Performance comparison of CPU and GPGPU calculations using three simple case studies

Branislav Lipovský, Slavomír Šimoňák

Abstract

In this work, we have prepared and analyzed three case studies comparing CPU and GPGPU calculations. After briefly introducing the topic of parallel programming by means of contemporary CPU and GPGPU technologies, we provide an overview of selected existing works closely related to the topic of the paper. For each of the case studies, a set of programs has been implemented using the following technologies: pure CPU, CPU SIMD, CPU multi-threaded, CPU multi-threaded with SIMD instructions, and GPU - Cuda. We also illustrate the basic idea of the operation of selected algorithms using code snippets. Subsequently, the particular implementations are compared, and obtained results are evaluated and discussed.

Keywords: CUDA, Multi-threading, SIMD, Matrix multiplication, Sobel operator, Template matching.

MSC 2020: 68W10, 68U10, 94A08.

ACM CCS 2020: Computing methodologies – Parallel computing methodologies – Parallel algorithms – Vector / streaming algorithms, Computing methodologies – Computer graphics – Image manipulation.

1 Introduction

In today's world, we strive to integrate modern computer systems into our lives. This is done simply because of the general desire for modernization, to increase comfort, to speed up various processes, or for automation. All these processes need ever-increasing computing power.

doi:10.56415/csjm.v31.07

When faced with such new tasks, people may not know which technologies to use, which ones will be the easiest to use, which ones will be the most appropriate, and which ones they should avoid [1].

Moore's Law tells us that the number of transistors in an integrated circuit doubles every 2 years.

Huang's law, as observed by Jensen Huang, CEO of Nvidia, says that graphics card performance more than doubles every 2 years. With the ever-increasing performance of graphics cards, people often ask about CPU usage cases, SIMD CPU cases, or multi-core calculations versus GPU calculations.

A common practice, we can observe presently quite often, is using the GPU implementations for increasing the effectiveness of solutions to a wide range of computational problems [2]–[4]. However, contemporary CPUs also offer solid options for efficient execution of parallel programs [5], [6]. In this work, we would like to explore several practical problems in order to find if parallel CPU implementation could be more efficient than the GPU implementation.

2 Parallel programming

Parallel computing is a type of calculation in which many calculations or processes are performed at the same time. Large problems can often be divided into smaller ones, which can then be solved simultaneously [7].

The main point of parallel programming is the use of concurrency. Concurrency exists in a computational problem if the problem can be decomposed into subproblems that can be safely run at the same time. In order to take advantage of concurrency, it must be possible to structure the code itself so that the problems run at the same time. Most of the major computational problems involve usable concurrency. Parallel programming presents unique challenges:

- Often, concurrent subproblems can contain various dependencies that will need to be identified and managed properly.
- The order of execution of subproblems can change the results of calculations in non-deterministic ways.

- It is necessary that the cost of concurrency control does not exceed the cost of running the program itself.
- Balancing the work between computing units may not be easy.
- Parallel algorithm that is very efficient on one platform may not be as effective on other platforms [8].

Parallel programming on the CPU can leverage the following methods.

2.1 SIMD Parallelism

Single instruction multiple data (SIMD) allows a single instruction to operate on multiple data – perform multiple calculations at once [6]. The most widely known form of SIMD parallelism is vector instructions[1].

2.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) allows an operating system to see a single processor core as a set of logical processors. The operating system can then schedule threads on these logical processors which will every cycle compete for the functional units of a processor core [6].

2.3 Multicore Processors

Multicore processors integrate multiple execution units into a single processor chip. The operating system can see multiple logical processors, all of which have their own execution units [6].

3 General purpose computing on graphics processing units

General purpose computing on graphics processing units (GPGPU) is a method for performing large-scale calculations using a graphics

[1] https://cvw.cac.cornell.edu/vector/overview_simd

processor. Nowadays, we can also use GPGPU to perform, in addition to generic calculations, training of various machine learning models, image / video manipulation, cryptography, and even emulation [9]. For a program to be suitable for GPGPU it needs to meet the following criteria:

- parallelism – the ability to process multiple data at once
- throughput – the ability to process large amounts of data [10].

In order to utilize GPGPU, we need to use a GPGPU programming framework. The most popular GPGPU programming framework is Cuda.

Cuda is a platform developed by NVIDIA for utilizing graphics cards for non-graphic computations. Cuda C++ is a variant of C++ language that allows users to create and execute code on the GPU[2][3].

4 Related work

In this section, we summarize the results of some of the already existing comparisons. These comparison works also had an impact on the selection of problems we decided to use in our work.

4.1 Performance comparison of FPGA, GPU and CPU in image processing

In 2009, S. Asano, T. Maruyama, and Y. Yamaguchi compared the performance of FPGA, GPU, and CPU in various image processing tasks, specifically two-dimensional filters, stereo-vision, and k-means clustering. In their work, they found that GPU can match the performance of FPGA, however only when naive computation methods are used, otherwise the GPU may even be slower than CPU. In their tests, the CPU was reaching 1/12 – 1/7 the performance of FPGA [11].

[2]https://developer.nvidia.com/cuda-zone

[3]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

4.2 Comparative performance of GPU, SIMD and OpenMP systems for raw template matching in computer vision

In 2011, J. Méndez, J. Lorenzo-Navarro, and M. Castrillón Santana compared the performance of CPU and GPU in raw template matching. In their research, they found the CPU to achieve better performance compared to the GPU. They, however, also note that newer GPU architectures could outperform the CPU computations given bigger mask sizes [12].

4.3 Comparing CPU and GPU Implementations of a Simple Matrix Multiplication Algorithm

In 2017, T. Dobravec and P. Bulić compared the performance between CPU and GPU in matrix multiplication. In their testing, the GPU outperformed CPU in all cases. It needs to be noted, however, that their testing hardware utilized integrated GPU with shared memory, and, as a result of that data transfer between the GPU and CPU, had minimal to no impact on performance [13].

4.4 Implementation of Sobel filter using CUDA

In 2021, A. Akasapu, V. Sailaja, and G. Prasad implemented the sobel filter using Cuda GPGPU programming framework. They found that, for smaller images, the CPU gives faster results, and, for larger scale and high-resolution data sets, it is better to use the GPU [4].

5 Testing environment

The testing system used was a gaming laptop Lenovo Legion 5 (2021), with the following configuration:

- Processor: Ryzen 7 5800H
- RAM: 64GB 3200Mhz CL20 Dual-Channel Dual Rank
- GPU: NVIDIA GeForce RTX 3070 Laptop 8GB VRAM

The testing system was using the following software:

- Operating system: Windows 10 Education
- Development environment: Visual studio 2019
- GPGPU programming framework: CUDA Toolkit 11.6

6 Selected problems for comparisons

6.1 Matrix multiplication

Matrix multiplication is one of the most important matrix operations. It is often used in network theory, solving linear equation systems, coordinate system transformations, and population modeling. When multiplying matrices, the number of rows in the first matrix must be the same as the number of columns in the second matrix. When calculating a certain number of the resulting matrix, we calculate the scalar product from the row of the first matrix and the columns of the second matrix in which the given number is located [14].

6.2 Raw template matching

Template matching is a method of finding and locating a template image in a larger image. Raw template matching is the simplest form of template matching in which we try to find an exact match of template in the larger image.

6.3 Edge detection

Edge detection is a common step in multiple image processing algorithms. In this work, we will be using the sobel operator[4] for edge detection.

[4]https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

7 Implementation

For each of algorithms chosen for comparison, a set of programs has been implemented:

- basic example on CPU
- CPU + SIMD
- multiple threads CPU
- multiple threads CPU + SIMD
- GPU - cuda

7.1 Matrix multiplication

For matrix multiplication, we have chosen a simplified algorithm that has hardcoded matrix sizes and matrix width and height of the same size.

In the actual implementation, we first swap columns of the second matrix with its rows and then we calculate dot product of the two matrixes. We verified the calculated results from our implementations against simple reference implementation.

```
void CPU_matrix_mult(float* h_a, float* h_b,
                     float* h_result) {
  for (int i = 0; i < MATRIX_SIZE; ++i)
  {
        for (int j = 0; j < MATRIX_SIZE; ++j)
        {
          float tmp = 0.0f;
          for (int h = 0; h < MATRIX_SIZE; ++h)
          {
                tmp += h_a[i * MATRIX_SIZE + h] *
                    h_b[h * MATRIX_SIZE + j];
          }
          h_result[i * MATRIX_SIZE + j] = tmp;
        }
```

```
    }
}
```

At first, in our implementation, we used C++ std::vector to store the matrixes; however, because of performance and no native CUDA support, we changed to C-style arrays. Auto-vectorization worked with our compiler (MSVC) and used a similar approach to our initial SIMD implementation:

```
float dotProduct(float* arr1, float* arr2) {
  float ret;
  __m256 sum = _mm256_setzero_ps();
  for (int i = 0; i < MATRIX_SIZE; i += 8) {
    __m256 vec_arr1 = _mm256_loadu_ps(&arr1[i]);
    __m256 vec_arr2 = _mm256_loadu_ps(&arr2[i]);
    sum = _mm256_add_ps(sum,
      _mm256_mul_ps(vec_arr1, vec_arr2)
    );
  }
  float buffer[8];
  _mm256_storeu_ps(buffer, sum);
  ret = buffer[0] + ... + buffer[7];
  return ret;
}
```

We further optimized this implementation by computing two sets of vectors instead of one.

In our GPU program, we used tiling for efficient matrix multiplication on GPU[5].

As we can see in the graph (Fig. 1), with high matrix sizes – starting at 2048*2048 – our GPU program is faster. On the other hand, in lower matrix sizes, our CPU program is faster; in fact, at matrix sizes lower than 1024*1024, even the pure CPU program is faster than the GPU program.

[5]https://penny-xu.github.io/blog/tiled-matrix-multiplication

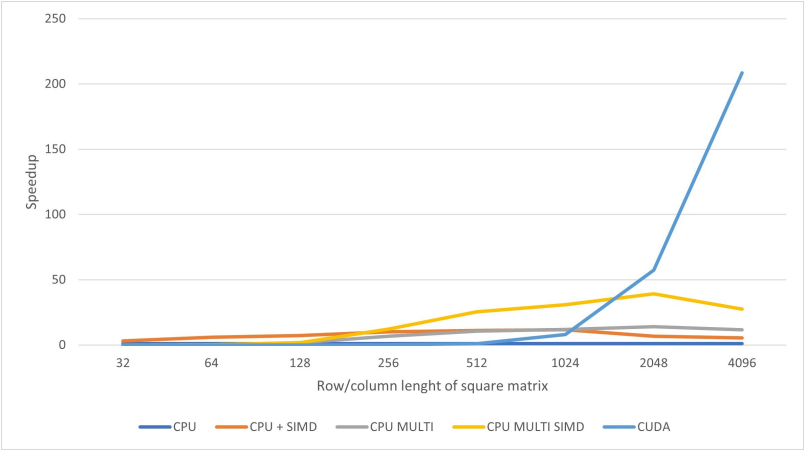

Figure 1. Graph of speedups achieved by our implementations of matrix multiplication.

7.2 Raw template matching

In our program, we load two images – the template and the image to search for the template. We loaded the images using bitmap library[6] and we also use that library to convert input images to grayscale.

In the implementation, we first iterate through every group of pixels that the template can fit in and then we compare the pixels in the given group with pixels from the template:

```
int compareTemplate(Image img, Image templ,
                    int posx, int posy) {
  for (int j = 0; j < templ.height; j++) {
    for (int i = 0; i < templ.width; i++) {
      if (img.data[(posy+j)*img.width+posx+i] !=
          templ.data[j * templ.width + i]) {
        return 0;
      }
    }
```

[6]https://github.com/wernsey/bitmap

```
    }
    return 1;
}
```

Our compiler (MSVC) could not auto-vectorize this algorithm. In our SIMD implementation, we created a function to compare two vectors, and we used this function to compare multiple pixels at once:

```
bool vec_equal(__m256i a, __m256i b) {
    __m256i pcmp = _mm256_cmpeq_epi32(a, b);
    unsigned bitmask = _mm256_movemask_epi8(pcmp);
    return (bitmask == 0xffffffffU);
}
```

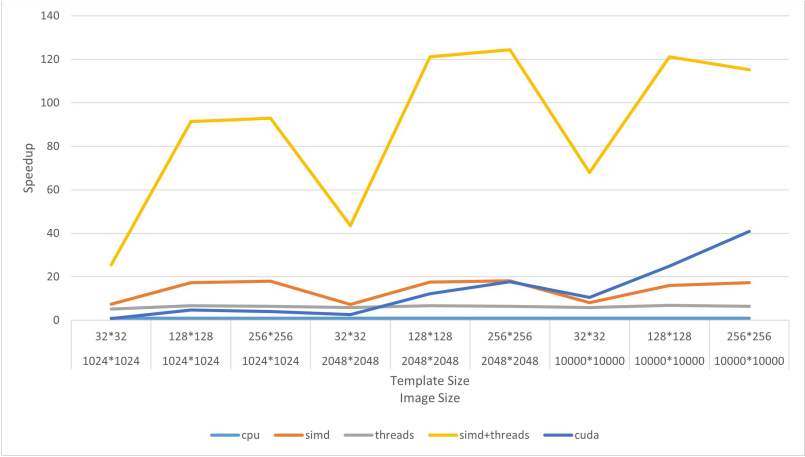

Figure 2. Graph of speedups achieved by our implementations of raw template matching.

From the graph in Fig. 2, we can make multiple observations:

- multi-threaded program is providing stable 6-7 times speedup compared to a single core;
- SIMD speedup is dependent on mask size: on 32*32 mask, there is 7-8 times speedup, on 128*128 and 256*256 – there is 16-18

times speedup;

- multithreaded program with SIMD is achieving up to 124 times speedup;
- GPU program, even though it is faster than CPU only, is reaching only slightly better performance and starts to be faster than SIMD only program at image size of 10000*10000.

7.3 Edge detection

Our program loads either one image in bmp format or a set of images in gif format using the bitmap library. The Sobel operator works by sliding two 3*3 convolution kernels over the input image and calculating gradients. Parts of the convolution kernels involve calculation which multiplies by zero, so we decided to unroll the calculations and skip the mentioned multiplies by zero:

```
sumX-=img[frame].data[x-1+(y-1)*img[frame].width];
sumX-=2*img[frame].data[x-1+(y)*img[frame].width];
sumX-=img[frame].data[x-1+(y+1)*img[frame].width];
sumX+=img[frame].data[x+1+(y-1)*img[frame].width];
sumX+=2*img[frame].data[x+1 +(y)*img[frame].width];
sumX+=img[frame].data[x+1 +(y+1)*img[frame].width];
```

Our compiler (MSVC) could not auto-vectorize this algorithm. In our SIMD implementation, we at first load 16 uint8_t numbers into ___m128i vector, zero extend that vector to 256 bits, add/subtract the numbers, and then we used saturation to convert int16_t numbers to uint8_t. Example code of loading and adding/subtracting a vector of numbers:

```
vec_sumX=_mm256_sub_epi16(
        vec_sumX,_mm256_cvtepu8_epi16(
            _mm_loadu_epi8(
                &img[frame].data[x-1 +(y-1)*width]
            )
        )
);
```

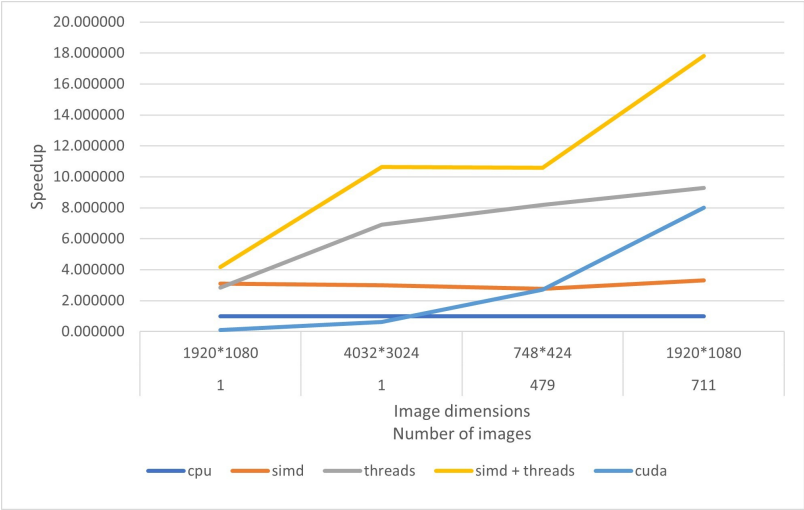

Figure 3. Graph of speedups achieved by our implementations of sobel algorithm.

In the graph in Fig. 3, we can see that our multithreaded program with SIMD instructions is always achieving greater speedup compared to our GPU program. Another result we can observe is that with single images the GPU program is slower than CPU program and only starts to achieve better performance when processing multiple images.

8 Evaluation

If we compare our matrix multiplication results with the results from T. Dobravec and P. Bulić [13], we can observe that in their case the calculation on the GPU was always faster. However, this comparison is not entirely appropriate, as they performed calculations on the NVIDIA GeForce 9400M graphics card that uses shared memory, so data transfer between CPU and GPU was virtually instantaneous. Furthermore, we can compare the results of work from K. Fatahalian, J. Sugerman, and P. Hanrahan [15], where they achieved results in which it was always

more appropriate to use a CPU when multiplying matrices. However, this work is relatively old (2004), and since then there have been many significant architectural changes in graphics cards.

We can observe that our template matching results agree with the conclusions of the work by J. Méndez, J. Lorenzo-Navarro, and M. Castrillón Santana [12] that the CPU achieves better results with small template dimensions, and only with large template/input dimensions will the GPU achieve better results. These results could be in part caused by the need to use synchronization in our GPU implementation, which causes performance penalty. Another factor could be using uint8_t data format for storing pixel values, which is not the preferred workload for GPUs.

Our edge detection results are consistent with the conclusions of work by A. Akasapu, V. Sailaja, and G. Prasad [4], where they claim that small images are achieving better performance on CPU, and for large images in large data sets it is better to use GPU. In this case, even though we did not use synchronization in our GPU implementation, we still used uint8_t data format for storing pixel values, which made this algorithm not perform as well on GPU as it did on CPU.

9 Conclusion

In this work, we briefly introduced the topic of parallel programming by means of contemporary CPU and GPGPU technologies. After this brief introduction we explored some of the existing works comparing CPU and GPU calculations. Based on our findings, we prepared three case studies and created programs for each type of parallelism. From the results obtained, we found that even in the cases that should be perfect for the GPU, like matrix multiplication, with small enough matrix sizes, calculations on the CPU could be more suitable. On the other hand, we found that for some image processing tasks, like edge detection and template matching, our GPU programs could not achieve better results than our CPU multithreaded programs with SIMD instructions despite both programs processing a relatively large number of pixels.

So we can conclude, that currently there certainly is a class of problems for which a parallel CPU implementation would be more efficient

than the GPU implementation. According to the results achieved in this work, we can formulate the following recommendations:

- Matrix multiplication: for matrices with dimensions up to 1024 x 1024, parallel CPU implementations were faster. However, for matrices of higher sizes, the GPU implementation is recommended, as it can achieve great speedup improvements over the CPU implementations.
- Raw template matching: SIMD multithreaded CPU implementation is recommended, as it achieved the best performance in all tested scenarios.
- Edge detection: SIMD multithreaded CPU implementation is recommended, as it achieved the best performance in all tested scenarios.

As we only performed our comparisons using three simple case studies, in the future it would be beneficial to include more diverse algorithms and create comparisons using bigger datasets. It would be also interesting to study the impact of the operating system and the compiler used on such comparisons. The additional research could help to better describe the class of problems for which CPU parallel implementation would be a better option. Another possibility to explore may be a combination of CPU and GPU parallel techniques in order to gain further efficiency improvements [16].

References

[1] B. Lipovský, "Comparison of use cases of simd cpu and gpgpu calculations," Master's thesis, Technical University of Košice, 2022, (in Slovak).

[2] J. R. Cheng and M. Gen, "Parallel Genetic Algorithms with GPU Computing," in *Industry 4.0*, T. Bányai, A. Petrillo, and F. De Felice, Eds. Rijeka: IntechOpen, 2020, ch. 6. [Online]. Available: https://doi.org/10.5772/intechopen.89152

[3] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012.

[4] A. Akasapu, V. Sailaja, and G. Prasad, "Implementation of sobel filter using CUDA," *IOP Conference Series: Materials Science and Engineering*, vol. 1045, no. 1, p. 012016, Feb. 2021. [Online]. Available: https://doi.org/10.1088/1757-899x/1045/1/012016

[5] T. Singh, D. K. Srivastava, and A. Aggarwal, "A novel approach for CPU utilization on a multicore paradigm using parallel quicksort," in *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*, 2017, pp. 1–6.

[6] T. Rauber and G. Rünger, *Parallel programming*. Springer, 2013.

[7] G. S. Almasi, *Highly Parallel Processing*. Reading, PA: Benjamin-Cummings Publishing Co., Subs. of Addison Wesley Longman, Nov. 1987.

[8] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Pearson Education, 2004.

[9] P. Jakubčo and S. Šimoňák, "Utilizing gpgpu in computer emulation," *Journal of Information and Organizational Sciences*, vol. 36, no. 1, pp. 39–53, 2012.

[10] T. A. C. Center, "8 things you should know about gpgpu technology," 2017. [Online]. Available: https://web.archive.org/web/20211126072222/https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf

[11] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 126–131.

[12] J. Méndez, J. Lorenzo-Navarro, and M. Castrillón Santana, "Comparative performance of gpu, simd and openmp systems for raw template matching in computer vision," pp. 9–15, 01 2011.

[13] T. Dobravec and P. Bulić, "Comparing cpu and gpu implementations of a simple matrix multiplication algorithm," *Interna-*

tional Journal of Computer and Electrical Engineering, vol. 9, pp. 430–438, 01 2017.

[14] B. D. Hahn and D. T. Valentine, "Chapter 6 - matrices and arrays," in *Essential MATLAB for Engineers and Scientists (Seventh Edition)*, seventh edition ed., B. D. Hahn and D. T. Valentine, Eds. Academic Press, 2019, pp. 127–161. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780081029978000129

[15] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 133–137. [Online]. Available: https://doi.org/10.1145/1058129.1058148

[16] V. Skorych and M. Dosta, "Parallel CPU–GPU computing technique for discrete element method," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 11, p. e6839, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6839

Branislav Lipovský[1], Slavomír Šimoňák[2]

Received July 01, 2022
Revised September 20, 2022
Accepted September 23, 2022

[1,2] Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovak Republic

[1] ORCID: https://orcid.org/0000-0001-7079-7519
E–mail: branislavlipovsky98@gmail.com

[2] ORCID: https://orcid.org/0000-0001-6505-3160
E–mail: slavomir.simonak@tuke.sk