Real-time enhanced efficient thread level parallelism scheme for performance improvement in heterogeneous edge computing

K. Indra Gandhi | G. Kannan | P. K. Jawahar

*Department of Electronics & Communication Engineering, B. S. Abdur Rahman Crescent Institute of Science & Technology, Chennai – 600048, India.

Abstract In the era of technology, there is a need to rely on new high performance Heterogeneous embedded computing device to process a huge amount of data for various smart applications. Packing different architecture processor into a system on chip provides a substantial potential improvement in computing horsepower, but the maximum processing power of this heterogeneous edge computing processor can only be harnessed if the target software is actually configured to utilize all the processing elements. The proposed Enhanced Efficient thread level parallelism (EETLP) is implemented using CUDA in CPU-GPU based heterogeneous edge computing platform and analyzed with different size of matrix multiplication. From the experiment results, it was clearly observed that for the matrix size 1024x1024, Efficient Thread Level Parallelism (ETLP) using quad core CPU processor reduces 71% execution time and EETLP reduces 99% execution time compared to Basic Sequential Execution (BSE). In terms of Speedup, EETLP has achieved 5.5Kx speedup compare with ETLP and 19Kx speedup against BSE on CPU.

Keywords: parallel computing, high performance computing, GPU programming

1. Introduction

Conventional edge computing systems lack the computational power required to meet the performance demands of modern multithreaded applications (Zu, 2020). While heterogeneous high-performance edge computing systems offer improved efficiency, researchers widely agree that programming these systems is complex and not straightforward. Parallelizing legacy code and developing parallel programs from scratch pose significant challenges, often referred to as the programmability wall in the multi-core environment (Amaral, 2016). Leveraging the increasing computational power of edge systems necessitates multithreaded and multiprocess programming. Heterogeneous edge computing devices consist of multiple CPU cores, a few GPU cores, memory, and communication links for data sharing among processors (Ben Abdallah, 2017).

To ensure optimal utilization of heterogeneous computing systems, all processors within the system must execute portions of the application program based on thread weights (Songwen et al., 2018). Applications such as medical image processing and weather monitoring systems often involve large matrix multiplications, with a complexity of $O(N^3)$, where $N$ represents the dimension of the square matrix (Yegnanarayanan, 2013). Multicore edge computing refers to utilizing the combined processing power of multiple cores within edge computing devices. Its goal is to enhance computational capabilities and enable efficient data processing at the network edge. Through parallel processing across multiple cores, this approach allows for the effective management of various workloads, resulting in enhanced system performance within edge computing environments. Dealing with issues in parallel programs can be more complex than in sequential programs (Indragandhi et al., 2021). Debugging parallel code necessitates the use of specialized tools and techniques, and addressing problems such as race conditions, deadlocks, or load imbalances can be a time-consuming process of reproducing and diagnosing the underlying causes. A parallel computing model for online binary computation offloading in mobile edge computing was introduced by Acheampong et al. (2023). In the study, a wireless MEC network with a binary offloading strategy was proposed, allowing dynamic decisions between local computation and MEC server offloading by mobile devices. Traditional optimization challenges were overcome, and a parallel computing architecture was employed, achieving optimal performance and reducing computing time. To enhance the computing power in CPU-GPU based heterogeneous edge computing device for a specific application, an Enhanced Efficient Thread Level Parallelism (EETLP) is proposed. EETLP introduces a novel approach leveraging GPU parallelism using CUDA on the Jetson Nano platform for matrix multiplication and also showcasing the potential of parallel programming for enhancing matrix computation efficiency.
2. Literature Survey

A comprehensive exploration of parallel programming and multicore processors emerges from a series of studies across various domains. In their work, Hahn Kim et al. (2009) underscored the importance of directing attention toward algorithm parallelization in image processing, advocating for a focus on enhancing application efficiency rather than burdening programmers with intricate details of multicore architectures. Cleverson et al. (2013) evaluated the parallel programming models OpenMP, CUDA and OpenACC performance for three different applications namely Mandelbrot set, N-Queens and matrix multiplication. In this work, Ubuntu 12.04, AMD Athlon(tm) II X2 270 Processor was used for OpenMP and serial execution. The GPU Processor NVIDIA GeForce GTX 650 was used for CUDA and OpenACC. The execution time of each model was measured and it was concluded that Open ACC and CUDA provides good performance when it is compared to other models. Rathore et al. (2014) developed a parallel programming model to estimate the performance of matrix multiplication on single-core, dual-core, and multicore desktop machines using OpenMP. A speedup of 3.6 times was achieved with OpenMP in dual-core and multicore setups compared to single-core. The authors also suggested incorporating loop optimization techniques along with OpenMP to further improve the application's speed. Marungo et al. (2015) proposed an innovative method for parallelizing embedded applications. They extended the OpenMP programming model onto a heterogeneous multiprocessor system, achieving significant speedup through task scheduling between FPGA accelerators and CPU hosts using OpenCL and OpenMP runtime environments.

Mittal et al. (2015) conducted a comprehensive survey on heterogeneous computing using CPU-GPU architecture, examining performance at multiple levels, from algorithm design to application execution. Singh et al. (2017) contributed tools for measuring multicore architecture, CPU utilization, and performance, with a notable finding highlighting the positive correlation between increased parallelism and improved CPU core utilization. Kang et al. (2015) reviewed parallel computing models OpenMP, MPI, and Map-Reduce and discussed choosing the appropriate tool based on the application. A cluster of five PCs with Intel Core i7-4770 CPUs, a 3.40GHz clock, 16GB RAM, and CentOS-6.4 Linux distribution were used to solve data join problems and all-pairs-shortest-path. The paper concluded that the Map-Reduce model was efficient for computationally intensive applications. A hybrid technique utilizing OpenMP, MPI, and CUDA models on a multicore system was implemented by Chang et al. (2016). The conversion of sequential code to parallel code was achieved with the help of an auto parallelism tool, and the performance and various issues were analyzed. The experiments concluded that the combination of CUDA, OpenMP, and MPI was a powerful process for high-performance computing platforms. The evaluation was performed on an Intel Xeon E5410 CPU with a 33GHz clock, 4GB RAM, and a Tesla C1060 GPU. Goyal et al. (2017) delved into the realm of image segmentation, comparing various parallel programming models such as OpenACC, OpenMP, MPI, and hybrid OpenMP/MPI on a multicore processor. Their findings emphasized the superior speedup achieved by the OpenACC model. Bernabe et al. (2018) introduced a novel parallel model for hyperspectral image sensing, leveraging Intel Math Kernel Library and OpenMP on CPU-GPU architecture, showcasing enhanced performance compared to GPU-only processing. Zahran (2017) discussed issues in programming CPU, GPU, and FPGA architectures. Hard-ware and software challenges were elaborated in terms of memory, power, compiler, portability, scalability, and reliability. Monteria et al. (2018) shifted their focus to fault detection, successfully implementing Convolutional Neural Networks on a Raspberry Pi 3 for structural health monitoring applications. Cesariani et al. (2018) achieved fine-grained parallelism using OpenMP on the Kalray MPPA 256 edge device, resulting in a remarkable 12x speedup. Kwedlo et al. (2018) contributed novel algorithms for minimizing distance calculation in K-means clustering, employing a hybrid MPI and OpenMP approach that outperformed traditional algorithms. Pekturk et al. (2019) applied the OpenMP programming model to online remote sensing applications, yielding substantial improvements in both offline and online data processing. Collectively, these studies offer a nuanced understanding of parallel programming applications, showcasing advancements in image processing, fault detection, and clustering algorithms while providing valuable insights into the intricacies of multicore processor utilization.

Chander et al. (2019) implemented OpenMP parallel programming model for matrix multiplication and analyzed the performance for different matrix size and different number of threads in terms of speedup, efficiency based on execution time. In this paper, the parallel matrix multiplication was executed on Intel core i5-6500 CPU with 3.20 GHz clock with Windows 10 operating system. For estimation, maximum of 10 threads were considered. Parallel programming OpenMP was useful only when the scale of the input problem is considerably greater and for smaller data sequential method of execution itself provides a better solution. In edge computing environments, the presence of diverse devices with varying architectures, processing capabilities, and memory hierarchies poses a challenge for developing parallel programs. To effectively utilize the resources of these devices, careful optimization and adaptation for each specific platform are necessary.

In order to optimize the performance of specific applications on heterogeneous edge computing systems, a novel approach called Enhanced Efficient Thread Level Parallelism (EETLP) has been proposed. EETLP leverages the power of CUDA to effectively utilize both the CPU and GPU resources of edge computing devices. The methodology of Efficient Thread Level Parallelism (ETLP) was initially developed to maximize the utilization of each CPU core in a dedicated edge computing system (Indragandhi et al., 2020). Conversely, Basic Sequential Execution (BSE) refers to the conventional sequential execution solely
on the CPU core. To explore and compare various levels of parallelism, we have implemented the BSE, ETLP, and EETLP schemes on the Jetson NANO, which is a heterogeneous edge computing NVIDIA device.

3. Efficient Thread Level Parallelism (ETLP) scheme

The Efficient Thread Level Parallelism (ETLP) scheme is designed to maximize the processing capabilities of multicores processors through the implementation of a parallel programming model tailored for multithreaded applications (Indragandhi et al., 2020). Its main objective is to improve performance and optimize the utilization of CPU cores. To achieve this, the ETLP scheme makes use of OpenMP, a parallel programming model that encompasses a set of runtime library routines, pre-processor directives, and environment variables. By analyzing the characteristics of the multithreaded application, programmers can identify specific sections of the source code that would benefit from multithreading or remain single-threaded. This identification process enables the effective utilization of OpenMP pragmas, runtime library routines, and other relevant components. OpenMP’s user-friendly directives enable incremental parallelization without extensive code restructuring. Its portability across architectures aligns with the ETLP scheme’s goal of adapting to different systems.

4. OpenMP

Using the OpenMP parallel programming model, the ETLP scheme was achieved. The specification for the OpenMP API includes a framework for parallel programming that is scalable across multi-vendor for shared memory architectures (Yegnanarayanan, 2013). OpenMP API is provided by compilers from various vendors.

![Figure 1 Model of Fork and join in openMP.](https://www.malque.pub/ojs/index.php/msj)
A CUDA kernel refers to a function executed on the GPU, operating on an array of threads in parallel. Each thread independently performs computations, and although they execute the same code, they may take different paths based on the processed data. Each thread is assigned a unique ID, enabling access to specific input and output data. Control decisions within the kernel allow for conditional execution of code paths based on criteria or calculations. This flexibility enables efficient data processing and harnesses the GPU’s parallel processing capabilities.

From figure 3, Threads in CUDA programming are organized into blocks, and blocks are further organized into a grid. The kernel, representing the executed function, operates as a grid of blocks of threads on the GPU. Blocks consist of multiple threads working together to perform specific tasks, simplifying execution coordination and resource management. The grid structure provides a framework for executing the kernel across the GPU, facilitating efficient data parallelism and workload distribution. This arrangement allows the GPU to process vast amounts of data simultaneously, leveraging the parallel execution of multiple threads. By dividing computations into manageable units and organizing them within a structured grid, the GPU achieves high-performance parallel processing for complex computations.

Cooperation among threads and sharing results in CUDA programming is accomplished through memory accesses. Shared memory, accessible by all threads within a block, facilitates this cooperation. By confining communication to within a block, scalability is maintained, enabling efficient processing across numerous threads. However, it’s important to note that fast communication between a large number of threads is not feasible due to limitations on simultaneous effective
communication. The decision to use CUDA as the GPU programming model was motivated by its compatibility with NVIDIA GPUs, well-established ecosystem, and prowess in performance optimization. Its integration into heterogeneous CPU-GPU architectures ensures smooth data transfer and effective utilization of parallel processing capabilities. CUDA's adaptable parallel programming model, coupled with support for languages like C and C++, makes it an attractive choice for developers seeking to optimize GPU performance across a range of applications. The sequence of operations within a CUDA program involves the interaction between the host and the GPU kernel, as depicted in Figure 4.

6. Proposed Scheme: Enhanced Efficient Thread Level Parallelism (EETLP)

The proposed approach of Enhanced Efficient Thread Level Parallelism (EETLP) leverages CUDA in a CPU-GPU based heterogeneous embedded computing platform to enhance the performance of a specific application. EETLP introduces innovative features and advantages aimed at optimizing application performance on a heterogeneous embedded computing platform. By harnessing CUDA for GPU parallelism, EETLP stands out for its emphasis on enhancing software components, distinguishing itself in the research domain. It adeptly utilizes both CPU and GPU, showcasing pioneering developments in parallel programming tailored for embedded devices. The efficiency and scalability demonstrated in matrix multiplication experiments set EETLP apart, marking it as a substantial advancement. Its unique concentration on software aspects underscores its potential for achieving superior performance, particularly in environments with limited resources, making it a notable contribution to the realm of parallel computing on embedded systems.

The performance analysis involves conducting matrix multiplication experiments of varying sizes using three different techniques: Basic Sequential Execution (BSE), Efficient Thread Level Parallelism (ETLP), and the EETLP approach, which incorporates parallel computing on GPU architecture. While various research endeavors focus on improving matrix multiplication efficiency through hardware and software methods, this work primarily emphasizes the software aspect. To illustrate the workflow of the EETLP scheme, Figure 5 provides a visual representation.

7. Experimental Setup

In the experimental setup, the Jetson Nano NVIDIA heterogeneous edge computing platform is utilized, running on the Ubuntu 18.04 Linux operating system. This setup comprised a Jetson Nano NVIDIA board interfaced to a 7-inch IPS touch HDMI screen LCD display, as well as a Bluetooth mouse and keyboard as shown in figure 6. The performance evaluation of EETLP is conducted by performing matrix multiplication operations with varying dimensions. Matrix multiplication finds applications in several domains such as signal processing, graph theory, image processing, numerical algorithms, and digital control systems. Figure 7 illustrates the CPU architecture of the Jetson Nano embedded device. Understanding the CPU architecture is essential for implementing ETLP and EETLP. The Jetson Nano features a quad-core Cortex A57 processor, with 50 iterations of threads distributed among the cores. Additionally, it incorporates NVIDIA Maxwell architecture GPU with 128 CUDA cores. The GPU allows a maximum of 2048 threads per core.
Figure 6 Experimental setup of EETLP.

Figure 7 CPU architecture of Jetson Nano Board.

Figure 8 presents the pseudo code for matrix multiplication using BSE, ETLP, and EETLP. Case 1 demonstrates the matrix multiplication operation using the BSE method, which represents the traditional approach without any specific programming model on the embedded device. In Case 2, the code showcases the ETLP scheme, achieved through the utilization of the OpenMP programming model. Workloads are divided into chunks, which are allocated to threads in a contiguous order based on the available workload. The ETLP approach dynamically schedules 50 iterations to each thread during runtime, depending on core availability. Case 3 introduces the EETLP scheme for matrix multiplication. It begins by initializing the kernel function and allocating a unique thread ID for each iteration involved in matrix multiplication. Memory space is then allocated in the device memory of the GPU for input and output matrices. Finally, the kernel is launched, passing the parameters to the GPU. Each iteration of matrix multiplication is executed in parallel on the GPU, and the results are subsequently passed back to the host CPU. Finally, the space in the device memory is reallocated.

```

```

Figure 8 Pseudo codes of BSE, ETLP and EETLP for matrix multiplication.
The matrix multiplication operation is performed four times using the BSE, ETLP, and EETLP approaches, and the average execution time is used for evaluation. Table 1 displays the execution times for matrix multiplication of sizes 64x64, 128x128, 256x256, 512x512, and 1024x1024 for each approach.

The results from Table 1 emphasize the significant performance improvements achieved by both the ETLP and EETLP approaches compared to the traditional BSE approach for matrix multiplication tasks of various sizes.

When considering smaller matrix sizes like 64x64 and 128x128, ETLP shows notable execution time reductions of approximately 65% and 71% respectively, outperforming the BSE approach. However, the EETLP approach surpasses ETLP by achieving even greater performance improvements, with execution times reduced by approximately 95% and 99% for the respective matrix sizes.

As the dimensions of the matrices increase to 256x256 and 512x512, both the ETLP and EETLP approaches consistently outperform the BSE method. ETLP demonstrates impressive execution time reductions of approximately 74% and 75% respectively, highlighting its effectiveness. Notably, the EETLP approach exhibits even lower execution times compared to ETLP, surpassing it by a significant margin.

When considering the largest matrix size of 1024x1024, the superiority of both ETLP and EETLP becomes even more apparent. ETLP achieves a noteworthy reduction in execution time of around 71% compared to the traditional BSE approach, while EETLP surpasses them all, achieving the lowest execution time with a remarkable reduction of approximately 95% compared to the BSE method.

These findings highlight the consistent and superior performance of both ETLP and, particularly, the highly efficient EETLP approach in optimizing the execution time of matrix multiplication tasks. The results underscore the immense potential of these parallel programming approaches, particularly EETLP, to significantly enhance the efficiency and speed of matrix computations across a wide range of matrix sizes.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Execution Time (ms)</th>
<th>Iteration</th>
<th>BSE</th>
<th>ETLP</th>
<th>EETLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>64X64</td>
<td></td>
<td>1</td>
<td>6.877</td>
<td>2.168</td>
<td>0.288</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6.673</td>
<td>1.973</td>
<td>0.265</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>6.902</td>
<td>2.071</td>
<td>0.428</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>6.739</td>
<td>3.109</td>
<td>0.266</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>6.798</td>
<td>2.33</td>
<td>0.312</td>
</tr>
<tr>
<td>128X128</td>
<td></td>
<td>1</td>
<td>52</td>
<td>15.313</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>52.2</td>
<td>14.675</td>
<td>0.382</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>53.03</td>
<td>15.783</td>
<td>0.319</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>52.207</td>
<td>14.313</td>
<td>0.318</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>52.359</td>
<td>15.021</td>
<td>0.345</td>
</tr>
<tr>
<td>256X256</td>
<td></td>
<td>1</td>
<td>399.58</td>
<td>105.015</td>
<td>0.552</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>400.895</td>
<td>104.398</td>
<td>0.386</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>401.25</td>
<td>103.563</td>
<td>0.638</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>399.49</td>
<td>103.264</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>400.304</td>
<td>104.06</td>
<td>0.519</td>
</tr>
<tr>
<td>512X512</td>
<td></td>
<td>1</td>
<td>3268.466</td>
<td>819.461</td>
<td>1.445</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3276.315</td>
<td>820.61</td>
<td>1.666</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3278.975</td>
<td>824.317</td>
<td>1.241</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3290.629</td>
<td>813.415</td>
<td>0.959</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>3278.596</td>
<td>819.451</td>
<td>1.328</td>
</tr>
<tr>
<td>1024X1024</td>
<td></td>
<td>1</td>
<td>53086.5</td>
<td>15277.332</td>
<td>3.414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>52684.812</td>
<td>15084.158</td>
<td>2.732</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>52826.054</td>
<td>15079.827</td>
<td>2.618</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>52599.992</td>
<td>15513.262</td>
<td>2.295</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>52799.34</td>
<td>15288.645</td>
<td>2.765</td>
</tr>
</tbody>
</table>

Figure 9 presents the comparison of execution times across different approaches for matrix multiplication, with the y-axis representing the execution time and the x-axis representing the approaches. In Figure 9a, it can be observed that for a matrix size of 64x64, BSE takes 6.798ms, ETLP takes 2.33ms, and EETLP takes 0.312ms. Compared to traditional execution with BSE, ETLP reduces the execution time by 65%, while EETLP reduces it by 95%. Similarly, in Figure 9e, for a matrix size of 1024x1024, BSE takes 52799.34ms, ETLP takes 15288.645ms, and EETLP takes only 2.765ms. This demonstrates that EETLP offers significantly improved performance for larger volumes of data.
Figure 9 Comparison of execution time in BSE, ETLP and EETLP schemes.

Figure 10a showcases the speedup achieved by EETLP and ETLP compared to BSE. It can be observed that the speedup of ETLP is relatively low compared to EETLP on the GPU. EETLP achieves a remarkable speedup of 19K for the matrix size of 1024x1024 compared to 64x64. Figure 10b illustrates the speedup of EETLP on the GPU in comparison to ETLP on the CPU. It reveals that EETLP achieves a speedup of 5511.26 for the matrix size of 1024x1024 compared to ETLP on the CPU. The significance of Enhanced Efficient Thread Level Parallelism (EETLP) surpasses that of Efficient Thread Level Parallelism (ETLP) and Basic Sequential Execution (BSE) due to its notable performance enhancements, especially when dealing with larger matrices. EETLP, utilizing CUDA in a CPU-GPU heterogeneous embedded computing platform, achieves substantial reductions in execution times compared to both ETLP and BSE. Its ability to efficiently scale and handle more extensive datasets becomes apparent. In contrast to ETLP, EETLP demonstrates superior performance, achieving a significant speedup of
5511.26 for a matrix size of 1024x1024 compared to ETLP on the CPU. The consistent and superior performance of EETLP across varying matrix sizes highlights its effectiveness in managing diverse computational workloads. In summary, EETLP’s capacity to optimize matrix computations positions it as a valuable solution for achieving enhanced performance on heterogeneous embedded computing platforms. The speedup values highlight Enhanced Efficient Thread Level Parallelism’s (EETLP) potential impact in domains like signal processing, image processing, numerical algorithms, and digital control systems. EETLP’s ability to achieve substantial speedups offers efficient computations for applications in artificial intelligence, deep learning, and data science. In essence, these values validate EETLP’s effectiveness in optimizing matrix multiplication, emphasizing its practical relevance for real-world scenarios with demanding computational requirements. This positions EETLP as a versatile tool capable of significantly enhancing computational efficiency across diverse domains.

8. Conclusions

The prevalent adoption of heterogeneous edge computing platforms reflects a strategic response to the specialized requirements of modern applications. The synergy between CPU and GPU processing power in these platforms has significantly advanced performance capabilities. The specific evaluation of three approaches—BSE, ETLP, and EETLP—on the Jetson NANO device, with a focus on matrix multiplication tasks, has demonstrated the remarkable effectiveness of the EETLP scheme. This approach not only outperforms its counterparts in reducing execution time but also excels in handling...
large datasets, a crucial factor for applications in AI, deep learning, and data science. The comprehensive experimental investigations provide clear evidence that EETLP is the optimal choice for independently and efficiently executing numerous lightweight threads. Looking ahead, addressing challenges related to scalability, adaptability across computing environments, real-time applications, and energy efficiency will be pivotal for ensuring the sustained significance and broader applicability of EETLP in the dynamic landscape of computational research.

**Ethical considerations**

Not applicable.

**Conflict of Interest**

Authors declare that there are no conflicts of interest/competing among themselves.

**Funding**

This research did not receive any financial support.

**References**


