We noticed that for most of the bootstrapping operations, setting a threshold value in the mid-range of around 40-60% leads to the fastest results in bootstrapping time. In the case of ResNet, using a peak asynchronous memory threshold of 65% gives the best performance. The red line denotes manual swap performance, where the programmer is responsible for swapping out GPU

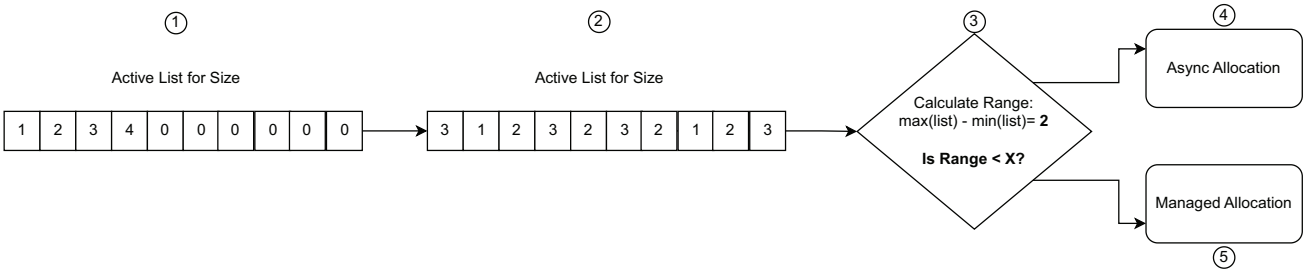

Fig. 9. Dynamic Allocation Scheme Example

objects into host memory manually. ResNet latency is about 1,657 seconds, and using profiled allocation reduces it by approximately 22% to 1,296 seconds.

Figure 11 and 12 show the latency results for multiple HE functions for the FGa, and FGb, and FVb parameters on the GeForce 1660Ti and 1050 GPU cards. For the larger parameters, static seems to have the best performance. For the smaller parameters, the optimal approach is not so clear. Dynamic is highly advantageous because it does not require any modifications of inner library source code. Profiling was performed and applied with a 68% asynchronous allocation rate. Fully managed memory had the slowest performance with the greatest variance in all tests. For the 1050 in Figure 11, we compared static and dynamic performance. Both had similar performance, with static being slightly faster. This is due to being able to exactly pinpoint which objects in code we had to allocate asynchronously, with a straightforward benchmark tool testing only the homomorphic functions.

V. Conclusion and Future Work

In conclusion, we designed three schemes that utilize a hybrid allocation strategy combining `cudaMallocAsync` and `cudaMallocManaged` to outperform the purely managed memory scheme in all HE workloads. We have provided a static allocation scheme that improves unified memory performance by 31%. We have also discovered that using the profiling allocation scheme gives a performance that is 22% better than manual swapping of objects in and out of GPU memory by an experienced programmer who extensively knows the HEaaN library code base. Finally, we implement a dynamic allocation scheme that gives up to 50% better performance in bootstrapping operations than the baseline fully unified memory case in without having the disadvantage of having to profile or alter the code-base, by dynamically adjusting the allocation of objects on-the-fly.

In our future work, we will improve upon the dynamic allocation scheme, so that it can give better performance, hopefully being closer to the ”oracle” profiling allocation version which knows the lifetime of all objects in advance, but without having the overhead of having to profile the entire workload before hand.

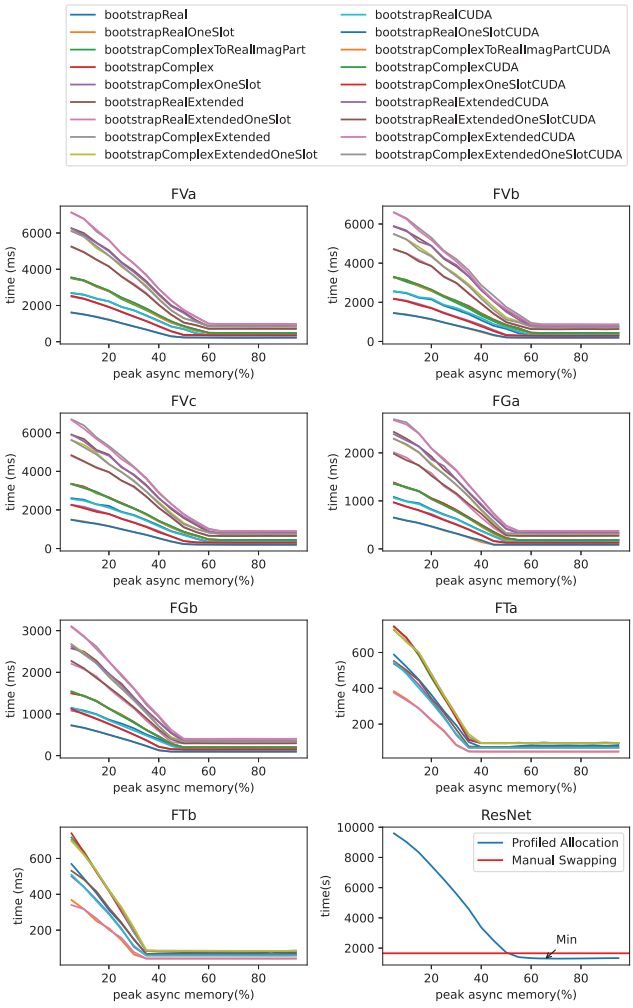

Fig. 10. Performance of BM-Bootstrap Depending on Peak Asynchronous Allocation Rate on A40 GPU with 48 GB of VRAM

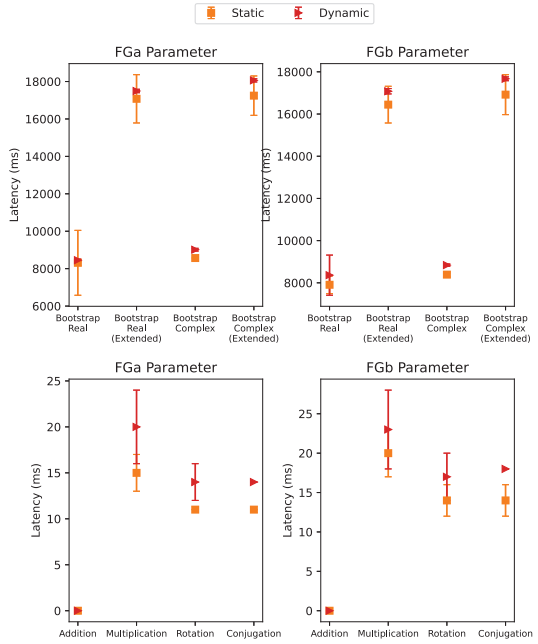

Fig. 11. Performance of Different HE Functions On 1050 with 2GB VRAM

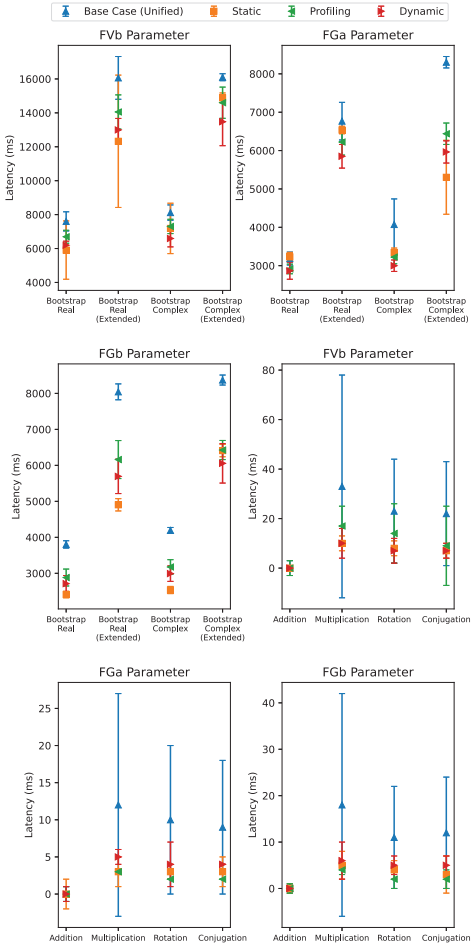

Fig. 12. Performance of Different HE Functions On 1660 Ti with 6GB VRAM

REFERENCES

[1] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," Cryptology ePrint Archive, Paper 2021/508, 2021, https://eprint.iacr.org/2021/508. [Online]. Available: https://eprint.iacr.org/2021/508

[2] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[3] N. Sakharnykh, "Beyond gpu memory limits with unified memory on pascal," Dec 2016. [Online]. Available: https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/

[4] V. Kini and J. Hemstad, "Using the nvidia cuda stream-ordered memory allocator, part 1," Aug 2022. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-1/

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus." *IACR Cryptology ePrint Archive*, vol. 2018, p. 421, 2018. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2018.htmlChillottiGGI18

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012, https://eprint.iacr.org/2012/144. [Online]. Available: https://eprint.iacr.org/2012/144

[8] [Online]. Available: https://heaan.it/docs/heaan/ParameterPreset 8hpp.html

[9] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "Heaan.mlir: An optimizing compiler for fast ring-based homomorphic encryption," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591228

[10] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, oct 2020. [Online]. Available: https://doi.org/10.1109/iiswc50251.2020.00033

[11] A. Kim, A. Papadimitriou, and Y. Polyakov, "Approximate homomorphic encryption with reduced approximation error," Cryptology ePrint Archive, Paper 2020/1118, 2020, https://eprint.iacr.org/2020/1118. [Online]. Available: https://eprint.iacr.org/2020/1118

[12] L. Jiang and L. Ju, "Fhebench: Benchmarking fully homomorphic encryption schemes," 2022.

[13] NVIDIA, "Nvidia nsight systems user guide." Mar 2023. [Online]. Available: https://docs.nvidia.com/nsight-systems/UserGuide/index.html

[14] Z. Li, H. Jia, Y. Zhang, T. Chen, L. Yuan, and R. Vuduc, "Automatic generation of high-performance fft kernels on arm and x86 cpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1925–1941, 2020.

[15] [Online]. Available: https://heaan.it/docs/heaan/

[16] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[17] J. H. Cheon, M. Kang, T. Kim, J. Jung, and Y. Yeo, "High-throughput deep convolutional neural networks on fully homomorphic encryption using channel-by-channel packing," Cryptology ePrint Archive, Paper 2023/632, 2023, https://eprint.iacr.org/2023/632. [Online]. Available: https://eprint.iacr.org/2023/632