SOFTENG 751 REPORT

Group 12 - Project 2

https://github.com/mkem114/SOFTENG-751-Assignment-ParaStats

Blair Cox[1], Logan Small[2], and Michael Kemp[1]

[1]Department of Electrical and Computer Engineering, University of Auckland
[2]Department of Engineering Science, University of Auckland

Contribution to Work Item	**Blair Cox**	**Logan Small**	**Michael Kemp**
Skeleton Code	60%	20%	20%
CI & Unit Testing (w/ Google Test)	10%	10%	80%
Command Line Interface	30%	10%	60%
Algorithm Design / Statistics	10%	80%	10%
File Reading	60%	20%	20%
Sequential Code	10%	35%	55%
OpenCL Host Code	45%	45%	10%
OpenCL Kernel	45%	45%	10%
Presentation	33%	33%	33%
Report	33%	33%	33%

1 Introduction

The aim of this report is to detail the work undertaken into implementing the calculation of statistical values from large datasets derived from natural processes. 'Statistical values' was understood as descriptive statistics that summarise a data set through representative scalar values, such as minimum, mean, standard deviation. We took 'natural processes' to mean a process that generates a data stream of continuous values with existing statistical values. Finally, the term 'large data sets' was taken to mean anything that is inefficient to process through ordinary means, but still manageable on a shared-memory system as distributed systems are out of our scope (on the order of 10^9, <10GiB).

2 Approach

It was essential to realise the requirements of this project before tackling the code. This meant ensuring a consistent understanding between group members of the process we would take, as well as the project specifications. As a result, we conducted some initial research and data creation before the implementation began.

2.1 Data Creation

As we would be dealing with large data-sets, we looked at creating a standard format which would reduce some of the difficulties of processing input data. We decided to use CSV files as they are a widely used standard with minimal file-size. The data was generated through a custom MATLAB script that could generate random continuous values from a given probability distribution. We assumed the format of the input data would contain these attributes:

- Clean data (no other characters, legitimate values, decimal notation etc.)
- No unnecessary gaps between data points
- The size of the data set was a multiple of the work group size.

The first two assumptions are valid as the input data is pre-processed so can be cleaned at the CSV-generation step (out of scope). The final assumption is valid as if the data set were not a multiple of the work group size, the final data points would be processed in sequential due to the overhead of creating another buffer. The test data was created by assuming normal and logistic probability distributions, and randomly generating data from these. We replicated 3 processes, temperature (μ(mean)=20°C), voltage (μ=230V) and air pressure (μ=1020hPa), each with 5 significant figures. These provide realistic data sets that might be expected in practice such as streams of data from power meters, barometers and thermometers which would create millions of data points.

2.2 Statistics

Descriptive statistics are the representative scalar values that describe a data set and its estimated probability distribution. This includes measures of location (mode, mean, median, etc), measures of spread (standard deviation, variance) and measures of shape (skewness and kurtosis). We have discarded co-variance statistics for multivariate datasets as data from multiple variables is out of scope. We can calculate all of these values from some minimal set of meta-statistics. These aggregate-statistics are the sample size, first 4 orders of statistical moments, min/max and a histogram. The first 4 statistical moments can be used to calculate the mean, standard deviation/variance, skewness and kurtosis respectively. The histogram can calculate the mode (and second-most-frequent, least-frequent, etc) as well as rank statistics (median, upper quartile, lower quartile, etc). Thus if we find efficient algorithms for calculating these meta-statistics, we can obtain any of the required statistics.

Minimum and Maximum are calculated by just comparing the current Min/Max against a new candidate. This can trivially be implemented as an online algorithm (for sequential) or parallel reduction (in OpenCL).

The first 4 statistical moments have well-defined and numerically-stable algorithms [1] for both online [2] and parallel [3] implementations that require only 1 floating point division per update. These algorithms do not suffer from the underflow/overflow that naive algorithms do as there is no summation occurring. The only case where underflow/overflow could occur is when the range of the dataset is very large, which is unlikely in the case of natural processes. In the case of reducing a vector where the length is a power of 2, we can simplify the equation to require no floating point divisions, as such we have implemented this algorithm in our OpenCL kernel. The first order moment M_1 is equal to the mean. Variance (square of standard deviation) is the 2nd moment divided by 2: $\sigma^2 = \frac{M_2}{2}$. The skewness can be calculated as $\frac{\sqrt{n}M_3}{M_2^1.5}$. Finally kurtosis is $\frac{nM_4}{M_2^2}$. These values along with min/max are sufficient to describe a dataset and estimate its probability distribution.

The histogram requires a finite number of bins, so inherently will lose accuracy due to larger representation error. However, if we know the properties of the probability distribution (such as the range, mean, dispersion, shape as calculated above)) we can use non-uniform bin sizes to minimise the errors to the point where double-precision error and sampling error have similar magnitude. Smaller bin sizes can be used closer to the peaks of the dataset where a finer distinction will be required. Finding the mode (or kth-frequent) value can be done by calculating the max (or kth-rank) statistic of the histogram. Finding the median (or k-th rank) can be found by summing along the histogram till you reach the bin that contains half of the sample size. Computing the histogram requires significant local (work-group) memory, so becomes infeasible in the case of GPU acceleration using OpenCL as memory overflows to the host and communication costs outweigh any potential speedup. The histogram calculation is more suited towards an OpenMP approach. Due to poor initial results in OpenCL and time constraints, this meta-statistic was not implemented.

2.3 Development setup

Before we started implementing our project, we first needed to think about how our development process would function. Our process would need to include both the approach we would need to take for coding the project, as well as how we could ensure that our system was both functioning correctly, and correctly implemented. These reasons provided us with the motivation to use technologies such as version control systems and continuous integration.

2.3.1 Version Control

We used version control as it would allow all group members to be able to concurrently develop the project and be able to implement different functionality in parallel. Version control also provided a code review process for this project, which helped ensure correct functionality of the developing project.

2.3.2 Continuous Integration/Testing

Being able to ensure correctness of implementation was a significant goal of the project. We chose Travis-CI as it would help provide checks that would ensure this correctness, by running tests anytime a branch is pushed to or pull requested. Continuous integration also allowed for the master branch to remain in an unbroken state so that each group member could work off of it. Tests were used both on Travis-CI and locally to ensure that results on parallel and sequential were within our accuracy goal.

3 Implementation

3.1 Sequential

We attempted to make the sequential implementation as fast as possible to give a valid comparison to the OpenCL version; this would show if the OpenCL implementation was worth the effort.

3.1.1 Optimisations

Our implementation uses an online algorithm to calculate the core values on every data point, this retains numerical accuracy and is the same speed as the ‘chunked’ version (even correctly tuned for the cache). Partial calculations are reused to determine more than one core statistical value to reduce operations. The current value of the statistical values is only updated in chunks to reduce overhead.

3.1.2 Implementation

```
n = mean = M2 = M3 = M4 = 0
for x in data:
    n1 = n++
    delta = x - mean
    delta_n = delta / n
    delta_n2 = delta_n**2
    term1 = delta * delta_n * n1
    mean += delta_n
    M4 += term1 * delta_n2 * (n*n - 3*n + 3) + 6 * delta_n2 * M2 - 4 * delta_n * M3
    M3 += term1 * delta_n * (n - 2) - 3 * delta_n * M2
    M2 += term1
```

Listing 1: The equations mentioned above were implemented in the incremental case

3.2 Parallel

3.2.1 Kernel

We reduced all our statistical calculations down to one kernel. One kernel allows for a lesser amount of overhead when transferring data to and from the host, which would increase the overall efficiency of our implementation. Minimum, maximum and all the moments are calculated in a singular kernel with a reduction based for loop with different strides for efficient and concurrent calculations. Barriers are used in assistance to strides to allow the kernel to execute without issues with concurrency. This kernel operates using the data parallelism approach as each data point is operated on by a different compute unit. The kernel is only compiled once but can be run multiple times in 'chunks' when the data size exceeds the supported size of the device.

3.2.2 Optimisations

When we were finished implementing the base statistics in the `stats.cl` kernel, we looked at optimising the calculations to reduce the number of operations, and to reduce the total memory used. Reducing space was vital as it allowed for more memory to be used on other statistical value storage, and reducing calculations decreased the time taken to calculate each statistical value, hence increasing speed up overall.

Data Transfer and Memory In order to increase the throughput of data transfer, we had to reduce both allocated memory on devices, and data transferred into buffers. These two are intertwined as reducing one, allows the other to increase. Being able to increase the rate of data transfer is important as it provides an increase in the work group size and amount of work groups that could be transferred. Therefore by implementing this optimisation technique, larger vectors could be transferred onto the input buffer, so that more data could be processed in parallel, with less transfer time overhead.

An example of when we executed this technique was min/max reduction and calculation, as can be seen in listing 2. Initially, we had both minimum and maximum being calculated in separate arrays, being reduced down to one work item per work group at the end. This process meant that one array was needed for both minimum and maximum, and required more memory to hold. What we did to solve this was to reduce and calculate on the first iteration, and max/min would naturally separate the maximum and minimum values. Once this was done, maximum and minimum calculations could be done on separate sections of the array, without having to worry about concurrent access.

```
if (!(id % (i * 2)) && ((id + i) < size)) {
            local_minmax[id] = (local_minmax[id] > local_minmax[id+i]) ?
            local_minmax[id+i] : local_minmax[id];
} else if (!((id - 1) % (i * 2)) && ((id - 1 + i) < size)) {
            local_minmax[id] = (local_minmax[id] < local_minmax[id+i]) ?
            local_minmax[id+i] : local_minmax[id];
}
```

Listing 2: Work item calculation splitting over min and max

The moments structure is an array of structures, but its length only requires half the amount of memory for the whole array, compared with the input array. It is able to be constructed like this as the initial moment calculation is done on the original array (min/max array), and reduced down to half of the total work group size instantly, onto this structure.

Calculations As stated above, we reduced the number of calculations by doing the first minimum and maximum calculation together. Another way we reduced calculations was to minimise the amount divisions used, as divisions can be rather expensive. When reducing down, we also made sure that as many work items would still be doing work, this would help increase occupancy and decrease downtime of work-items hence increasing overall speed.

3.3 File Reading

File reading was a common functionality that was needed between the parallel and sequential implementation. It was essential that both sequential and parallel used the same file reading process as it would help isolate the speedup between calculations, and not reading files. When dealing with large files, it is inefficient to read a file fully into memory and needs to be buffered in some way. Originally we had files being read in using the typical `std::stringstream` and `getline()`, but this proved to be inefficient and slow. Now the file is loaded into an internal stream buffer; the buffer is traversed to find the number of characters in it. Then a char pointer is allocated the size calculated from before and `buffer->sgetn(pointer, size)` is called to copy the buffer into the pointer.

3.4 Command line arguments

We have added some command line argument options for the user to change the run time behaviour to pick what is best for them. The sequential and OpenCL implementation can be used, output file writing, input file to read etc.

For the OpenCL implementation the user can specify different vector input sizes which corresponds to a work group size. Usually, with GPUs, a larger work group size will give better performance but too large could either crash the device (bigger than it can fit) or cause incorrect results to be given (using zeros because it's going out of bounds). Furthermore, selecting a vector size that is not an exponential of 2 crashes some devices.

4 Results and Evaluation

4.1 Data set sizes

Using a computer with only 8GB of memory a 16GB file with 2,621,440,000 records ran successfully, due to time constraints it's difficult to say with certainty that significantly larger files will work. When talking about hundreds of GBs or TBs of data then it is likely that a distributed system will be used, not a shared-memory system.

4.2 Comparisons

4.2.1 Parallel vs Sequential

As can be seen in Figure 4.2.1, there is a lack of speed up when running the same data set in parallel and sequential. We tested each problem in parallel on both CPU and GPU, and found that using the GPU was significantly faster, so our results are from the GPU. Parallel run time is consistently higher than sequential, with times converging with larger datasets. The main factor causing OpenCL to run slower was that the overhead of setting the device up and transferring the data far outweighs the benefit of running the calculations on a compute device. Running these computations with OpenMP would provide the needed speed up and due to the problem structure it would also suit OpenMP more than OpenCL. When operating on the same dataset, changing work group size had minimal effects. We believe this is again due to the overhead of transferring data. As the same amount of data will be needed to have been transferred no matter the work group size, it far outweighes the small benefit of changing the work group size [4].

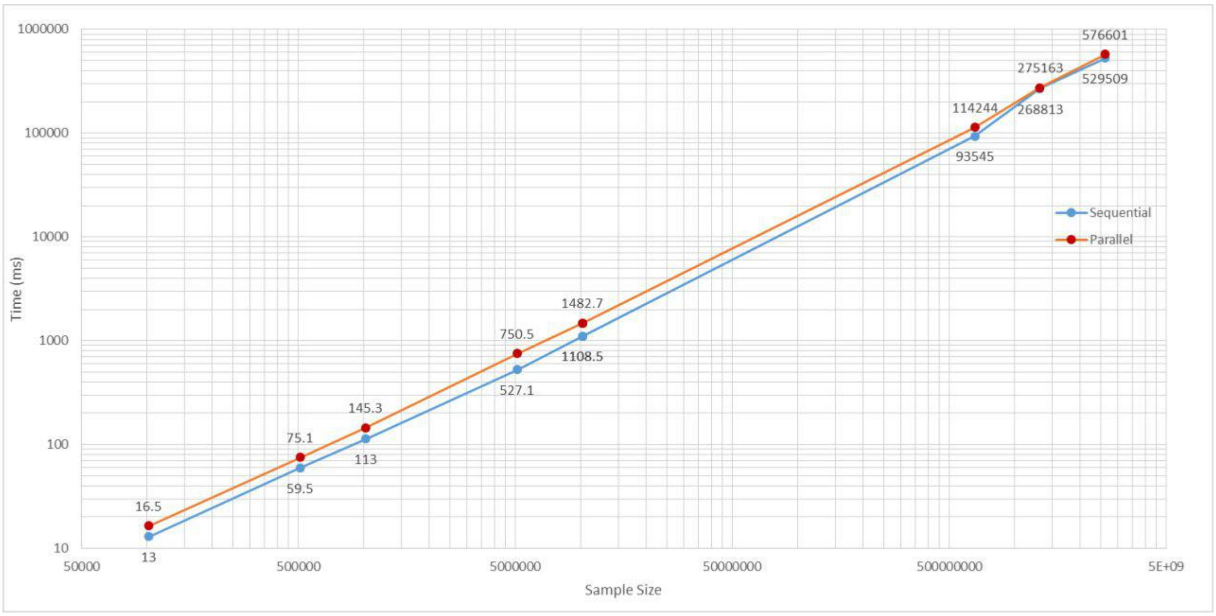

Figure 1: Sequential vs Parallel with different data set sizes

4.2.2 MATLAB

We had issues creating dataset files greater than 4GB because even 64-bit MATLAB wouldn't allocate that much memory on the heap. To generate larger files we had to concatenate the 4GB file with `cat`. MATLAB is also affected by `double` representation error in the same way and its complexity caused CSV reading to be longer.

4.2.3 Other

The most popular packages for analysis of large datasets are statistical programs such as Microsoft Excel, SAS or R. Excel is limited by spreadsheet size and thus is limited to approx 1million rows of data. SAS has limits on local-data imports of 4GB. The recommended approaches for using data sets larger than these on desktop programs is to split the input into multiple files. The 32-bit version of R has a limit of $2^{32}-1$ vector length which is the same order of size as we tested with our program. R also supports using single-file large data sets; however, it employs big-data concepts that were beyond the scope of this project.

5 Limitations and Challenges

From our results it's clear to see that calculating statistical values in OpenCL do not provide a speedup in comparison to in sequential on the host code. There are many reasons behind these findings, and some of them are due to the limitations of OpenCL, and the nature of the problem.

5.1 Complexity of Algorithms

Sequential algorithms for statistical calculations are inherently fast; especially when using the algorithms we decided to use. As the sequential algorithms are online, we can run them with $O(n)$ time complexity. Getting a parallel algorithm that can achieve a better speedup is difficult, and with the addition of the overhead from transferring data to and from the host and device, parallel will nearly always be slower. With a more naive solution for the sequential algorithms, it may be possible that a speed up would occur.

5.2 Problem Structure

The given brief was to perform a reduction in parallel. OpenCL and acceleration devices typically are employed for vector-to-vector operations as synchronisation via barriers is not needed. The results clearly show that no speedup was gained from performing these simple calculations as a reduction in parallel. In order for parallelisation using OpenCL to be worth the effort, the computational effort during reduction would need to be greater (compared to the incremental case), or the problem would need to involve large vector operations.

5.3 Bandwidth and Data Transfer

According to Nvidia [5], bandwidth is one of the most important gating factors for performance. The significant limiting factor with OpenCL and the main reason why we didn't get a speedup is because of our data transfer rate which is limited by the bandwidth of the device. We attempted to isolate the transfer time by testing using a work group size of 1024, on a 10 million data point csv file while changing different parts of the code. Testing on this file, we found out that it usually requires 4 seconds to run, where 2 seconds of this are for the "base" of the run, while 1 second is used for transferring and another second for the kernel to execute. As mentioned in the profiling section, we had to use a more primitive form of timing resulting in these findings.

Another challenge we encounter was loading the data from a csv file and into an OpenCL en-queue buffer. Seeing as we needed to specify data-type both in the host code and in the kernel, we required our file to be parsed into that format. This process creates some overhead compared with loading the file straight into the kernels en-queue buffer. It is possible to load a host made buffer into an OpenCL en-queue buffer, but data-types and sizing prove to be another difficulty (`char` to `double`). If the initial read from file and transfer could be removed and put straight into the buffer, there may have been a substantial performance increase, which possibly could have surpassed the speed of sequential processing.

5.4 Representation Error

Representation error is not an issue to seven significant figures, and we probably achieve even more considering we compile for 64-bit systems and use doubles to represent each data point. The accuracy provided is almost unusable, statistics usually do not use such levels, and neither do most engineering applications. If more precision is required than speed is of a lesser priority, it's highly likely that our application can probably support this with some modification but with a performance penalty. Most numbers are not near the limit of 64-bit doubles either. Double precision gives typically 15 to 17 accurate significant figures, so by claiming 7, we are quite conservative based on our divisions, chunking and huge data sizes of billions of data points. NASA's JPL has an article about this [6].

5.5 Nvidia and OpenCL

We targeted both CPUs and GPUs with our OpenCL application, thus we naturally had to support Nvidia's processors. Nvidia's products do not support OpenCL nearly as well as AMD, nor Intel (including GPUs) so we were forced to use OpenCL 1.2. We used compiler directives such as "define #NVIDIA"which forced us to use version 1.2 if it was on a Nvidia system, and 2.0 on any other.

5.6 Linux vs Windows 10

Linux naturally reads faster than Windows, hence it can have a much lower run time of the sequential and parallel implementation. Windows was also unable to use a better token splitter that was provided on Linux (`strsep()`), and hence we had to settle for standard, more dangerous splitters, `strtok` and `strtok_r`. With our current file buffering and reading system, Windows cannot read certain file sizes.

We believe that this is the result of a 32-bit pointer overflow, as the number of characters read when the exit condition is reached is around 2.1 billion. This problem does not occur on Linux, and we believe this is due to the environment we used in Windows (MinGW), but we cannot be sure from our research. Compiling via the Visual Studio C++ Compiler would likely resolve this issue.

5.7 Profiling

Due to a lack of profiling tools (NVIDIA dropped support for nvvp, open-source profilers not working correctly), we could not correctly analyse our OpenCL run time, which meant we also could not identify statistics about timing, occupancy and bottlenecks.

6 Discussion & Conclusion

The OpenCL implementation for calculation of statistical values only provided us with a decrease in speed when compared to sequential. The problem we were solving is not suited to be processed in compute devices due to the vast amount of overhead when transferring data between the host and the device. Using OpenMP instead would likely provide a better performance increase than OpenCL due to not having a large overhead. Our initial sequential algorithms are also highly efficient so parallelisation without taking into account overhead does not provide much speedup. Both implementations process billions of data points in a matter of minutes which indicates that for most applications sequential may be good enough and Big-Data concepts such as MapReduce on distributed systems would have to be employed for larger data sets. Reduction problems such as statistical values do not fully saturate the acceleration devices resulting in poor occupancy. According to Stone et al., vector problems such as matrix multiplication or manipulation with large work item and work group size can fully saturate the devices and thus hide the effects of latency [7]. With more time we would have looked into further reducing the transfer time of data between the host and devices by looking further into the specification[8].

References

[1] P. Pébay, “Formulas for Robust, One-Pass Parallel Computation of Covariances and Arbitrary-Order Statistical Moments,” *Sandia Report*, vol. SAND2008-6, no. September, pp. 1–18, 2008.

[2] M. Choi and B. Sweetman, “Efficient calculation of statistical moments for structural health monitoring,” *Structural Health Monitoring*, vol. 9, no. 1, pp. 13–24, 2010.

[3] T. Chan, G. Golub, R. LeVeque, and S. U. C. D. o. C. Science., “Updating formulae and a pairwise algorithm for computing sample variances,” *Annals of Physics*, vol. 54, p. 258, 1979.

[4] K. Karimi, “The Feasibility of Using OpenCL Instead of OpenMP for Parallel CPU Programming,” *arXiv:1503.06532 [cs]*, 2015.

[5] NVIDIA, “OpenCL Best Practices Guide,” 2010.

[6] “How many decimals of pi do we really need? - edu news | nasa/jpl edu,” Mar 2016.

[7] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–72, 2010.

[8] Khronos Group, “The OpenCL Specification,” 2012.