THE LIMITS OF PARALLELISM IN GEOINFORMATICS DATA PROCESSING: WHAT YOU CAN AND CANNOT GET FROM GPU PROCESSORS

Kurt T. Rudahl[*a]* and Sally E. Goldin*[b]

[a]*Faculty, Department of Computer Engineering, King Mongkut's University of Technology Thonburi*
91 Pracha Uthit Road, Bangkok 10140, Thailand
E-mail: kurt@cpe.kmutt.ac.th

[b]*Faculty, Department of Computer Engineering, King Mongkut's University of Technology Thonburi*
91 Pracha Uthit Road, Bangkok 10140, Thailand
E-mail: seg@goldin-rudahl.com

KEY WORDS: Parallel Algorithms, Raster Image Processing, GPGPU, Performance Monitoring

ABSTRACT: The continuing explosion in available geospatial data imposes rapidly increasing demands for computational power. At the same time, computer manufacturers are reportedly reaching the limits of single processor capability. A parallel processing paradigm, where processing load is spread across multiple computing units, has been widely promoted as a solution to this dilemma. The General Purpose Graphical Processing Unit (GPGPU) hardware produced by AMD/ATI and Nvidia appears to offer an effective, low-cost approach for achieving parallel execution, even though in every case the software needs to be partly rewritten to take advantage of the GPGPU capabilities.

In theory, raster image processing algorithms should be extremely appropriate for parallel processing. However, our work adapting our Dragon/ips® software to use parallel hardware, including GPUs and multi-core CPUs, has revealed that the reality of converting a full end-to-end task is very different from the theory which considers the core processing algorithm in isolation. Factors such as available memory sometimes have a larger impact on performance than the parallelism *per se*.

This paper reports on measurements made during the software conversion, using both internal instrumentation and external profiling tools. We show that the measurements generally agree with theory, but only after the theory has been extended to consider the entire end-to-end process. Most tests were done using Linux; however, checks using Windows indicate that the results do not depend significantly on the operating system. Our results should provide guidance to geoinformatics software developers, helping them to evaluate the performance improvement which can be anticipated from converting geospatial software for parallel processing.

1. BACKGROUND

We have been personally involved with geospatial data processing for more than 25 years, and with computers for longer than that. During that period, the capabilities of computers have increased at an almost inconceivable rate. For example, the cost of data storage on hard disks has decreased in cost by almost eight orders of magnitude (100,000,000 times). However, the amount of geospatial data we need to process is growing even more rapidly than computer capabilities. Thus, even with the vast increases in computational resources available, we are losing ground in our ability to process the data that we have.

Unfortunately, the disparity between needed data processing and computer speed is likely to get worse. As the potential for increasing the speed of single-core CPUs reaches its limits, the industry is turning to parallel-processing architectures to be able to continue to achieve increases in performance. Currently popular parallel-processing architectures fall into three categories: multiple computers working cooperatively (a *cluster*) (Rudahl, 2002), multiple *cores* within one computer, and *coprocessors* (possibly with multiple cores) which are installed into a standard computer and can offload processing burden from the main computer. The currently popular coprocessor is the *General Purpose Graphics Processing Unit* (GPGPU), which is a display processor repurposed to make it available for non-display tasks.

Of these architectures, only the multiple-core form can run software which has not been rewritten specifically to exploit parallelism, and even with these units it is difficult to achieve proportionate speedups of non-modified software when the number of cores exceeds two or three.

Rewriting software to use parallelism involves significant obstacles and substantial expense. Problems include the lack of developers familiar with parallel architectures, lack of a consistent programming model which can be applied uniformly across the different parallel architectures, and lack of tools for measuring and tuning processes running on parallel systems. However, regardless of the difficulties, revising software to use a parallel architecture is now, or soon will be, an area of concern to every producer of high-performance software including geospatial analysis software.

The difficulty of rewriting existing software to benefit from a parallel architecture varies, depending on the nature of the software, from relatively easy to almost impossible. In principle, raster based image processing such as most of the operations in our well-known Dragon/ips® should fall into the "relatively easy" category because for many algorithms there is little data dependency between different regions of a raster image.

The newest version of Dragon has expanded capabilities that allow it to process image data sets as large as 32,000 rows by 32,000 pixels, which is as large as the data set produced by any commercial satellite that we know of. The new processing works well, but is not as fast as we would like.

How fast is fast enough? There are two measures. For the user sitting at a computer waiting for results, the processing should be fast enough that the user doesn't become impatient, bored, or unable to complete her work on schedule. However, the user generally understands that a "difficult" task (e.g. processing a large image) takes more time than an "easy" task. Any task which requires more than a few minutes, however, risks incurring the user's displeasure.

A more stringent requirement is *real-tim*e processing. Real-time constraints arise when there is some type of event which comes from outside the system and cannot be controlled by the system. A *real-time system* is one which can complete the processing of one event before the next one occurs. For example, an on-board image processing system used in aerial image acquisition needs to process each image received before the next one arrives.

In preparation for the next major release of Dragon, we have been exploring the possibilities for speed increases both by use of parallel processing such as a GPGPU, and by other means. Even in the best case, however, rewriting software is a time and effort consuming process. Therefore, we wanted to evaluate in advance just what benefits we might achieve from the rewrite process. Although our testing has focused on the GPGPU, the results presented here have also been tested on a multi-core processor and should be equally applicable to any other parallel-processing architecture.

2. THEORY

The textbook principle of parallel programming is that different parts of a program, which would normally happen one after another, are made to happen at the same time, but on different compute units. Thus, ideally, something which takes 40 seconds on one processor should take only 10 seconds on four parallel processors. **Note that this ignores communication time between the processors.** However, this parallelization can only be applied when the different parts of the program are independent of each other. For example, in the code segment

```
int x[100];
int y[100];
for (j=0; j<100; j++)
    {
    x[j] = y[j];
    }
```

each `x[j]` is independent of every other `x[j']`, so this could be parallelized. (Please just ignore these code samples if you are not comfortable with programming languages.)

However in,

```
int count=0
int y[100];
for (j=0; j<100; j++)
    {
    count = count + y[j];
    }
```

the variable `count` is used in each cycle, so we cannot assign different iterations of the loop to different processors.. This means that that this code cannot be parallelized. The classical example of a non-parallelizable activity (in a non-computer context) is: “if a woman can produce a child in nine months, how long will it take nine women to produce a child?”

The above principle of parallelization is only relevant to those parts of a program which are concerned exclusively with computation. This is of great interest to someone analyzing algorithms, but any real program contains other activities as well. For Dragon, we consider that an invocation of an operation by a user has the components shown in Figure 1.

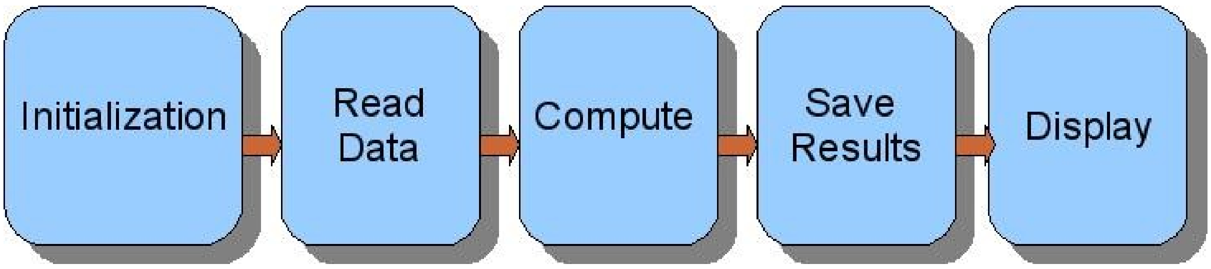

Figure 1. Typical Dragon Compute Sequence

Initialization consists of any processing needed to analyze and set up parameters specified by the user, scripts, or configuration files. In general, it cannot be parallelized, but also does not consume much time.

Read data consists of reading data (for example, image data sets) from somewhere (usually the hard disk or network), into memory. Note that the diagram implies that all data is read before computation begins, which in turn implies that memory is large enough to hold all of the data. This is discussed in more detail in the next section.

Compute is the computational processing section, which by assumption happens as quickly as possible, using only the computation units and memory. This is the only part of the program which can be parallelized.

Save results consists of writing data (for example, a classified image) from memory to disk or network. In some cases, there may not be any resulting image data to save.

Display consists of showing the results to the user. The Dragon operations almost always conclude by showing the resulting image, if any, to the user. However, this step is not actually necessary, so we will ignore it for purposes of this discussion.

The remainder of this discussion is based on the above assertion that it is only the compute component which can be parallelized. There are a couple of questions which should be considered first.

1. Is it possible to parallelize reading data from the disk? In principle it is possible, but we do not believe that a standard desktop computer has the hardware to support this. Furthermore, the speedup from parallelizing could not be a larger factor than the number of hard disks which is normally only two or three. Furthermore, the measurements reported below indicate that the disk reading speed is not limited by the disk itself, so parallel reads are unlikely to reduce time significantly.

2. The normal behavior for a display processor such as the GPGPU is to not save the results, but to display them directly. Could that be a way to speed up the above processing? In general, this strategy could not be applied to geoinformatics data processing, partly because the results are usually needed for further analysis, and partly because the image size is far larger than can be displayed on current display hardware.

It is clear that, in Figure 1, the total time for execution is the sum of the times of each of the five components. For very fast operations, the time used by initialization and display may be a significant fraction of the total, but for slow operations, which are the ones we are interested in, these two components may be ignored. Therefore, the total time is approximately the sum of the read and save times, plus the compute time.

$$\text{time}_{\text{Total}} \cong \text{time}_{\text{Read}} + \text{time}_{\text{Compute}} + \text{time}_{\text{Write}} \qquad (1)$$

Because the read time and write time will equal the amount of data multiplied by the speed of reading and writing data (and assuming that read speed is equal to write speed), we rewrite equation (1) as

$$time_{Total} \cong (data_{Read} + data_{Write}) * iospeed + time_{Compute} \qquad (2)$$

In general, the approximate amount of data to be transferred can be determined by inspection of the task, and the speed of data transfer can be measured independently of any reprogramming for purposes of parallelization. This means that we can calculate a lower bound for the execution time (or, equivalently, an upper bound for the speedup factor) by simply assuming that the compute time is zero.

3. EXPERIMENTAL RESULTS

We selected four Dragon operations as candidates for testing:

1. Single band display (D/1BA), which has only a trivial compute component.
2. Minimum distance to mean supervised classification (C/SUP/MDM), which has a moderately complex compute component
3. Clustering (C/UNS/CLU) which has essentially the same compute component as C/SUP/MDM, but iterates over the input data many times.
4. Maximum likelihood supervised classification, which has a quite complex compute component.

Each GPGPU manufacturer has an unique programming interface, different from each other and different, also, from the programming interface which might be used for a multi-core computer of a computer cluster. Because we are writing software which will be distributed to many users, it is necessary for us to write the parallel component in a form which is portable to as many different parallel platforms as possible. We chose OpenCL (Khronos, 2011) as the interface because it can be used on both GPGPUs as well as on a multi-core machine, because it can be used with Linux, Windows, and Apple, and because it is a familiar C-like language. Although it has been reported that though OpenCL is slower on the GPGPUs than their native languages (See, 2011), this fact does not alter the conclusions reached by our research.

The first step was to instrument the software so that we could measure how much time was being spent in each of the conceptual parts of the program. As it turned out, it was also necessary to do significant rewriting in order to achieve the structure shown, in which all data is read only before any computation is begun. Having accomplished those steps, it was quite easy to measure the component execution times with excellent repeatability. However, it also became apparent that the process of revisiting and rewriting old software was *by itself* a source of considerable performance improvement. The apparent reason for this effect is that the most recent significant revisions to the source code were done when memory and other compute resources were more expensive. At that time, conserving memory as well as saving time were both design objectives, while in the current revision we essentially assumed unlimited memory to be available.

We started the examination by rewriting clustering. By converting it into a parallel form, we were able to achieve several times speedups of the compute component while testing it on a multi-core computer and on two different GPGPUs from Nvidia and ATI/AMD. The test data set we have chosen is a four-band, 50 million pixel data set (6589 lines x 7819 pixels) provided to us by the Geo Informatics and Space Development Technology Agency (GISTDA) from Thailand. Thus, the input data for all the classification operations is about 200 Mbyte, and the result is about 50 Mbyte. Because our test machine has 4GByte of memory installed, the memory is not a constraint in these tests.

The clustering operation cycles through the input data numerous times (30 in our tests) but reads it and writes the result only once. Accordingly, parallelizing the software did realize a significant overall time improvement.

Our next interest was to determine what the best execution time might be, given the theoretical discussion above. To do this we needed to know that actual time needed for hard disk reads and writes of the 250 Mbyte used by this operation. The manufacturer specifies the disks as 3 gbits/second (375 Mbyte/second) maximum transfer rate, which implies that our 250 Mbyte should transfer in less than a second. However, actual measurement of writing a large file by the operating system showed a transfer rate of only about 40 Mbyte/second (2.4 Gbyte/minute) so that about seven seconds were needed for 250 Mbyte. Note that this implies that the data transfer rate is being limited by something other than the disk drive, so that parallelizing the data to come from several disks would probably not be productive.

4. CONCLUSIONS

Based on our efforts so far, we conclude that:

1. It is possible to gain significant processing speed by reworking the code to use a parallel processing architecture.
2. The process of reworking the code in preparation for parallel processing can itself have significant benefits in processing speed. For example, we reduced the time for a 30-iteration clustering by more than 20%, and minimum distance to mean classification time was decreased by 50%. (However, these speed increases require that the system have enough memory to hold all the data at once.)
3. The speed increases in the part of the code which was parallelized were measured as up to a five-times increase, even though the overall improvement was much less.
4. Parallel processing does not produce benefits in all cases.

In addition, there were some significant issues in using the GPGPUs:

5. The AMD/ATI device can apparently only be used as a GPGPU when it is also driving the display. Thus, the GPGPU cannot be used as a coprocessor when the session is being run remotely as an X client. (This apparently is not true of the Nvidia device.)
6. For both boards, if an error occurs which causes the GPGPU to hang (which seems to happen frequently), there is no way to reset the GPGPU from the host computer. The only recourse is to reboot.
7. If the GPGPU is being used as the computer's display board, then the display appears to freeze while the board is being used as a coprocessor. (That is, the screen doesn't refresh and the mouse cursor does not move.)

Some of the above issues might not be problems when using the native GPGPU APIs rather than OpenCL.

5. EXAMPLES

As an example of how the above results can be used to estimate the benefits of parallel processing, even in the absence of any currently existing software, we will describe a design we were asked to prepare on behalf of a Thai agency. The problem consisted of a system which would use a low-elevation unmanned autonomous vehicle to gather several hundred photographs by flying in a zigzag over a target area. The photographs then needed to be automatically registered and mosaicked.

Leaving aside the question of how to automatically register the photographs, the agency needed to know how long the processing would need to run in order to assemble the photographs into a composite image.

Assuming 256 photographs, each three-band at 5000 x 5000 resolution with 8-bit data, we would have about 20 Gbyte of input data. Assuming a 32,000 x 32,000, three-band generated result, there will be about 3 GB of output data. 23 Gbyte of memory to hold all of this data at once is large, but not impossible. Reading the input data once and writing the result data would require at least 11 minutes. Thus, the fastest this program could possibly run would be 11 minutes.

6. FUTURE WORK

We plan to continue our exploration of parallelizing Dragon especially for our slower operations. If successful (which we expect it will be) we will be offering a new Dragon version in the near future.

The limitations of the GPGPUs reported above are disappointing, particularly since OpenCL does not currently offer the alternative of using cluster machines as a parallelizing architecture. The multi-core approach was successful, but is limited by the availability of processors with only a small number of cores. Therefore we want to explore both possibilities for extending OpenCL to a larger class of situations, and other methodologies for achieving parallelism.

Another question we would like to explore is why there is such a disparity between the manufacturer's specified disk data transfer rate (375 Mybte/second) and the measured rate (50 Mbyte/second). Our measurements were done on several computers and several disks, and were quite consistent.

We would also like to do further testing, especially on Windows and Macintosh, and on a larger Nvidia GPGPU.

7. REFERENCES

Khronos OpenCL Working Group, The OpenCL Specification, www.khronos.org, 2011

Rudahl, K. and Goldin, S., PC Clusters as a Platform for Image Processing: Promises and Pitfalls, Proceedings of the 23rd Asian Conference on Remote Sensing, 2002

See, S., HPC-Where Have We gone!, keynote speech, Thai Grid and Cloud Conference, 2011