PARALLEL NUMERICAL COMPUTATION: A COMPARATIVE STUDY ON CPU-GPU PERFORMANCE IN PI DIGITS COMPUTATION

Yozef Tjandra[1]; Sanga Lawalata[2]

[1,2] IT and Big Data Analytics Study Program
Calvin Institute of Technology
www.calvin.ac.id
[1*)] yozef.tjandra@calvin.ac.id, [2] sanga.lawalata@calvin.ac.id
(*) Corresponding Author

Abstract—As the usage of GPU (Graphical Processing Unit) for non-graphical computation is rising, one important area is to study how the device helps improve numerical calculations. In this work, we present a time performance comparison between purely CPU (serial) and GPU-assisted (parallel) programs in numerical computation. Specifically, we design and implement the calculation of the hexadecimal n-digit of the irrational number Pi in two ways: serial and parallel. Both programs are based upon the BBP formula for Pi in the form of infinite series identity. We then provide a detailed time performance analysis of both programs based on the magnitude n. Our result shows that the GPU-assisted parallel algorithm ran a hundred times faster than the serial algorithm. To be more precise, we offer that as the value n grows, the ratio between the execution time of the serial and parallel algorithms also increases. Moreover, when n it is large enough, that is $n \geq 3 \times 10^6$This GPU efficiency ratio converges to a constant 105.53, showing the GPU's maximally utilized capacity. On the other hand, for sufficiently small enough n, the serial algorithm performed solely on the CPU works faster since the GPU's small usage of parallelism does not help much compared to the arithmetic complexity.

Keywords: GPU, parallel computing, BBP formula, CPU-GPU comparison, parallel numerical method.

Abstrak—*Seiring bertumbuhnya penggunaan GPU (Graphical Processing Unit) untuk komputasi non-grafis, salah satu wilayah kajian yang penting adalah bagaimana piranti tersebut mampu meningkatkan perhitungan numerik. Dalam artikel ini, akan dibahas perbandingan kinerja waktu antara dua program komputer yang murni menggunakan CPU (seri) dan yang ditingkatkan oleh GPU (paralel), untuk melakukan perhitungan numerik. Secara spesifik, penelitian ini memberikan perancangan dan implementasi dari komputasi digit heksadesimal ke-n dari bilangan irasional Pi dalam dua cara: seri dan paralel. Kedua program berbasiskan pada rumus BBP untuk Pi dalam bentuk identitas deret tak hingga. Artikel ini kemudian akan menampilkan analisis mendetail mengenai kinerja waktu kedua program berdasarkan tingkat besarnya nilai* ***n****. Hasil penelitian menunjukkan bahwa algoritma paralel yang dioptimalkan oleh GPU berhasil bekerja ratusan kali lebih cepat daripada algoritma seri. Persisnya, ketika nilai n meningkat, maka rasio waktu eksekusi antara program seri dan paralel juga ikut meningkat. Lebih lanjut lagi, saat n cukup besar, yaitu ketika* $n \geq 3 \times 10^6$*, rasio efisiensi GPU ini cenderung melandai ke suatu nilai konstanta* 105.53*, yang menunjukkan penggunaan kapasitas GPU yang termaksimalkan. Sementara itu, pada nilai n yang cukup kecil, algoritma seri yang dijalankan murni oleh CPU bekerja lebih cepat karena paralelisme GPU dalam skala kecil tidak mampu mengimbangi kecepatan CPU dalam mengerjakan operasi aritmatika yang kompleks.*

Kata Kunci*: GPU, komputasi paralel, rumus BBP, perbandingan CPU-GPU, metode numerik paralel.*

INTRODUCTION

For decades, the emergence of the Graphical Processing Unit (GPU) had been extremely successful in helping to boost graphic-related computation, for example rendering high-resolution images and videos. Many threads in the GPU allow many simple arithmetical calculations to be done massively in parallel. Therefore, this feature suits the graphical computational demand very well as many image processing algorithms include matrix operations which naturally could be performed using blocks of parallel agents (David Kirk, 2017).

It should be noted, however, that GPU does not provide a universal solution for all types of problems. For some surveys on challenges in GPU programming, one could consult (Brodtkorb, Hagen, & Sætra, 2013). There are three general characteristics of problems that are well suited to be computed by GPU: 1) The demand for computational amount is enormous; 2) Parallel computation scheme is substantial, and 3) Throughput is prioritized over latency.

PILAR Nusa Mandiri: Journal of Computing and Information System
Vol. 18, No. 2 September 2022 | DOI: 10.33480/pilar.v18i1.3291

Despite the limitation of the GPU-solvable problem domain, it is apparent that the trend to harness GPU for non-graphical-related computation is sharply rising due to its ability to speed up analysis using relatively affordable devices. It is often coined the term GPGPU (General Purpose GPU) Computing. Some possible applications of GPGPU computing include conducting agent-based modelling (Baylor G.Fain, 2022), computational fluid dynamics (Reguly & Mudalige, 2020), accelerating convolutional neural networks (Hu, Liu, & Liu, 2022), accelerating data query (Rosenfeld, Breß, & Markl, 2022), and numerical computation (Abdelfattah, et al., 2020).

One possible GPU application of interest is efficiently computing large amounts of non-recurrent digits of some irrational numbers. As the oldest recognized irrational number, the Archimedes constant Pi (π) might be the one which generates the most excitement in the scientific community. Many global attempts had been made to compute as many digits of the number as possible. The first ever Pi digits computation using a computer was done as early as 1950 (Reitwiesner, 1950), with 2.037 digits of Pi presented. The race to compute greater and greater number of digits was quickly emerging. It includes the ones which harness some parallel computation schemes using GPU and other devices. The latest record holder for the computation of the longest Pi digits is Emma Haruka Iwao from Google (Iwao, 2022), which was just done very recently in June 2022. The work provided 100 trillion digits of Pi under a computation time of 158 days and 12.6 hours of verification. It outnumbered roughly 60% of the previous record holder, an academics team from the University of Applied Sciences Grisons in 2021 that was successfully computed $\lfloor 2\pi \times 10^{13} \rfloor$ digits of Pi (Keller, 2021).

This challenge brings sensation for a limited ambitious community and provides a useful application in some areas. It is well known that the non-recurring digits of many irrational numbers, including Pi, have been guiding many advanced-level random number generators (Jeong, Oh, Cho, & Choi, 2020). Moreover, the infinitude of the irrational's unorderly digits also recently inspiredany physicists to suggest a new interpretation of time (Wolchover, 2022).

In this paper, we focus on studying the computational performance comparison between CPU and GPU in numerical calculations. Many such comparative studies exist, such as under the context of Convolutional Neural Networks (Yunus, Kanata, & Ariessaputra, 2021), solving partial differential equation problems (Semenenko, Kolesau, Starikovičius, Mackūnas, & Šešok, 2020), and Bayesian estimation (Kim, Williams, Hernandez-Fernandez, & Bjornson, 2022). Specifically, this paper's contribution is to compare these devices while computing long hexadecimal digits of Pi. While the GPU-assisted computation would surely be faster than the one served by the CPU alone, we will provide some quantitative data to illustrate the significance of the GPU power in optimizing numerical computation tasks.

MATERIALS AND METHODS

The main methods used in this research are the algorithm design and implementation methods, equipped with quantitative analysis of the numerical data obtained from the algorithm running results. Two approaches to algorithm computing the n-th hexadecimal digit of Pi, which are the serial (CPU only) and the parallel (GPU assisted), are designed and implemented in this project. Both algorithms are then run to obtain the comparison performance data. The running times of both algorithms are collected for various sizes of the instance. Those values are then analyzed quantitatively based on their trend concerning the instances' small and large variational size.

Three main steps to conduct this research are presented as follows.

1. Numerical Formula Manipulation

Among many mathematical identities of Pi, we pick a certain formula, commonly known as the BBP formula, introduced by Bailey-Borwein-Plouffe in 1997 (Bailey, Borwein, & Plouffe, 1997) and lately popularised by Takahashi in 2020 (Takahashi, 2020). The following is the expression of the formula

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \quad (1)$$

The main feature of the identity is the factor 16^{-k} Appearing in each summand of the infinite sum. We follow some methods and notations taken from (Bailey D. H., 2006). By this, one could compute hexadecimal digits of pi starting from the $(n+1)$-th position effectively without adding any previous numbers. If a_k is the k-th term of the sum, then by simple manipulation, from equation (1), one could obtain the following expression

$$\lfloor 16\{16^n \textstyle\sum_{k=0}^{\infty} a_k\}\rfloor \quad (2)$$

is exactly the $(n+1)$-th hexadecimal digit of pi where the notation $\{a\}$ represents the fractional part of a. We will call this expression the ***Main Value of Interest*** for the upcoming sections. For

convenience, define $S_j = \sum_{k=0}^{\infty} \frac{1}{16^k(8k+j)}$ So that the curly bracket expression in (2) can be rewritten as $\{16^n\pi\} = \{4\{16^nS_1\} - 2\{16^nS_4\} - \{16^nS_5\} - \{16^nS_6\}\}$...(3)

Now, we focus on the expression $\{16^nS_j\}$ and expose some ways to compute it parallelly. After some manipulations, the expression can be split into two parts:

$\{16^nS_j\} = \left\{\left\{\sum_{k=0}^{n} \frac{16^{n-k} \bmod (8k+j)}{8k+j}\right\} + \sum_{k=n+1}^{C} \frac{16^{n-k}}{8k+j}\right\}$(4) The upper bound of the second sum C, is a constant independent of n, chosen large enough to achieve the desired degree of precision.

We will mainly focus on computing the first sum in Equation (4) since it involves $O(n)$ number of addition and exponentiation operations. We label this term as the *Major Sum*. In contrast, the second summand only accounts for an insignificant fraction of the algorithm's running time, and we label it the *Minor Sum*. It is because the iteration only occurs a constant number of times that depends only on how much precision the user desires.

2. Serial and Parallel Schemes Design

To design the algorithm scheme, we first break down the tasks involved in the computation based on Equation (4) from the previous subsection. A high-level overview of the algorithm is also described in (Bailey D. H., 2006). The breakdown of the tasks is listed in Table 1 below. It should be noted that the referenced paper does not provide a detailed task breakdown since they are merely labels defined by ourselves for the convenience and organization of this article.

Table 1. Tasks Breakdown

Task	Location	Description	Prerequisite	Scheme Design
1	Major Sum	Computing summand terms $\frac{16^{n-k} \bmod (8k+j)}{8k+j}$	N/A	Serial/ Parallel
2	Major Sum	Summing all the terms from Task 1	Task 1	Serial/ Parallel
3	Minor Sum	Computing the summation	N/A	Serial
4	Equation 4	Computing $\{16^nS_j\}$	Task 2, Task 3	Serial
5	Equation 3	Computing the main value of interest	Task 4	Serial

Source: Bailey (2006)

The terms "Major Sum", "Minor Sum", and some other equations on the Location column refer to the definition in the previous subsection. A detailed explanation of each task is given in the next section.

3. Implementation using C and CUDA

We then run two different computer programs: 1) a purely serial program and 2) the one enhanced with the GPU parallel scheme. Both programs receive the same input n, that is, the starting digit position, after which the programs are required to output the hexadecimal digits of pi. All procedures designed serially are implemented using C, while the parallel ones use the CUDA programming language.

The device used to implement the programs has the specifications described in Table 2 below.

Table 2. Device Specification

Device	Details
Processor*	Intel Core i5-9400F, 2.90 GHz
RAM**	8 GB
System Type	64-bit
Operating System	Windows 10 Pro
GPU***	NVIDIA GTX1060 6 GB

Source: *) Intel Corp. (2019) , **) V-gen (n.d.), ***) Nvidia (2016)

RESULTS AND DISCUSSIONS

1. Design of the Algorithms

In this section, we discuss the strategy for how each task listed in Table 1 is designed. First, note that the main tasks of interest are Task 1 and Task 2 because they are the GPU-optimized sub-hairs. Both charges are responsible for computing the Major Sum, whose computing performance depends on n. On the other hand, all the other tasks mainly deal with a constant number of operations, so it seems best to compute these using CPU alone. Figure 1 below provides the visualization of the tasks involved.

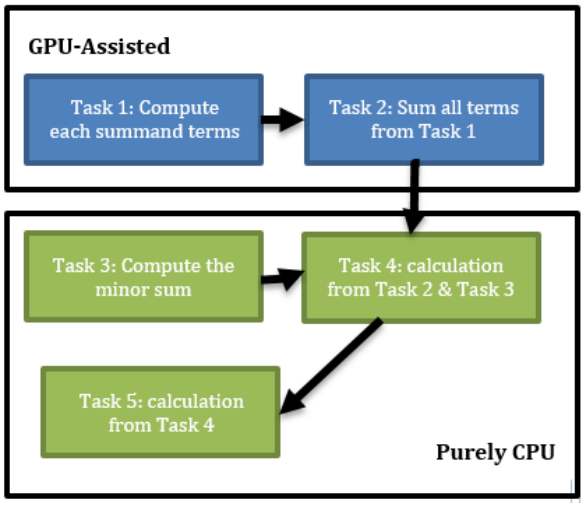

Figure 1. Tasks Dependency Diagram

PILAR Nusa Mandiri: Journal of Computing and Information System
Vol. 18, No. 2 September 2022 | DOI: 10.33480/pilar.v18i1.3291

The exponentiation procedure within Task 1 is a common computational problem solved by the Divide and Conquer strategy (Kumar & Sen, 2019). It results in $O(\log n)$ number of multiplication and modulo operations. Now, we have n typical procedures to compute the k-th term for $k = 0, 1, \ldots, n$. To compute all the summand terms serially, one could simply iterate each exponentiation computation. Meanwhile, for the parallel scheme, these n subtasks are to be split into several batches, each of which has a size of the number of GPU threads to be computed simultaneously.

Task 2, which is performing a sum of a finite sequence of numbers, is a classic task to conduct in a parallel scheme. It is often given as a standard exercise for undergraduate-level similar programming courses. The main idea used in the algorithm design is Parallel Prefix Computation. For reference, one could consult many textbooks, for instance (Hockney & Jesshope, 2019). Here, the summands are stored in an array partitioned into several blocks of subarrays. Using a divide and conquer paradigm, the algorithm keeps breaking down those subarray blocks until they are small enough to be computed for each of their paired summations. The results are then summed from other blocks in a balanced-binary-tree-like structure. Thus, if there are n numbers to be summed, then the complexity of the task is roughly $O(n \log n)$.

On the other hand, Task 3, Task 4, and Task 5 are handled solely on the CPU due to their complexity's independence of n, which means that the tasks are relatively not demanding. The number of operations (the number C of the sum upper bound) taken to perform Task 3 depends only on the degree of accuracy, the number of digits for which the result assures the exact value. The number C would not grow big as n would, so this task is not very suitable to be done in parallel. Finally, Task 4 and Task 5 are small single sets of arithmetical operations combining all other results from previous tasks. It explains the reason why both studies are naturally designed in serial schemes.

2. CUDA Implementation Details

While the algorithm implementation in the serial scheme using CPU programming is a standard exercise in terms of algorithm design, the parallel version is the main interest of this paper. In this subsection, we present the CUDA kernel functions used in this project, that is, the function to instruct how the host feeds the input data for the computation process done in the kernels, which are the heart of the parallelism.

Before exposing each of the details, it is worth noting that we use some special bit operation commands replacing routine arithmetic procedures on the kernel functions to optimize the computation. It is because the bitwise operations are substantially faster than ordinary arithmetic operations. The left shift operator (<<) and right shift operator (>>) have the arithmetical meaning of multiply by two and integral divide by 2, respectively. The 'AND' bitwise operator is applied to a variable, and the number 1 (for example: "`var & 1`") is a logical operator that returns TRUE if and only if the variable has zero value.

The following figures (Figure 2, Figure 3, and Figure 4) show the kernel functions performing Task 1 and Task 2 parallelly.

```
__global__ void HitungSukuBinary(int j, long long d, long long start,
    long long stop, double *out)
{
    // This function computes (16^(d-k) mod (8k+j) / (8k+j))
    // and stores the value on the host array named "out"
    // k-start is the thread address
    long long k = (long long)(blockIdx.x * blockDim.x + threadIdx.x) + start;
    long long res = 1LL, modulus = (8LL * k + (long long) j);
    long long k_tmp = d-k;

    if (k < stop)
    {
        long long EnamBelas=16LL;
        while (k_tmp > 0LL)
        {
            if (k_tmp & 1LL)
            {
                res = (res * EnamBelas) % modulus;
            }
            k_tmp = k_tmp >> 1; // bit shift equals to dividing by 2
            EnamBelas = (EnamBelas * EnamBelas) % modulus;
        }
        out[k-start] = (double)res / (double)modulus;
    }
}
```

Figure 2. Kernel Function for Task 1

The kernel function in Figure 2 above performs Task 1, that is, computing the value of $\frac{\left(16^{d-k} \bmod (8k+j)\right)}{8k+j}$ for all values of k with a `start` $\leq k \leq$ `stop`, where the `start` and `stop` are parameters of the function. The main algorithm would firstly break down the main range of k, which suppose to be from 0 to n, into smaller subranges whose size is suited to the capacity of the GPU number of threads and then call the kernel function using the appropriate choices of `start` and `stop`. In addition, the sub-algorithm presented in lines 14-22 of the process above employs the classic divide-and-conquer approach to compute the modular exponentiation efficiently.

```
__global__ void binary_sum(double *in, double *out, int pj)
{
    // This function computes summation of all values in array named "in"
    // by performing addition of each pair of adjacent values
    // and store its value to the array named "out".

    // The "out" array's size is half of that of "in" array
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int pjBagiDua = pj >> 1;
    if (i < pjBagiDua)
    {
        out[i] = in[i << 1] + in[(i << 1) + 1];
        out[i] -= floor(out[i]);
    }
    if (i == pjBagiDua && pj & 1) //Special handling when the array has odd size
    {
        out[i] = in[i << 1];
    }
}

__global__ void binary_copy(double *from, double *to, int pj)
{
    // This is an auxiliary function to copy values between two arrays parallelly
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < pj)
    {
        to[i] = from[i];
    }
}
```

Figure 3. Kernel Functions for Task 2

Two kernel functions described in Figure 3 above provide ingredients to implement the Parallel Prefix Computation method to compute the sum of all terms computed in Task 1 previously. First, we describe how the `binary_sum` function works. The summand terms from Task 1 are first stored in an array that would be passed as the parameter `*in` the function. The function then computes the sum of each pair of consecutive summand terms on the array and stores the result in another array whose pointer is `*out`. Thus, the resulting array *out length is half that of `*in`. The process is done for the first `pj` elements of the array, which are parameterized by the function. Some special treatment in lines 15-17 is provided to handle the case where the array length is odd. Next, the e `*binary_copy` function copies the values of the array `*from` into the array `*to` parallel with size of pj. Both functions would be called in a loop many times with decreasing values of pj that will be later explained in the next figure.

Lines 12-19 of the Host-Kernel Interaction code presented in Figure 4 describe how the kernel functions are called repetitively inside a loop. Both lines 16 and 18 indicate that variable `m` is passed as the parameter `pj` of both kernel functions in Figure 3. Also, its value always decreases as the iteration continues, as indicated by line 17.

```
1  while (pos<=d)
2  {
3      long long cut=min(d,pos+steps);
4      // << # of BLOCKS, # of THREADS >>
5
6      // Computing T summand terms from digit position 'pos'
7      HitungSukuBinary <<< B, T >>> (j, d, pos, cut, in);
8      // The summand terms are stored in the array 'in'
9
10     long long m=cut-pos;
11     // The following code performs the parallel binary addition by a batch of T threads.
12     while (m>1LL)
13     {
14         // In a sequence of operations whose array size is halving on each round,
15         // the binary_sum and binary_copy function is called to compute the summation.
16         binary_sum <<< (int)ceil((double)m / (double)(T)), T >>> (in, out, m);
17         m = (long long) ceil ((double)m/2.0);
18         binary_copy <<< (int)ceil((double)m / (double)(T)), T >>> (out, in, m);
19     }
20     cudaMemcpy(&threads_chunk_tmp, out, sizeof(double), cudaMemcpyDeviceToHost);
21
22     gpu_sum_tmp+=threads_chunk_tmp;
23     gpu_sum_tmp -= floor(gpu_sum_tmp); // Cuts all the fractional part
24     pos+=steps;
25 }
```

Figure 4. Host-Kernel Interaction

The sub-algorithm explains the communication between the host and kernel function. Note that the sub-algorithm harnesses `B` number of blocks, each of which has `T` number of threads. In our implementation, we pick the maximum number of such `B` and `T` available on the device to optimize the computation fully.

Next, we present some results obtained by the purely CPU and the GPU-assisted algorithms regarding the Pi hexadecimal digits computation. The digits outputted by the two schemes agree for all digit positions. Table 3 displays some selected Pi hexadecimal digits computed by the algorithms.

Table 3. Pi Hexadecimal Digits Obtained by BPP Formula in both Serial and Parallel Scheme

Starting digit position	Pi Hexadecimal Digits
5001	CAD18
10001	8AC8F
50001	940C2
100001	35EA1
500001	DD637
1000001	6C65E
5000001	EE394
10000001	7AF58

As the serial algorithm starts to work considerably slowly (more than 3 minutes) when the starting digit position is greater than 10^8, we now employ only the parallel scheme to explore the greatest starting digit position, which is feasibly and reasonably able to be computed by our parallel program. The furthest possible point we have already tested is the hundred-billionth hexadecimal of pi, "7FB5B", which is computable within 2156,397 seconds.

3. Running Time Analysis

In this section, we report the running times of both algorithms concerning various values of n. As the data points, we measure the running times of the algorithms for $n = 10.000, 20.000, 30.000, ..., 5.000.000$. So, in total, there are 500 data points where each consecutive pair of points has a gap of 10.000. Figure 5 shows the running time behaviour of both algorithms. The x-axis represents the n-digit position, and the y-axis (log-scaled) represents the running time.

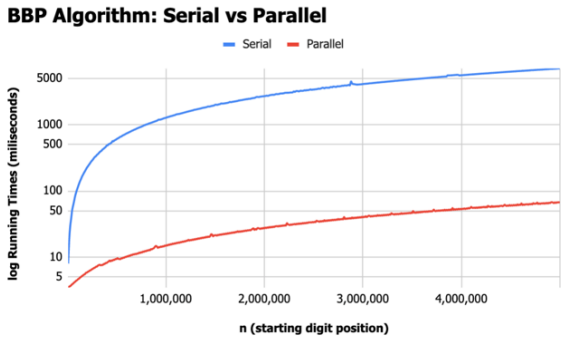

Figure 5. Running times of BBP Algorithm: two ways

From Figure 5, it is apparent that the serial algorithm runs significantly slower than the parallel one. To provide a more detailed comparison, we measure the ratio between the running times of the serial scheme to that of the parallel one. We can use this ratio as a performance measure. The greater the ratio value is, the better the parallel scheme performs compared to the serial scheme. The value of the ratio is presented in figure 6.

PILAR Nusa Mandiri: Journal of Computing and Information System
Vol. 18, No. 2 September 2022 | DOI: 10.33480/pilar.v18i1.3291

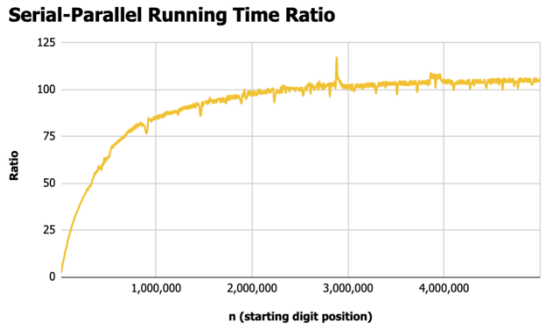

Figure 6. The ratio between running times of Serial & Parallel schemes of BBP Algorithm

Figure 6 represents the serial and parallel running time (y-axis) ratio against the n-digit position (x-axis). A closer inspection of this figure shows that when n is large enough, the running time ratio tends to converge into a certain constant, around 105.53. Its stable is stabilized once the value of n has passed over a certain threshold around 3×10^6. It means that when n is slightly larger than the threshold, the ratio number is about to converge to the ratio constant; otherwise, the percentage is still increasing as n grows.

The constant corresponds to the thread capacity of the GPU. As n grows larger, the performance improvement provided by the GPU has been optimal as all the threads have already been completely utilized. On the other hand, when the algorithms compute the formula with a value of n below the threshold, some lines of the GPU are still vacant. So, in this case, increasing the value of n would improve the performance ratio. It can be seen by the increasing curve of the running time ratio on such an interval of n (when n is less than the threshold).

Although for large n, the parallel algorithm runs faster, it is not exactly the case for some small instances of n. To be more precise, we can take a closer look at the running time curves of both algorithms when the value of n is small enough, as presented in Figure 7.

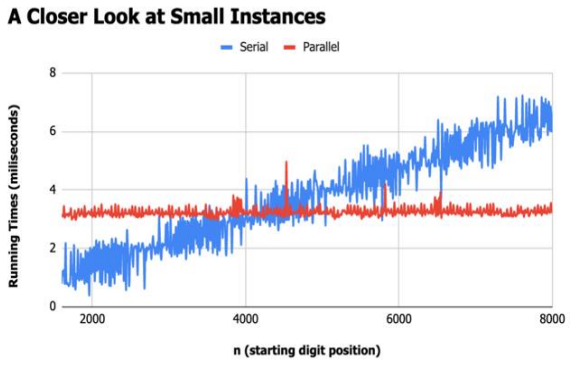

Figure 7. Running times at small instances.

From closer inspection of Figure 7, we observe that for a small value of n (around $n < 5.000$), the parallel scheme performs slower than the serial scheme. However, right after this, a similar algorithm always performs better.

We can understand this phenomenon by the following explanation. When n is small, the parallel algorithm utilizes the GPU threads for a relatively fewer number of batches. Although in GPU, many lines can work simultaneously, the computing capability of each thread is much less sophisticated than those of CPU. Consequently, when two single threads of CPU and GPU are given the same computational task, the CPU would likely perform better than GPU. It is because the CPU is designed for computing complex computations.

In contrast, GPU has a simpler computing feature in the hope that massive parallelism would help the GPU to excel over the CPU. To be more precise, by having small enough n, tasks 1 and 2 of the parallel scheme would not optimally utilize all the GPU threads, so the delay times caused by the latency might dominate the whole running time. In fact, when only a small number of GPU threads are used in the computation, the repetitive nature of serial CPU processing would perform better.

CONCLUSION

In this paper, we optimize the n-digit Pi computation by utilizing GPU parallelism. The performance of GPU and CPU processing for computing n-digit Pi is then compared. For most values of n, the parallel algorithm assisted by the GPU runs significantly faster than the serial algorithm. Furthermore, this research measures the running time ratio between CPU and GPU algorithms. In other words, this ratio measures how many times the GPU works better than the CPU. From the implementation data collection, we found that the ratio tends to increase as the n is rising. It shows the supremacy of GPU performance over CPU in n-digit Pi calculation. In addition, the balance appears to converge to 105.53 once n has passed over 3×10^6. It shows the fully utilized capacity of the GPU in performing computation with a large enough size instance. However, a closer inspection of small cases indicates that the CPU performs better than GPU due to a lack of parallelism. The finding shows that numerical computation optimization should be strategically planned rather than relying on purely GPU computation. The computation decision should be carefully designed depending on the context of the use case. Furthermore, a more rigorous investigation of several alternatives of parallel numerical computation for various irrational numbers will be reserved for future works.

REFERENCES

Abdelfattah, A., Anzt, H., Boman, E. G., Carson, E., Cojean, T., Dongarra, J., . . . Li, S. (2020). A survey of numerical methods utilizing mixed precision arithmetic. *arXiv preprint arXiv:2007.06674*. https://arxiv.org/pdf/2007.06674.pdf

Bailey, D. H. (2006). *The BBP Algorithm for Pi.* Berkeley: Lawrence Berkeley National Lab.(LBNL). https://www.osti.gov/servlets/purl/983322

Bailey, D., Borwein, P., & Plouffe, S. (1997). On the rapid computation of various polylogarithmic constants. *Mathematics of Computation, 66*(218), 903-913. https://doi.org/10.1090/S0025-5718-97-00856-9

Baylor G.Fain, H. M. (2022). GPU acceleration and data fitting: Agent-based models of viral infections can now be parameterized in hours. *Journal of Computational Science*. https://doi.org/10.1016/j.jocs.2022.101662

Brodtkorb, A. R., Hagen, T. R., & Sætra, M. L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing, 73*(1), 4-13. https://doi.org/10.1016/j.jpdc.2012.04.003

David Kirk, W.-m. H. (2017). *Programming Massively Parallel Processors.* Cambridge: Elsevier.

Hockney, R. W., & Jesshope, C. R. (2019). *Parallel Computers 2: Architecture, Programming and Algorithms.* CRC Press.

Hu, Y., Liu, Y., & Liu, Z. (2022). A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. *2022 14th International Conference on Computer Research and Development (ICCRD)* (pp. 100-107). IEEE. https://doi.org/10.1109/ICCRD54409.2022.9730377

Intel Corp. (2019). Intel® Core™ i5 Processors. (Processors Productions) Retrieved 8 10, 2022, from Intel.com: https://www.intel.com/content/www/us/en/products/sku/190883/intel-core-i59400f-processor-9m-cache-up-to-4-10-ghz/specifications.html

Iwao, E. H. (2022, June 8). *Even more pi in the sky: Calculating 100 trillion digits of pi on Google Cloud*. Retrieved from Google Cloud Blog: https://cloud.google.com/blog/products/compute/calculating-100-trillion-digits-of-pi-on-google-cloud

Jeong, Y.-S., Oh, K.-J., Cho, C.-K., & Choi, H.-J. (2020). Pseudo-random number generation using LSTMs. *The Journal of Supercomputing, 76*(10), 8324-8342. https://doi.org/10.1109/BigComp.2018.00091

Keller, T. (2021, August 14). *World record attempt by UAS Grisons: Pi-Challenge*. Retrieved from University of Applied Science Grisons Website: https://www.fhgr.ch/en/themenschwerpunkte/applied-future-technologies/davis-centre/pi-challenge/

Kim, D. H., Williams, L. J., Hernandez-Fernandez, M., & Bjornson, B. H. (2022). Comparison of CPU and GPU bayesian estimates of fibre orientations from diffusion MRI. *Plos one, 17*(4), e0252736. https://doi.org/10.1371/journal.pone.0252736

Kumar, A., & Sen, S. (2019). *Design and Analysis of Algorithms.* Cambridge University Press.

Nvidia. (2016). Geforce GTX 1060 Specifications. (Graphics Cards Productions) Retrieved 8 10, 2022, from nvidia.com: https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1060/specifications/

Reguly, I. Z., & Mudalige, G. R. (2020). Productivity, performance, and portability for computational fluid dynamics applications. *Computers & Fluids*, 104425. https://doi.org/10.1016/j.compfluid.2020.104425

Reitwiesner, G. W. (1950). An ENIAC Determination of π and e to more than 2000 Decimal Places. *Mathematical Tables and Other Aids to Computation, 4*(29), 11-15. https://doi.org/10.2307/2002695

Rosenfeld, V., Breß, S., & Markl, V. (2022). Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR), 55*(1), 1-38. https://doi.org/10.1145/3485126

Semenenko, J., Kolesau, A., Starikovičius, V., Mackūnas, A., & Šešok, D. (2020). Comparison of GPU and CPU efficiency while solving heat conduction problems. *Mokslas-Lietuvos ateitis/Science-Future of Lithuania*, 12. https://doi.org/10.3846/mla.2020.13500

Takahashi, D. (2020). On the computation and verification of π using BBP-type formulas. *The Ramanujan Journal*, 177-186. https://link.springer.com/article/10.1007/s11139-018-0104-x

V-Gen. (n.d.). V-GeN Platinum DDR 4 PC 19200 - 2400 MHz. Retrieved 8 10, 2022, from V-Gen.co.id: https://v-gen.co.id/ram/v-gen-platinum-ddr-4-pc-19200-2400-mhz-ecc/

Wolchover, N. (2022). Does time really flow? New clues come from a century-old approach to math. *The Best Writing on Mathematics 2021, 19*, 183.

Yunus, I., Kanata, B., & Ariessaputra, S. (2021). Perbandingan Kinerja CPU dengan GPU dan Tanpa GPU dalam Pemrosesan Gambar Menggunakan Metode Convolutional Neural Network. *Indonesian Journal of Applied Science and Technology, 2*(4), 127-134. Retrieved from https://journal.publication-center.com/index.php/ijast/article/view/1310