applied sciences

MDPI

Article

Optimized Deep Neural Networks for Real-Time Object Classification on Embedded GPUs †

Syed Tahir Hussain Rizvi [1,*], Gianpiero Cabodi [1] and Gianluca Francini [2]

[1] Dipartimento di Automatica e Informatica (DAUIN), Politecnico di Torino, 10129 Turin, Italy; gianpiero.cabodi@polito.it

[2] Joint Open Lab, Telecom Italia Mobile (TIM), 10129 Turin, Italy; gianluca.francini@telecomitalia.it

* Correspondence: syed.rizvi@polito.it

† This paper is an extended version of our paper published in Proceedings of International Conference on Control, Decision and Information Technologies (CoDIT'17) as Syed Tahir Hussain Rizvi, Gianpiero Cabodi and Gianluca Francini, "GPU-only Unified ConvMM Layer for Neural Classifiers", 2017 International Conference on Control, Decision and Information Technologies (CoDIT'17), Barcelona, Spain, 2017.

Received: 19 July 2017; Accepted: 10 August 2017; Published: 11 August 2017

Abstract: Convolution is the most computationally intensive task of the Convolutional Neural Network (CNN). It requires a lot of memory storage and computational power. There are different approaches to compute the solution of convolution and reduce its computational complexity. In this paper, a matrix multiplication-based convolution (ConvMM) approach is fully parallelized using concurrent resources of GPU (Graphics Processing Unit) and optimized, considerably improving the performance of the image classifiers and making them applicable to real-time embedded applications. The flow of this CUDA (Compute Unified Device Architecture)-based scheme is optimized using unified memory and hardware-dependent acceleration of matrix multiplication. Proposed flow is evaluated on two different embedded platforms: first on an Nvidia Jetson TX1 embedded board and then on a Tegra K1 GPU of an Nvidia Shield K1 Tablet. The performance of this optimized and accelerated convolutional layer is compared with its sequential and heterogeneous versions. Results show that the proposed scheme significantly improves the overall results including energy efficiency, storage requirement and inference performance. In particular, the proposed scheme on embedded GPUs is hundreds of times faster than the sequential version and delivers tens of times higher performance than the heterogeneous approach.

Keywords: concurrent computing; general purpose GPU; unified memory; convolutional neural networks; heterogeneous; matrix multiplication; CUDA basic linear algebra subroutines (cuBLAS); embedded platform

1. Introduction

Convolutional Neural Network (CNN) exhibits unmatched performance in different computer vision applications. CNN is the type of neural classifier that generates features hierarchically using convolutions. Convolution is a computationally intensive task that can be accelerated and optimized by exploiting the computational power and resources of a GPU [1]. Convolutional Neural Network (CNN) is normally composed of convolutional layers, pooling layers, some normalization functions and fully connected layers. However, high accuracy of neural classifiers is mainly dependent on the number of stacked convolutional layers [2,3]. State-of-the-art deep classifiers are comprised of tens or hundreds of these convolutional layers [3–5]. Where the increasing number of convolutional layers boosts the accuracy of a classifier, this enlarged depth also increases the training/classification time, arithmetical complexity and storage requirements. With the ongoing traction of embedded computing, the efficiency of an algorithm is important. Therefore, the realization of deep architectures

is critical on a single GPU of an embedded platform. A significant performance improvement can be achieved by reducing the computational burden of these convolutional layers, making them realizable on embedded GPUs.

There are several approaches to compute the Convolution operation [6–12]. Fast Fourier transformation (FFT), Winograd minimal filtering algorithm, the look-up table and matrix multiplication-based convolution are a few of them. In FFT-based convolution, signals (images and filters) are transformed and multiplied pointwise in the frequency domain to reduce the number of multiplications [6]. However, this transformation between the time and the frequency domain is computationally intensive and limits the performance gain. Furthermore, FFT-based convolution is fast for the large filters but state-of-the-art deep classifiers are using small filters in convolutional layers [4]. For networks having small size filters, Winograd-based convolution shows good results [7]. This algorithm reduces the arithmetic complexity of the convolutional layer by using a minimal filtering technique. These approaches to compute the convolution can further be optimized by using different techniques and schemes [12–14].

Convolution operations can also be computed using standard matrix multiplication. Various software libraries provide matrix multiplication subroutines having near-optimal performance; however, it still remains a challenge to utilize these libraries and the computational power of a single GPU efficiently and provide an optimized framework for deep classification on embedded platforms.

In this paper, an accelerated and optimized scheme for object classification is proposed for the embedded platforms. Results show that the proposed extensions enable a speedup of several orders of magnitude over the heterogeneous and the sequential versions and can be used to perform the real-time image classification on an embedded platform with lesser memory requirement, opening up a whole range of applications. In order to achieve the optimized flow, first of all, the heterogeneous matrix multiplication-based convolution (ConvMM) approach is parallelized using the concurrent computational resources of the GPU [1]. This transformation is performed to eliminate the extra memory transfers required by the Heterogeneous ConvMM approach [15]. Flow of this GPU-only ConvMM layer and all CUDA-based required layers (pooling, Regularization Units etc.) is optimized using Unified memory. By adopting this optimized data transfer scheme, the memory access latency of CUDA-based deep architectures is reduced and overhead caused by the explicit data movements is eliminated. Additionally, this flow is further optimized and accelerated using two different hardware dependent matrix multiplication approaches for Convolution operation. The proposed scheme is realized by reforming the flow of heterogeneously implemented image classification architectures presented in [15]. Results are evaluated on two different types of embedded GPUs. The experiments demonstrate that the proposed optimized scheme shows order of magnitude performance improvements over the sequential and the heterogeneous versions for both platforms.

The remaining sections of this paper are arranged as follows: Section 2 reviews the related work in the field of mobile phone-based image classification frameworks. Section 3 discusses the flow of the Heterogeneous ConvMM layer and its transformation into the concurrent GPU-only ConvMM layer. Section 4 explains the data transfer scheme used in this work to optimize the flow of the GPU-only ConvMM layer. Two different Matrix multiplication approaches to accelerate the optimized flow of the convolutional layer are described in Section 5. Section 6 reports the experimental results and discusses the performance of proposed scheme. Finally, Section 7 concludes the research.

2. Related Work

Compared with desktop-based real-time image classification, which has been broadly studied and addressed, there have been fewer studies contributing to mobile and embedded platforms-based classifiers. There are many libraries such as Torch and Caffe for desktop and server platforms; however, these optimized libraries cannot be directly used on mobile platforms due to hardware and software constraints and dependencies. Furthermore, these libraries require installation of multiple computational packages on an embedded device having limited storage capacity available. Realization

of deep classifiers on a low-power mobile platform is still a challenge due to massive computational and storage requirements.

An Open Computing Language (OpenCL)-based framework "DeepSense" is proposed for mobile devices in [16]. Results show that the proposed scalable framework can execute a variety of CNN models with no or marginal accuracy tradeoffs. However, optimization of this framework is suggested to execute the large-scale models in real time along with lower power consumption.

A GPU-accelerated library "CNNdroid" is presented in [17]. This library is realized to execute the trained deep CNNs on Android-based mobile devices. However, the proposed library does not support all CNN layers required by the state-of-the-art deep architectures like Residual Network.

An efficient deep neural network (DNN) flow optimized for mobile GPU is presented in [18]. Different optimization techniques to achieve the higher performance and better power efficiency are discussed. However, the hardware information that is presented is not sufficient to compare the results.

Some frameworks are also proposed to compress the convolutional neural networks for mobile frameworks [19,20]. By quantizing the trainable parameters, computations are accelerated, and power consumption and storage overhead are reduced at the cost of some loss in classification accuracy. However, state-of-the-art deep architecture like ResNet-34 can be directly used for mobile-based real-time classification, which has fewer parameters than other networks, yet provides state-of-the-art accuracy.

Concluding, the contributions of the proposed scheme and improvements over the mentioned references can be summarized as follows. First, the proposed scheme supports nearly all required layers of deep architectures realized using CUDA computing framework. Intermediate frameworks are avoided to resolve the problem of software dependencies and memory consuming computational packages on the embedded platforms; only CUDA-based libraries and functions are used for the realization of optimized flow. Second, optimized GPU-only flow and Unified memory transfer scheme are proposed to avoid extra memory transfers and double allocation of parameters, resulting in low power and storage consumption. Finally, hardware dependent selection of accelerated matrix multiplication operation for the convolutional layer is employed to further optimize and accelerate the deep architectures for the real-time object classification.

3. Heterogeneous and GPU-Only ConvMM Layers

One approach to compute the convolution operation is Matrix multiplication-based convolution (ConvMM). Using this approach, first multi-dimensional input image and filters are transformed into two-dimensional matrices and then multiplied to achieve the results equivalent to the traditional convolution operation. The transformation step can be performed by copying the tiles of pixels from original image/filter to the matrices in a specific order. This is an operation where the sequence has to be considered and can be performed efficiently using the CPU [15]. Once transformed, these matrices can be multiplied efficiently using concurrent resources of GPU. Using the heterogeneous resources (CPU-GPU) of a system, an algorithm like this can benefit in terms of execution time [15,21,22]. However, extra memory transfers caused by the heterogeneous implementation can also break the overall performance.

Figure 1 illustrates the flow of heterogeneous ConvMM layer where the transformation of data (input maps and filters) is done using the CPU and multiplication of this transformed data is performed using the GPU [15]. Before transformation, the input data is padded with zeros according to the architecture of the trained network.

In this transformation step, multi-dimensional input maps and filters are converted to 2-dimeniosnal matrices. Data transformation is performed by copying the tiles from multi-dimensional maps and placing them in a specific order in the matrices as demonstrated by Figure 2.

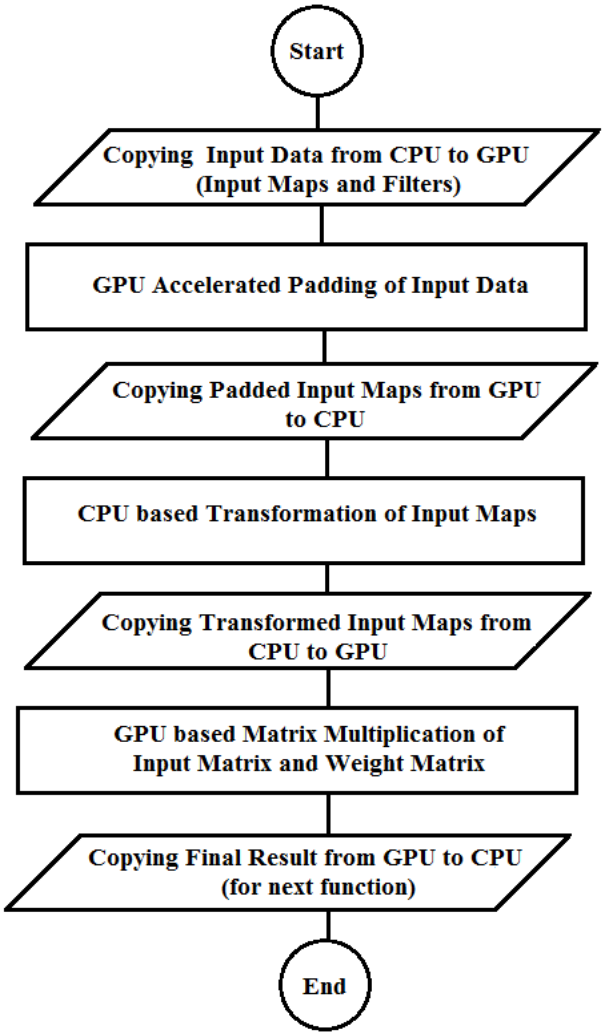

Figure 1. Flow of the heterogeneous ConvMM layer.

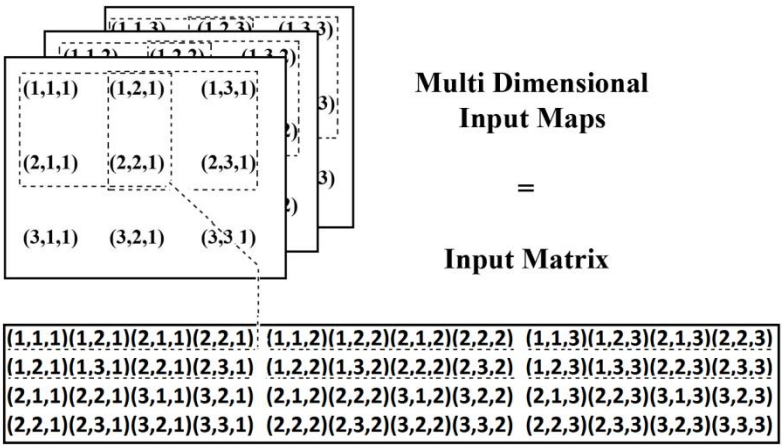

Figure 2. Transformation of multi-dimensional input maps to the 2-dimensional input matrix.

There are four data transfers between the CPU and the GPU as depicted in Figure 1. Out of four, two data transfers are made to perform the CPU-based transformation, which is a sequence-sensitive operation and is performed using the high operational frequency of CPU [15].

If this CPU-based transformation step is parallelized and performed using the GPU, so these extra two transfers can be avoided and all tasks can be performed using the GPU as shown in Figure 3. This multi-dimensional input map can be concurrently converted to the input matrix using following formula. In this formula, *k_h* and *k_w* are representing the height and width of the filter. Number of rows, columns and dimensions of the input image are represented by *i_Row*, *i_Col* and *i_Dim* respectively. *o_Row* and *o_Cols* are representing the number of rows and columns of final output after convolution.

$$Input\_matrix_{(row,col)} = \sum_{i=1}^{o\_Row} \sum_{j=1}^{o\_Col} \left(\sum_{k=1}^{i\_Dim} \sum_{h=1}^{k\_h} \sum_{w=1}^{k\_w} Input\_map_{(i+h,j+w,k)} \right) \quad (1)$$

where $o\_Row = i\_Row - k\_h + 1$; $o\_Col = i\_Col - k\_w + 1$; $row = i \times (o\_Col) + j$; $col = (k \times k\_w \times k\_h + (h \times k\_w) + w)$.

If input map is of size 3 × 3 (*i_Row* × *i_Col*) and size of the filter is 2 × 2 (*k_h* × *k_w*), so the output map after convolution would be of size 2 × 2 (*o_Row* × *o_Col*) with padding and stride of 0 and 1 respectively.

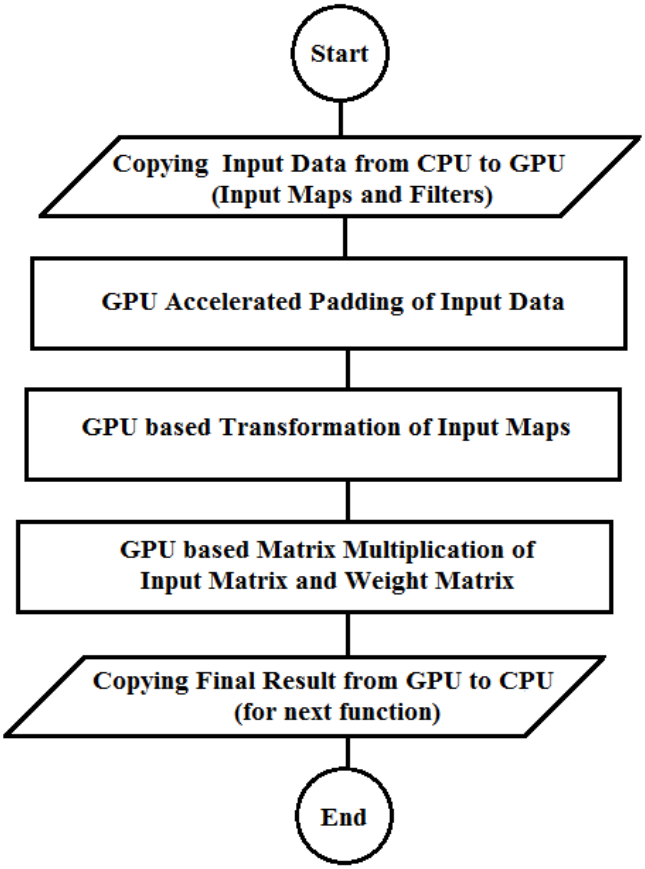

Figure 3. Flow of the GPU-only ConvMM layer.

By parallelizing the transformation step, the number of data transfers between the host and the device are minimized that can provide the substantial improvement in performance.

4. Optimized Data Transfers Scheme for GPU-Only ConvMM

In this section, different memory-based schemes are discussed and unified memory is proposed to optimize the architecture of GPU-only ConvMM and other required layers for the image classification on embedded devices.

4.1. Pinned Memory Based Allocations

For CUDA-based realized layers, input images or data must be transferred from the host (CPU) memory to the device (GPU) memory, and the final results must be copied back to the CPU memory from the GPU memory. These transfers can occur several times depending on the architecture of a deep classifier and may affect the overall performance of the system. By default, host allocations are pageable and the GPU cannot copy data directly from the pageable host memory. So whenever a data transfer is invoked, CUDA driver must allocate and copy host data into the pinned (page-locked) memory from the pageable memory and then transfer it to the device memory via PCI-express (Peripheral Component Interconnect-Express) bus as shown in Figure 4.

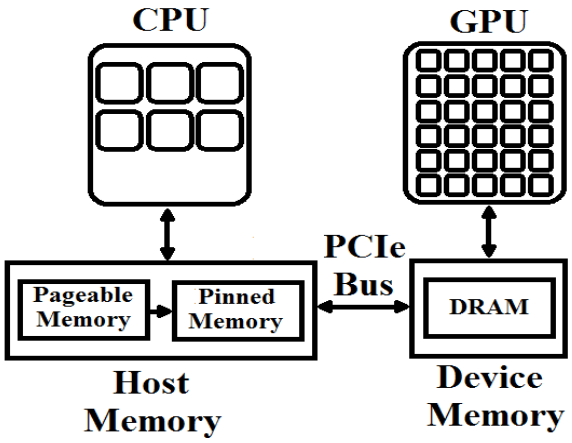

Figure 4. Pageable and pinned memories on the host side.

The cost of this extra transfer between the pageable and the pinned memories can be avoided by directly allocating the host data in the pinned memory as shown in Figure 5. CUDA provides the feature of allocating the input data into the pinned memory.

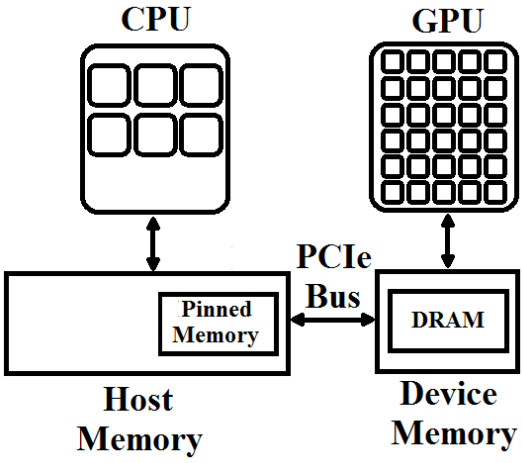

Figure 5. Pinned data transfer.

Use of pinned memory is a common practice in CUDA-based acceleration. However, there are some constraints to be considered before using the pinned memory specifically on an embedded platform (like size of trainable parameters to be imported). Pinned memory should not be over-allocated because this can reduce the amount of physical memory (Random Access Memory (RAM)) to the other programs and operating system, which can reduce the overall performance of the

used system. Hence, the storage capacity of an embedded device is limited, so use of pinned memory can affect the performance of deep classifiers.

4.2. Proposed Unified Memory-based Allocations for GPU-Only ConvMM and Other Layers

Typical desktop systems have the separate host (CPU) and device (GPU) memories as shown in Figure 6. As discussed in the previous section, data transactions between the CPU and the GPU memories are very frequent. The speed of these data transfers is dependent on the PCI express bus. These transfers add the overhead and a lot of complexity to flow of an algorithm as can be compared and visualized by the Figures 1–3. A programmer has to consider these separated memories to define the flow of an algorithm.

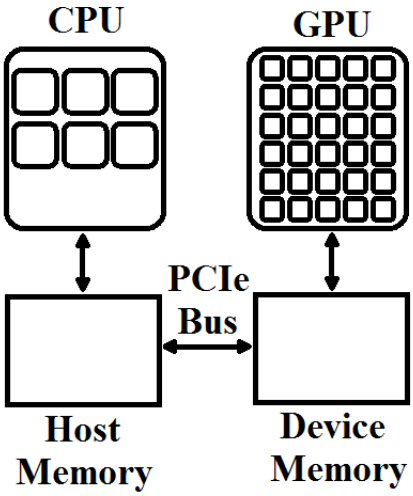

Figure 6. Host (CPU) and device (GPU) memories.

With CUDA 6.0, Unified memory (UM) is introduced which defines a pool of shared memory for the CPU and the GPU as shown in Figure 7. This shared memory can be accessed by both CPU and the GPU. The same allocated variable can be used by a CPU function and a GPU kernel according to the requirement of the program. However, data synchronization is essential after kernel execution.

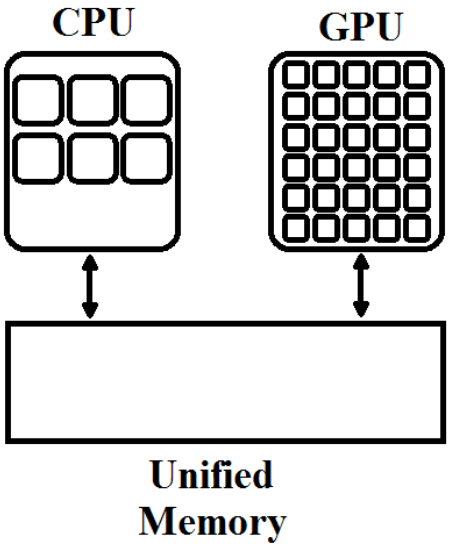

Figure 7. Unified memory.

Most of the embedded and mobile platforms have physically unified memories, so this includes both GPU as well as CPU requirements. Prior to CUDA v6.0, two copies of data elements were required, one in CPU memory and another in GPU memory. However, by using unified memory scheme, a single copy of trainable parameters is required to be allocated in memory. This can be useful for realization of deep architectures on embedded platforms having low storage capacity.

In this work, Unified memory is used to realize the optimize flow of deep architectures on embedded platforms.

As depicted in Figure 3, at least two data transfers are essential for every layer to bring the input data to the GPU for the computations and to move back the results from the GPU to the CPU for the next operation/layer. Using unified memory, the flow of this GPU-only ConvMM and other required layers can be further simplified and optimized as shown in Figure 8. This is done by allocating the inputs and the outputs in the unified memory space from the main program. By using unified memory in the ConvMM and other layers, the output of one layer can be directly fed to the next layer.

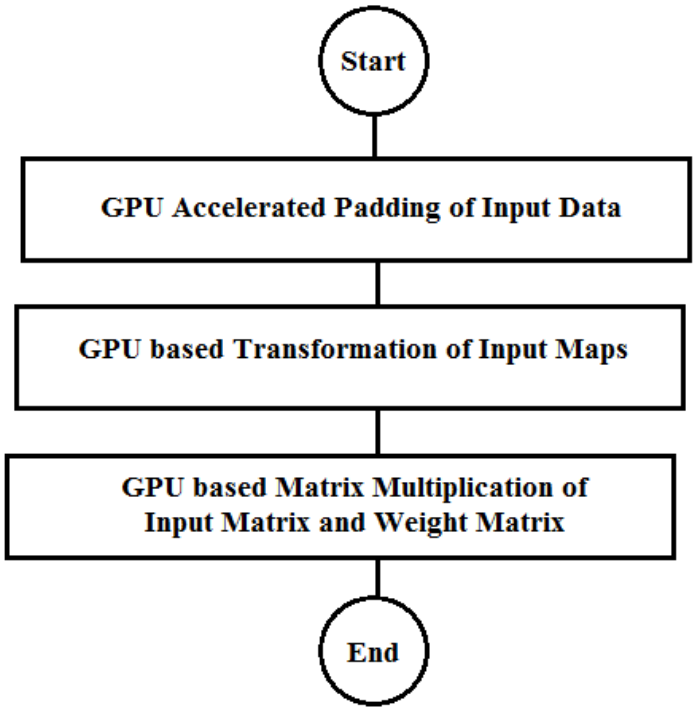

Figure 8. Flow of the GPU accelerated Unified ConvMM Layer.

5. Accelerated Matrix Multiplication for GPU-Only ConvMM

In this section, matrix multiplication operation required by the ConvMM layer is accelerated for the real-time image classification. Depending on the different compute capabilities of the used embedded platforms, two different approaches to compute the accelerated matrix multiplication (AMM) are proposed as shown in Figure 9. Compute Capability (C.C.) of a GPU device defines its available features and restrictions. By using accelerated matrix multiplication, proposed scheme is further accelerated and optimized.

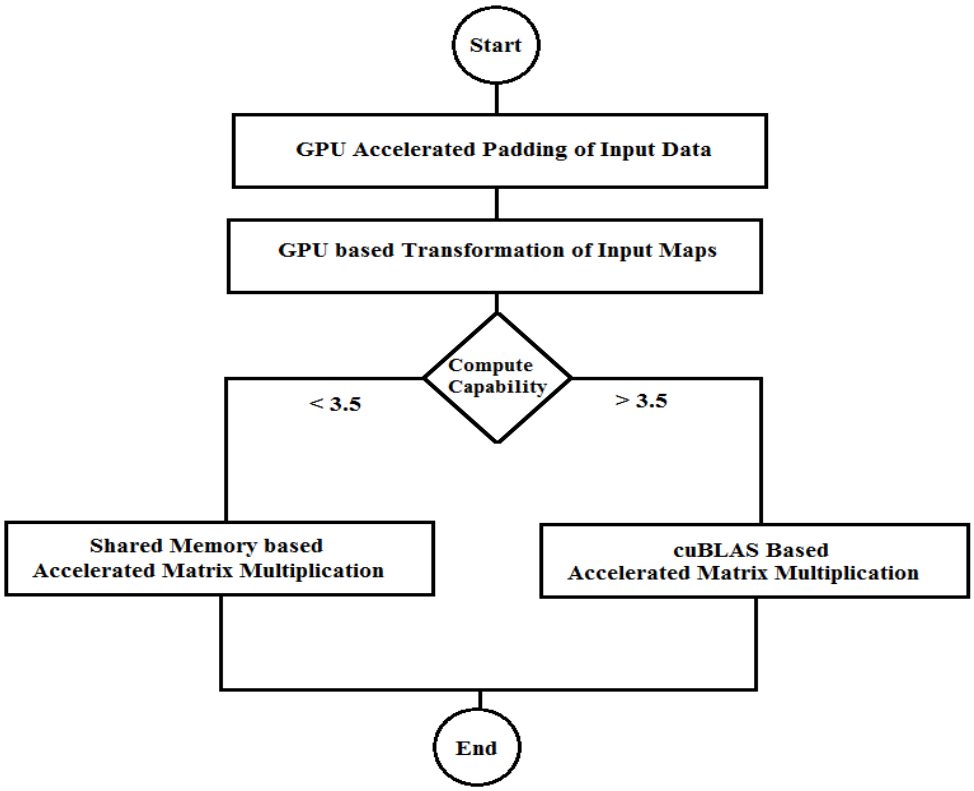

Figure 9. Flow of the accelerated matrix multiplication-based convolution with unified memory.

5.1. cuBLAS Accelerated Matrix Multiplication Convolution (ConvCAMM)

CUDA Basic Linear Algebra Subroutines (cuBLAS) library provides the high-performance functions for the numerical operations and shows the order of magnitude performance improvements over the other libraries [23,24]. This library also contains various General Matrix-Matrix Multiply (GEMM) routines for the different types of operands (like complex, single data type, double data type etc.) that can be useful to accelerate the ConvMM layer [25,26]. However, some constraints need to be considered before using the subroutines of cuBLAS.

The first aspect to consider is the compute capability of the GPU platform because the cuBLAS library cannot be used on a GPU having compute capability less than 3.5. Next constraint is the memory storage layout of the cuBLAS library, which is in column-major order. Since C++ language follows the row-major order, applications written in C/C++ language cannot use the native semantics for the 2-dimensional arrays and data needs to be transformed from row to column-major order for the cuBLAS subroutines. This transformation between the row and the column-major order for every subroutine can break the performance of a program instead of accelerating it.

In this work, single precision GEMM (SGEMM) routine of cuBLAS is used to accelerate the matrix multiplication of the input and the filter matrices (for GPU devices having compute capability greater than 3.5). This cuBLAS accelerated Matrix Multiplication (CAMM) is employed in the proposed GPU-only Convolution scheme.

The problem of storage layout is resolved for the matrix multiplication subroutine. Since the cuBLAS always assume that the matrices are stored in the column-major order, so the matrices are transformed implicitly before passing to the multiplication subroutine and after getting back the resultant matrix. This does not cause a buffer overrun, but it effectively transposes the matrices without actually moving data around in the memory. Thus, the problem of memory layout is resolved by passing the matrices in reverse order ($C' = B' \times A'$ instead of $C = A \times B$), and the resultant matrix

would be implicitly transposed before passing to main application that is the required result ($C = (C')'$) in the row-major order for the ConvMM layer.

5.2. Shared Memory Based Accelerated Matrix Multiplication Convolution (ConvSAMM)

GPUs are equipped with different memory hierarchies which offer various performance characteristics. For the GPU platforms having compute capability less than 3.5, shared memory-based accelerated matrix multiplication (SAMM) is employed in the proposed flow of the deep architectures. Shared memory is much faster than the global memory used by the current version of GPU-only ConvMM layer. A profitable way to perform the computations on the GPU is to divide the data into the subsets or the tiles that fit into the shared memory and copying these tiles from the global memory to the shared memory. Using this technique, memory level parallelism can be achieved and the global memory access can be reduced which is a performance bottleneck.

There are many redundant global memory accesses in the naive matrix multiplication approach as shown in Figure 10a. It can be analyzed that the same row elements of matrix A are required for every row element of resultant matrix C and similarly to compute the every column element of the resultant matrix C, all the column elements of matrix B are required from the global memory. Therefore, shared memory can be used to eliminate this redundancy and reduce the global memory accesses.

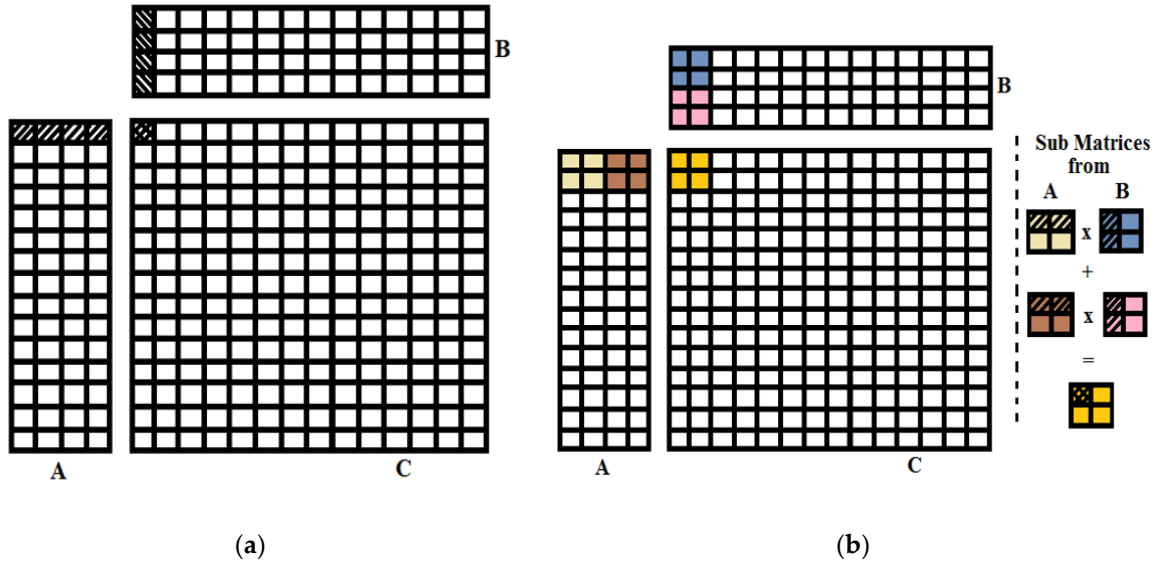

Figure 10. (**a**) Naive matrix multiplication approach; (**b**) Shared memory-based matrix multiplication approach (Different colors are representing the division of data into tiles in shared memory).

Using the shared Memory, input matrices (A and B) are divided into the tiles (sub-matrices) to assign to the thread blocks, and multiplication of these tiles is performed independently as shown in Figure 10b. Shared memory based division of input matrices and resultant matrix into tiles is represented by different colors. By summing up the results of these multiplications, elements of the resultant tile (of matrix C) are computed. It can be noted that there is a lot of data redundancy within a tile. To compute the all elements on the same row of the submatrix C, same data of the submatrix A is required and hence it is in shared memory, so it can be accessed faster. Performance of this matrix multiplication is depended on the appropriate selection of the tile size. This tile size should be kept variable and selected according to size of matrices.

6. Experiments and Results

In this work, CUDA computing framework is used to realize the optimized GPU-only ConvMM layer and its heterogeneous version for comparison. All required layers (like Pooling, Rectifier

Linear Units) to construct a deep architecture are implemented and optimized using the discussed Unified memory scheme. The performance of proposed flow is evaluated over three different image classification networks (Alex Krizhevsky's network for CIFAR-10, OverFeat and ResNet-34) [4,5]. These classification networks are constructed and trained using the Torch computing framework. Alex's CIFAR-10 is trained over the CIFAR-10 dataset while the ImageNet dataset is used to train the other two networks. These neural classifiers ranging from smaller network to deeper architectures are selected to verify the performance of proposed framework for the different type of architectures. Trained parameters are imported in CUDA and utilized by the proposed scheme.

Experiments are performed on two different types of embedded platforms. The first platform is the Nvidia Jetson TX1 development board. Jetson TX1 is an embedded platform with quad-core ARM Cortex A-57 processor, 4 GB of RAM and a Maxwell GPU having 256 CUDA cores. The second platform is the Nvidia Shield K1 tablet equipped with Kepler K1 GPU, which has a quad-core Cortex A-15 CPU of 2.2 GHz, 192 CUDA Cores and 2 GB of shared RAM memory. Compute capability of Jetson TX1 and Tegra K1 GPUs are 5.3 and 3.2 respectively.

6.1. Performance Evaluation of Proposed Scheme on Jetson TX1 (For GPUs having C.C. > 3.5)

First of all, comparison of different versions of ConvMM layers is performed on the Jetson TX1 embedded board. Table 1 presents the performance of different ConvMM layers for various image sizes. Results show that the GPU-only ConvMM layer is approximately 2× faster than the heterogeneous version. This gain in performance is due to the elimination of extra memory transfers required by the CPU-based transformation step as discussed previously. In case of slower CPU hardware, this gain can be significantly higher.

Figure 11 shows that the when GPU-only ConvMM layer is combined with the proposed memory optimization scheme (Unified), there is a huge performance gain in terms of execution time.

Table 1. Comparison of different versions of ConvMM layers (Single convolutional layer), best results are written in bold.

Size of Input Image	Sequential ConvMM (CPU)	Hetero ConvMM (CPU + GPU)	GPU-Only ConvMM	GPU-Only Unified ConvMM
		Double		
		(Milliseconds)		
Cifar (32 × 32 × 3)	11.17	8.03	5.12	**2.75**
Imagenet (224 × 224 × 3)	348.67	89.39	35.6	**13.13**
VGA (480 × 640 × 3)	2048.63	375.02	215.7	**75.03**
SVGA (600 × 800 × 3)	3212.65	584.13	369.9	**117.33**

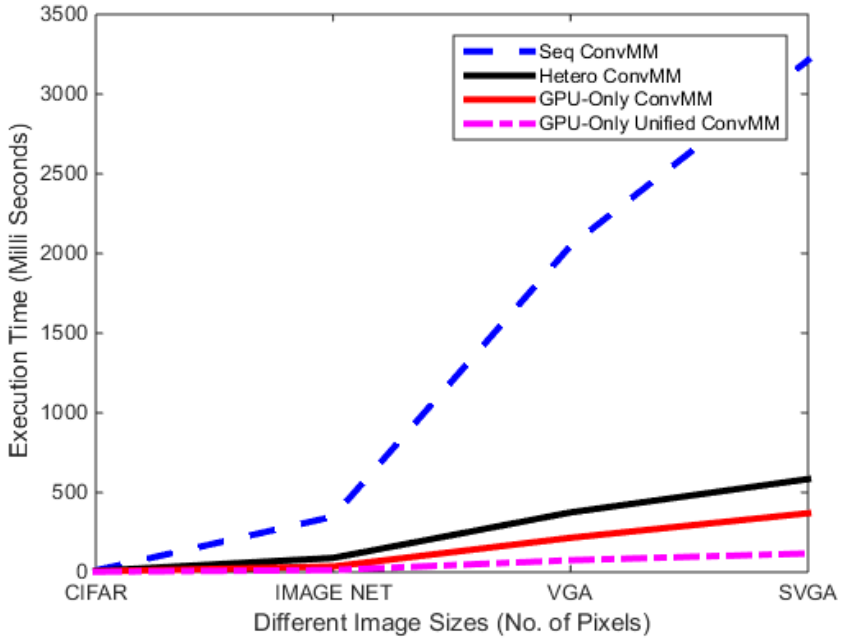

Figure 11. Comparison of different versions of ConvMM layers on Jetson TX1 Board.

This GPU-only unified memory scheme also outperforms the all other versions for the deeper networks as shown in Table 2. This performance gain is more evident here because of large data transfers required in the deeper networks.

Table 2. Performance of ConvMM layers on the deep classifiers (Jetson TX1 Board), best results are written in bold.

Model	Layers	Sequential ConvMM (CPU)	Hetero ConvMM (CPU + GPU)	GPU-Only ConvMM	GPU-Only Unified ConvMM
			Double (Milliseconds)		
Alex's CIFAR-10	5	7814.25	129.9	107.03	**49.76**
OverFeat	8	254,285.12	2492.94	1924.41	**704.27**
ResNet-34	34	190,054.39	2361.18	1217.33	**1092.23**

After this memory-optimized scheme, the cuBLAS library is used to accelerate the matrix multiplication. cuBLAS accelerated matrix multiplication (CAMM) is employed to compute the solution of matrix multiplication-based convolution algorithm. As the compute capability of Jetson TX1 board is greater than 3.5, cuBLAS library is best choice to access the computational resources of the Nvidia GPU. To use the cuBLAS API, the application must allocate the matrices in the device memory and as transformed input image and filters are already in the GPU memory, they can directly be used by the cuBLAS matrix multiplication subroutine as shown in Figure 9.

It can be noted that all the trainable parameters imported from the Torch software are of double data type. These parameters and complete CUDA-based implementations discussed previously are reformed to use the single data type. This is done to reduce the size of the trainable parameters and to take benefit from the Single-Precision General Matrix-Matrix Multiply (SGEMM) routine of the cuBLAS library.

Table 3 and Figure 12 depict the performance of this accelerated ConvCAMM layer. Results show that the cuBLAS-based ConvCAMM layer gains significant speedup over double and single precision GPU-only ConvMM layers.

Table 3. Comparison of cuBLAS-based ConvCAMM layer), best results are written in bold.

Size of Input Image	GPU-Only ConvMM	GPU-Only ConvMM	ConvCAMM
	Double	Single	
		(Milliseconds)	
Cifar ($32 \times 32 \times 3$)	5.12	3.2	**2.1**
Imagenet ($224 \times 224 \times 3$)	35.6	21.63	**14.59**
VGA ($480 \times 640 \times 3$)	215.7	119.17	**92.65**
SVGA ($600 \times 800 \times 3$)	369.9	169.8	**131.69**

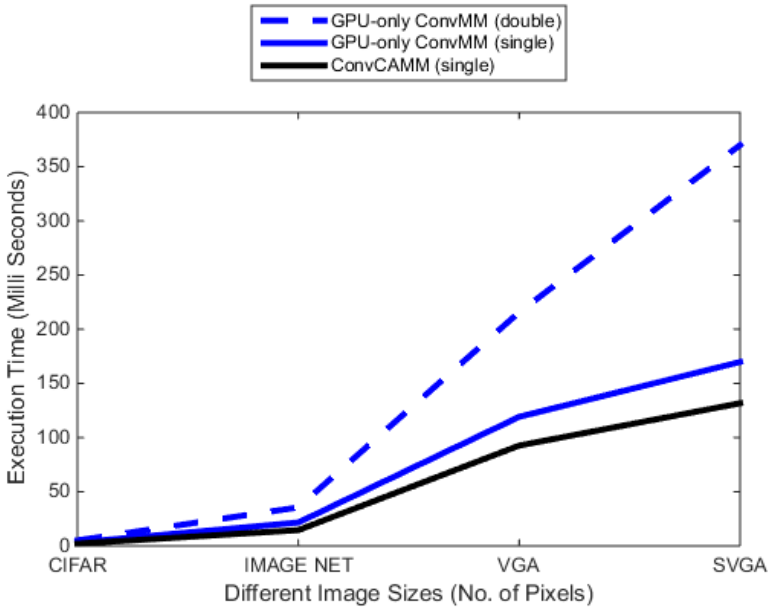

Figure 12. Comparison of cuBLAS-based ConvCAMM layer.

Table 4 list the performance of this single-precision ConvCAMM layer over three image classification architectures. Results show that the cuBLAS-based ConvCAMM layer is approximately 3× faster than the double precision GPU-only ConvMM layer.

Table 4. Performance of the ConvCAMM layers on the deep classifiers (Jetson TX1 Board), best results are written in bold.

Model	Layers	GPU-Only ConvMM	GPU-Only ConvMM	ConvCAMM
		Double	Single	
			(Milliseconds)	
Alex's CIFAR-10	5	107.03	76.36	**35.53**
OverFeat	8	1924.41	1212.7	**496.83**
ResNet-34	34	1217.33	887.53	**577.66**

Finally, the flow of this single-precision ConvCAMM layer is further optimized using unified memory. Results are summarized in Table 5. Results show that the Unified ConvCAMM layer is hundreds of time faster than the sequential version and gain tens of times speedup over the Heterogeneous version.

Table 5. Performance comparison of Unified ConvCAMM layer with other versions on Jetson TX1 embedded Board, best results are written in bold.

Model	Layers	Sequential ConvMM (CPU)	Hetero ConvMM (CPU + GPU)	Unified ConvCAMM	Speedup Over Sequential	Speedup Over Hetero
		Double		Single (Milliseconds)		
Alex's CIFAR-10	5	7814.25	129.9	**19.13**	408	7
OverFeat	8	254,285.12	2492.94	**122.9**	2069	20
ResNet-34	34	190,054.39	2361.18	**423.86**	448	6

6.2. Performance Evaluation of Proposed Scheme on Mobile Tablet (For GPUs Having C.C. < 3.5)

The second platform used for the performance evaluation of proposed scheme is Nvidia Shield K1 tablet. It has Kepler K1 GPU with 192 CUDA cores. Same experiments are performed to evaluate the proposed scheme on this second embedded GPU having compute capability less than 3.5.

First of all, the GPU-only ConvMM layer is compared with its sequential, heterogeneous and memory-optimized version and results are listed in Table 6 [15]. Figure 13 shows that the GPU-only ConvMM layer is 2× faster than the heterogeneous version where the transformation step is performed sequentially using the CPU and the multiplication is performed using the GPU. It can also be noted that for smaller workload sizes, Unified memory-based GPU-only ConvMM layer is slower because of its lengthy context initialization time while for the larger workloads, it outperformed the other versions.

Table 6. Comparison of different versions of ConvMM layers on a mobile device; best results are written in bold.

Size of Input Image	Sequential ConvMM (CPU)	Hetero ConvMM (CPU + GPU)	GPU-Only ConvMM	GPU-Only Unified ConvMM
			Single (Milliseconds)	
Cifar (32 × 32 × 3)	37.21	4.20	**3.95**	12.52
Imagenet (224 × 224 × 3)	1570.98	62.27	28.01	**26.59**
VGA (480 × 640 × 3)	10,055.24	327.53	154.32	**116.85**
SVGA (600 × 800 × 3)	15,852.86	519.08	231.95	**181.85**

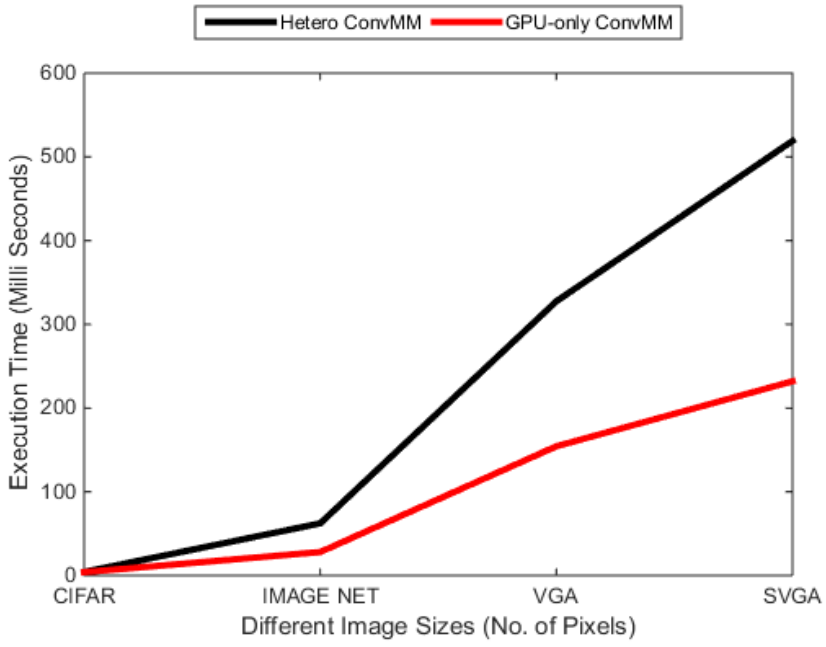

Figure 13. Comparison of hetero and GPU-only ConvMM layers on Nvidia shield K1 tablet.

As mentioned previously, the compute capability of Nvidia Shield K1 tablet is 3.2 and cuBLAS-based matrix multiplication routine cannot be used on this platform. Therefore, shared memory-based accelerated matrix multiplication (SAMM) is proposed for the devices having compute capability less than 3.2. This shared memory-based ConvSAMM layer is further optimized using the unified memory scheme. Results are listed in Table 7. Results show that the accelerated ConvSAMM layer is faster than the GPU-only ConvMM layer for each case of workload and it can be further accelerated using unified memory for larger workloads.

Table 7. Comparison of different versions of accelerated and optimized ConvMM layer, best results are written in bold.

Size of Input Image	GPU-Only ConvMM	ConvSAMM	Unified ConvSAMM
		Single	
		(Milliseconds)	
Cifar (32 × 32 × 3)	3.95	**3.20**	29.15
Imagenet (224 × 224 × 3)	28.01	**19.91**	34.75
VGA (480 × 640 × 3)	154.32	127.32	**62.98**
SVGA (600 × 800 × 3)	231.95	143.45	**102.69**

Tables 8 and 9 tabulate the performance of different versions of ConvMM layers over the same classification models. Results show that the proposed Unified ConvSAMM layer boosts the performance of deep classifiers on the mobile platform and outperforms the other versions for the same classification problems.

Table 8. Performance of ConvMM layers on deep classifiers (Nvidia Shield K1 tablet), best results are written in bold.

Model	Layers	Sequential ConvMM (CPU)	Hetero ConvMM (CPU + GPU)	GPU-Only ConvMM	ConvSAMM	Unified ConvSAMM
				Single		
				(Milliseconds)		
Alex's CIFAR-10	5	10,449.21	419.67	144.23	74.20	**38.57**
OverFeat	8	440,631.27	5792.54	3899.43	1784.24	**632.91**
ResNet-34	34	252,955.27	3319.34	2545.83	1210.16	**767.53**

By accelerating the matrix multiplication function and using optimized memory flow, proposed Unified ConvSAMM layer is 271×–696× faster than the sequential version and 4×–11× faster than the heterogeneous version.

Table 9. Performance gain achieved by proposed scheme on Nvidia Shield K1 tablet.

Model	Layers	Speedup Over Sequential	Speedup Over Hetero
Alex's CIFAR-10	5	271×	11×
OverFeat	8	696×	9×
ResNet-34	34	330×	4×

Furthermore, battery life is a critical performance metric of the mobile devices. Android phones are getting more powerful with time, and with that, energy consumption of them is gradually going up. Unacceptable energy consumption can reduce the life of the battery. For this reason, energy consumption is an important factor to consider when performing the real-time image classification using a mobile phone. The Nvidia Shield K1 tablet has a battery of 5192 mAh and 19.75 Wh. A software

profiler is used to measure the actual energy consumed by the android application through hardware. Results are listed in Table 10 and represent the energy consumption per image frame.

Table 10. Comparison of power consumption on Nvidia Shield K1 tablet (joule); best results are written in bold.

Model	Layers	Sequential ConvMM	Hetero ConvMM	Unified ConvSAMM	Efficiency Over Sequential	Efficiency Over Hetero
Alex's CIFAR-10	5	16	0.410	**0.202**	79×	2×
OverFeat	8	850.8	3.2	**0.529**	1608×	6×
ResNet-34	34	480	1.8	**0.432**	1111×	4×

Results show that the proposed scheme has a significant impact on the total amount of consumed energy and heterogeneous approach is consuming up to 50% more energy than the proposed accelerated flow.

6.3. Performance Comparison of Proposed Scheme with Torch Framework

All these experiments are performed on three classification networks (Alex's CIFAR-10, OverFeat and ResNet-34). These networks are trained in Torch framework and trainable parameters are imported to utilize by the proposed scheme. Table 11 presents a comparison of Torch-based networks and proposed scheme utilizing trained parameters from former networks. Jetson TX1 embedded board is used to perform this comparative analysis. Jetson TX1 board has Maxwell GPU with 256 CUDA cores and 4 GB of RAM. It provides 16 GB of embedded MultiMediaCard (eMMC) to store the operating system, required frameworks, packages, and files.

Performance of cuDNN torch library is evaluated for the same three classification networks. cuDNN is a wrapper of Nvidia's cuDNN library for optimized GPU implementations of Neural Networks.

It can be noted that for Alex's CIFAR-10 and OverFeat models, the proposed flow shows the comparable performance to the Torch-based library while the proposed flow is slower for ResNet-34 model having small filters. However, proposed flow can be accelerated for networks having small filters using Winograd's minimal filtering algorithms.

Table 11. Performance comparison of proposed flow with torch libraries (on Jetson TX1 embedded board).

Model	Input Image Size	Layers	Proposed Unified ConvCAMM Flow	Torch (cudNN)
			(Milliseconds)	
Alex's CIFAR-10	(1, 3, 32, 32)	5	19.13	37.11
OverFeat	(1, 3, 231, 231)	8	122.9	140.78
ResNet-34	(1, 3, 224, 224)	34	423.86	102.10

Because embedded devices have limited memory storage, so one constraint to consider is the size of the trained model and installation of required packages on an embedded platform. Table 12 lists the sizes of trained models imported from torch and converted trained parameters for proposed optimized models.

It can be noted that a complete trained model with installation of main framework (in this case torch, consuming 828 MB of memory on Jetson TX1 board) require a significant amount of memory storage. While just using trainable parameters, a comparable optimized scheme can be realized on an embedded platform.

Table 12. Size of Model/Parameters.

Model	Layers	Trainable Parameters (Float)	Torch
		Megabytes	
Alex's CIFAR-10	5	19	41
OverFeat	8	556	1190
ResNet-34	34	83	215

7. Conclusions

Exploitation of hardware capabilities plays a vital role to enhance the performance of an algorithm. In this paper, a set of optimization techniques is proposed to accelerate the heterogeneous convolutional layer to bridge the gap toward the real-time image classification on embedded platforms. The effectiveness of this proposed flow is evaluated on two different type of embedded GPUs (Jetson TX1 development board and Nvidia Shield K1 tablet). The experimental results indicate that the proposed flow can take benefits from the concurrent implementation of convolution algorithm, optimized data transfer scheme, and hardware dependent matrix multiplication approach for better exploitation of GPU resources. The performance results illustrate that the proposed optimized scheme can efficiently perform image classification in real time on embedded platforms with less memory requirements and is more power efficient than the heterogeneous and sequential versions.

This suggests promising future work towards the realization of useful real-time classification and recognition problems on the embedded platforms [27–29]. This framework can be further accelerated by exploiting the hardware capabilities and reducing the computational complexity of the convolution operation. Embedded devices having Pascal architecture GPUs support 16-bit floating point (FP16) data storage and arithmetic computations. By using FP16 data on an embedded platform like Jetson TX1 board, storage requirement and memory bandwidth can be further reduced for large neural network models. Furthermore, the complexity of convolution operation over small tiles can be minimized by using Winograd's minimal filtering algorithm. This can provide significant performance improvement for small filter and batch sizes as used by state-of-the-art deep architectures.

Acknowledgments: This research work is funded by Telecom Italia and performed at Joint Open Lab, Telecom Italia, Torino, Italy.

Author Contributions: Syed Tahir Hussain Rizvi conducted the experiments and worked on the draft of the paper. Gianpiero Cabodi and Gianluca Francini are the academic tutors. They coordinated, supervised and approved the entire work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rizvi, S.T.H.; Cabodi, G.; Francini, G. GPU-only Unified ConvMM Layer for Neural Classifiers. In Proceedings of the 2017 International Conference on Control, Decision and Information Technologies (CoDIT'17), Barcelona, Spain, 5–7 April 2017. in press.
2. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In Proceedings of the Twenty-sixth Annual Conference on Neural Information Processing Systems (NIPS), Lake Tahoe, NV, USA, 3–8 December 2012; pp. 1097–1105.
3. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.E.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going Deeper with Convolutions. In Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015; pp. 1–9. [CrossRef]
4. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778. [CrossRef]
5. Sermanet, P.; Eigen, D.; Zhang, X.; Mathieu, M.; Fergus, R.; LeCun, Y. OverFeat: Integrated Recognition, Localization and Detection Using Convolutional Networks. Computer Science—Computer Vision

and Pattern Recognition. 2013. Available online: https://arxiv.org/abs/1312.6229 (accessed on 12 November 2016).
6. Mathieu, M.; Henaff, M.; Lecun, Y. Fast Training of Convolutional Networks through FFTs. In International Conference on Learning Representations (ICLR2014), CBLS. 2014. Available online: http://arxiv.org/abs/1312.5851 (accessed on 10 November 2016).
7. Lavin, A; Gray, S. Fast Algorithms for Convolutional Neural Networks. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021. [CrossRef]
8. Chellapilla, K.; Puri, S.; Simard, P. High Performance Convolutional Neural Networks for Document Processing. In Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule, France, 23–26 October 2006.
9. Cong, J.; Xiao, B. Minimizing Computation in Convolutional Neural Networks. In Proceedings of the 24th International Conference on Artificial Neural Networks and Machine Learning—ICANN 2014, Hamburg, Germany, 15–19 September 2014; pp. 281–290.
10. Wang, J.; Lin, J.; Wang, Z. Efficient convolution architectures for convolutional neural network. In Proceedings of the 8th International Conference on Wireless Communications & Signal Processing (WCSP), Yangzhou, China, 13–15 October 2016; pp. 1–5. [CrossRef]
11. Jiang, W.; Chen, Y.; Jin, H.; Luo, B.; Chi, Y. A Novel Fast Approach for Convolutional Networks with Small Filters Based on GPU. In Proceedings of the 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), New York, NY, USA, 24–26 August 2015; pp. 278–283. [CrossRef]
12. Li, X.; Zhang, G.; Huang, H.H.; Wang, Z.; Zheng, W. Performance Analysis of GPU-Based Convolutional Neural Networks. In Proceedings of the 45th International Conference on Parallel Processing (ICPP), Philadelphia, PA, USA, 17–19 February 2016; pp. 67–76. [CrossRef]
13. Park, H.; Kim, D.; Ahn, J.; Yoo, S. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In Proceedings of the 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS), Pittsburgh, PA, USA, 1–7 October 2016; pp. 1–10.
14. Li, K.; Shi, H.; Hu, Q. Complex convolution Kernel for deep networks. In Proceedings of the 2016 8th International Conference on Wireless Communications & Signal Processing (WCSP), Yangzhou, China, 13–15 October 2016; pp. 1–5. [CrossRef]
15. Rizvi, S.T.H.; Cabodi, G.; Patti, D.; Francini, G. GPGPU Accelerated Deep Object Classification on a Heterogeneous Mobile Platform. *Electronics* **2016**, *5*, 88. [CrossRef]
16. Huynh, L.N.; Balan, R.K.; Lee, Y. DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices. In Proceedings of the 2016 Workshop on Wearable Systems and Applications, Singapore, 30 June 2016; pp. 25–30.
17. Oskouei, S.S.L.; Golestani, H.; Hashemi, M.; Ghiasi, S. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In Proceedings of the 2016 ACM on Multimedia Conference (MM'16). ACM, New York, NY, USA, 15–19 October 2016; pp. 1201–1205. [CrossRef]
18. Tsung, P.K.; Tsai, S.F.; Pai, A.; Lai, S.J.; Lu, C. High performance deep neural network on low cost mobile GPU. In Proceedings of the 2016 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 7–11 January 2016; pp. 69–70. [CrossRef]
19. Wu, J.; Leng, C.; Wang, Y.; Hu, Q.; Cheng, J. Quantized Convolutional Neural Networks for Mobile Devices. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 4820–4828. [CrossRef]
20. Kim, Y.-D.; Park, E.; Yoo, S.; Choi, T.; Yang, L.; Shin, D. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. CoRR, 2015. Available online: http://arxiv.org/abs/1511.06530 (accessed on 14 November 2016).
21. Naik, V.H.; Kusur, C.S. Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment. In Proceedings of the 2015 National Conference on Parallel Computing Technologies (PARCOMPTECH), Bangalore, India, 19–20 February 2015; pp. 1–5. [CrossRef]
22. Torti, E.; Danese, G.; Leporati, F.; Plaza, A. A Hybrid CPU–GPU Real-Time Hyperspectral Unmixing Chain. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2016**, *9*, 945–951. [CrossRef]

23. Barberis, A.; Danese, G.; Leporati, F.; Plaza, A.; Torti, E. Real-Time Implementation of the Vertex Component Analysis Algorithm on GPUs. *Geosci. Remote Sens. Lett.* **2013**, *10*, 251–255. [CrossRef]
24. Bernabé, S.; Sánchez, S.; Plaza, A.; López, S.; Benediktsson, J.A.; Sarmiento, R. Hyperspectral Unmixing on GPUs and Multi-Core Processors: A Comparison. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2013**, *6*, 1386–1398. [CrossRef]
25. Yang, Y.; Feng, M.; Chakradhar, S. HppCnn: A High-Performance, Portable Deep-Learning Library for GPGPUs. In Proceedings of the 2016 45th International Conference on Parallel Processing (ICPP), Philadelphia, PA, USA, 16–19 August 2016; pp. 582–587. [CrossRef]
26. Shi, Y.; Niranjan, U.N.; Anandkumar, A.; Cecka, C. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), Hyderabad, India, 19–22 December 2016; pp. 193–202. [CrossRef]
27. Huang, Y.; Wu, R.; Sun, Y.; Wang, W.; Ding, X. Vehicle Logo Recognition System Based on Convolutional Neural Networks with a Pretraining Strategy. *Trans. Intell. Transp. Syst.* **2015**, *16*, 1951–1960. [CrossRef]
28. Qian, Y.; Bi, M.; Tan, T.; Yu, K. Very Deep Convolutional Neural Networks for Noise Robust Speech Recognition. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2016**, *24*, 2263–2276. [CrossRef]
29. Christiansen, P.; Nielsen, L.N.; Steen, K.A.; Jørgensen, R.N.; Karstoft, H. DeepAnomaly: Combining Background Subtraction and Deep Learning for Detecting Obstacles and Anomalies in an Agricultural Field. *Sensors* **2016**, *16*, 1904. [CrossRef] [PubMed]