Energy Efficient Energy Analytics

by

Sagnik De

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

© Sagnik De 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Smart meters allow for hourly data collection related to customer's power consumption. However this results in thousands of data points, which hides broader trends in power consumption and makes it difficult for energy suppliers to make decisions regards to a specific customer or to large number of customers. Since data without analysis is useless, various algorithms have been proposed to lower the dimensionality of data, discover trends (eg. regression), study relationships between different types (eg. temperature and power data) of collected data, summarise data (eg. histogram). This allows for easy consumption by the end user.

The smart meter data is very compute intensive to process as there are a large number of houses and each house has the data collected over a few years. To speed up the smart meter data analysis, computer clusters have been used. Ironically, these clusters consume a lot of power. Studies have shown that about 10 % of power is consumed by the computing infrastructure. In this thesis a GPU will be used to perform analysis of smart meter data and it will be compared to a baseline CPU implementation. It will also show that GPUs are not only faster than the CPU, but they are also more power efficient.

Acknowledgements

I would like to thank my adviser Dr W. Golab for his helpful advice in preparing the thesis. I would also like to thank my readers Dr. Mark Crowley and Dr. Patrick Lam for proofreading the thesis and providing helpful feedback. Also this thesis would not be possible without the NVidia grant program for generously donating a GPU (Tesla K40).

Dedication

I would like to dedicate this thesis to my parents for their help and support for which this thesis was made possible.

Table of Contents

List of Figures ix

1 Introduction 1

2 Background and Related Work 4

2.1 Graphics Processing Unit 4

2.2 GPU Software 5

2.3 GPU Hardware 9

2.4 Execution of software on the hardware 12

2.5 Branch and memory divergence 12

2.6 Related work 14

3 Experimental Setup 16

3.1 Software implementation 16

3.2 Benchmark validation 18

3.3 Power Measurement 19

3.4 Plots 19

4 Histogram Benchmark 22

4.1 Implementation details 22

4.2 Validation 23

4.3 CPU benchmarks . . . 23
4.4 GPU benchmarks . . . 26
4.5 GPU benchmark block version . . . 28
4.6 Power consumption . . . 31
4.7 Summary . . . 33

5 Similarity Search Benchmark **34**
5.1 Implementation details . . . 34
5.2 Validation . . . 36
5.3 Similarity CPU Benchmarks . . . 36
5.4 Similarity GPU Benchmarks . . . 38
5.5 Similarity Block GPU Benchmarks . . . 40
5.6 Power consumption . . . 42
5.7 Summary . . . 45

6 ParX Benchmark **46**
6.1 Implementation details . . . 46
6.2 Validation . . . 47
6.3 CPU Performance . . . 47
6.4 GPU Performance . . . 49
6.4.1 1 thread per house . . . 50
6.4.2 24 threads per house . . . 52
6.4.3 Different threads per house . . . 54
6.5 Power consumption . . . 57
6.6 Summary . . . 59

7 Energy Disaggregation Model Benchmark **61**

7.1 Implementation details 62

7.2 Validation 62

7.3 CPU benchmarks 63

7.4 GPU benchmarks 65

7.5 Power consumption 69

7.6 Summary and discussion 70

8 Discussion **72**

References **75**

APPENDICES **79**

A Parallel Reduction **80**

List of Figures

2.1 CUDA model showing 9 blocks of threads arranged in 2D grid and each block with 12 threads arranged in 2D. 7

2.2 An example of CUDA program executed on the CPU. 8

2.3 Diagram of Kepler hardware (K40) as seen on page 6 of [29]. 10

2.4 Diagram of a single SMX of K40 as seen on page 8 of [29] 11

2.5 An example of 2-way branch divergence. 13

2.6 An example of memory divergence. 13

3.1 Experimental setup. 16

3.2 Data extraction. 17

3.3 Sample ParX CPU plot. 19

3.4 Sample GPU plot for Energy Disaggregation. 20

3.5 Sample power consumption plot for ParX CPU-based implementation. . . . 20

4.1 Scalability of the extraction process with increasing thread count to a maximum of 8 threads. 24

4.2 Scalabilty of the process of generating a histogram with increasing thread count to a maximum of 8 threads. 25

4.3 Total speed up is shown with the increase of thread count to a maximum of 8 threads. 25

4.4 Speed up of GPU execution of extraction compared to single threaded CPU implementation. 26

4.5 Speed up of GPU execution of extraction compared to octa threaded CPU implementation. 26
4.6 Speed up of GPU execution of histogram compared to single threaded CPU implementation. 27
4.7 Speed up of GPU execution of histogram compared to octa threaded CPU implementation. 27
4.8 Speed up of GPU execution of computational time compared to single threaded CPU implementation. 27
4.9 Speed up of GPU execution of computational time compared to octa threaded CPU implementation. 27
4.10 Speed up of GPU execution of total time compared to single threaded CPU implementation. 28
4.11 Speed up of GPU execution of total time compared to octa threaded CPU implementation. 28
4.12 Speed up of GPU execution of histogram (block) compared to single threaded CPU implementation. 29
4.13 Speed up of GPU execution of histogram (block) compared to octa threaded CPU implementation. 29
4.14 Speed up of GPU execution of compute (block) compared to single threaded CPU implementation. 29
4.15 Speed up of GPU execution of compute (block) compared to octa threaded CPU implementation. 29
4.16 Speed up of GPU execution of total (block) compared to single threaded CPU implementation. 30
4.17 Speed up of GPU execution of total (block) compared to octa threaded CPU implementation. 30
4.18 Power consumption of the CPU implementation of histogram benchmark. . 31
4.19 Power consumption of the GPU implementation (1 thread/house) of histogram benchmark. 32
4.20 Power consumption of the GPU implementation (1 block/house) of histogram benchmark. 32
4.21 Breakdown of histogram execution times of 28165 houses. Time in seconds. 33

5.1 Similarity calculation speed up on a CPU with increasing thread count. . . 36
5.2 Similarity matching speed up on the CPU with increasing thread count. . . 37
5.3 Total matching speed up with CPU with increasing thread count. 37
5.4 Speed up of GPU execution of cosine similarity calculation compared to single threaded CPU implementation. 38
5.5 Speed up of GPU execution of cosine similarity calculation compared to octa threaded CPU implementation. 38
5.6 Speed up of GPU execution of similarity search compared to single threaded CPU implementation. 39
5.7 Speed up of GPU execution of similarity search compared to octa threaded CPU implementation. 39
5.8 Speed up of GPU execution of the computational workload compared to single threaded CPU implementation. 39
5.9 Speed up of GPU execution of the computational workload compared to octa threaded CPU implementation. 39
5.10 Speed up of GPU execution of total workload compared to single threaded CPU implementation. 40
5.11 Speed up of GPU execution of total workload compared to octa threaded CPU implementation. 40
5.12 Speed up of GPU execution of block version of the similarity compared to single threaded CPU implementation. 41
5.13 Speed up of GPU execution of block version of the similarity compared to octa threaded CPU implementation. 41
5.14 Speed up of GPU execution of block version of the computational workload compared to single threaded CPU implementation. 41
5.15 Speed up of GPU execution of block version of the computational workload compared to octa threaded CPU implementation. 41
5.16 Speed up of GPU execution of block version of the total compared to single threaded CPU implementation. 42
5.17 Speed up of GPU execution of block version of the total compared to octa threaded CPU implementation. 42

5.18 Power consumption when executing CPU implementation of similarity algorithm. . . . 43
5.19 Power consumption when executing GPU implementation (1 thread per pair of houses) of similarity algorithm. . . . 43
5.20 Power consumption when executing GPU implementation (1 block per pair of houses) of similarity algorithm. . . . 44
5.21 Breakdown of similarity execution times of 880 houses. Time in seconds. . . 45

6.1 ParX speed up with increasing thread count. . . . 48
6.2 Total speed up with increasing thread count. . . . 49
6.3 Speed up of ParX (1 thread per house) compared to the single threaded CPU implementation. . . . 50
6.4 Speed up of ParX (1 thread per house) compared to the octa threaded CPU implementation. . . . 50
6.5 Speed up of compute (1 thread per house) compared to the single threaded CPU implementation. . . . 51
6.6 Speed up of compute (1 thread per house) compared to the octa threaded CPU implementation. . . . 51
6.7 Speed up of total execution time (1 thread per house) compared to the single threaded CPU implementation. . . . 51
6.8 Speed up of total execution time (1 thread per house) compared to the octa threaded CPU implementation. . . . 51
6.9 Speed up of ParX (24 threads per house) compared to the single threaded CPU implementation. . . . 52
6.10 Speed up of ParX (24 threads per house) compared to the octa threaded CPU implementation. . . . 52
6.11 Speed up of compute (24 threads per house) compared to the single threaded CPU implementation. . . . 53
6.12 Speed up of compute (24 threads per house) compared to the octa threaded CPU implementation. . . . 53
6.13 Speed up of total execution time (24 threads per house) compared to the single threaded CPU implementation. . . . 53

6.14 Speed up of total execution time (24 threads per house) compared to the octa threaded CPU implementation. 53
6.15 Speed up of ParX (3520 houses, multiple threads per house) compared to the single threaded CPU implementation. 54
6.16 Speed up of ParX (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation. 54
6.17 Speed up of compute (3520 houses, multiple threads per house) compared to the single threaded CPU implementation. 55
6.18 Speed up of compute (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation. 55
6.19 Speed up of total execution time (3520 houses, multiple threads per house) compared to the single threaded CPU implementation. 55
6.20 Speed up of total execution time (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation. 55
6.21 Speed up of ParX (28165 houses, multiple threads per house) compared to the single threaded CPU implementation. 56
6.22 Speed up of ParX (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation. 56
6.23 Speed up of compute (28165 houses, multiple threads per house) compared to the single threaded CPU implementation. 56
6.24 Speed up of compute (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation. 56
6.25 Speed up of total execution time (28165 houses, multiple threads per house) compared to the single threaded CPU implementation. 57
6.26 Speed up of total execution time (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation. 57
6.27 ParX CPU power consumption with increasing thread count. 58
6.28 ParX GPU power consumption (1 thread/house) with increasing thread count. 58
6.29 ParX GPU power consumption (24 thread/house) with increasing thread count. 59
6.30 Breakdown of ParX execution times of 28165 houses. Time in seconds. . . 59

7.1 Extraction speed up. 63
7.2 Sorting speed up. 64
7.3 Coefficient generation speed up. 64
7.4 Total speed up, labels indicate the number of threads. 65
7.5 Speed up of GPU execution of extraction compared to single threaded CPU implementation. 66
7.6 Speed up of GPU execution of extraction compared to octa threaded CPU implementation. 66
7.7 Speed up of GPU execution of sorting compared to single threaded CPU implementation. 67
7.8 Speed up of GPU execution of sorting compared to octa threaded CPU implementation. 67
7.9 Speed up of GPU execution of coefficient generation compared to single threaded CPU implementation. 67
7.10 Speed up of GPU execution of coefficient generation compared to octa threaded CPU implementation. 67
7.11 Speed up of GPU execution of compute compared to single threaded CPU implementation. 68
7.12 Speed up of GPU execution of compute compared to octa threaded CPU implementation. 68
7.13 Speed up of GPU execution of total execution compared to single threaded CPU implementation. 68
7.14 Speed up of GPU execution of total execution compared to octa threaded CPU implementation. 68
7.15 Power consumption of CPU in energy dissagregation. 69
7.16 Power consumption of GPU in energy dissagregation. 70
7.17 Breakdown of Energy Dissagregation execution times of 28165 houses.Time in seconds. 70

A.1 Parallel reduction stages 80

Chapter 1

Introduction

Global warming has caused changes in weather patterns, loss and destruction of habitat [25] [20], loss in human productivity [11], and negative impacts on food production, as well as spread of plant diseases and pests [34]. One of the main causes for the global warming is the increased presence of CO_2 and methane in the atmosphere, the two most important green house gases. CO2 is generally caused by the use of fossil fuels. Major sources of the CO_2 production are electrical power generation [30] [2] and the combustion of vehicular transport. Methane is generally produced by the agricultural industries [16].

To reduce usage of fossil fuels like coal, gas and oil in power stations, governments have pursued other sources of power generation like nuclear power, solar, wind, hydro and geothermal. Unfortunately each of the power sources has its own disadvantages. Solar power suffers from poor efficiency. Wind power causes noise pollution [35]. Hydro-electric power damages eco-system [10]. Nuclear power suffers from high cost [28].

Since it is difficult to generate electricity in a sustainable way, governments have tried to both move away from fossil fuels and reduce energy consumption. The problem of reducing electrical consumption has been approached in many ways, including reducing peak load, which determines the scale and size of power generation infrastructure. It involves improving electricity generation, transmission and distribution. At off-peak times the electric grid is running underutilized to a large extent. The peak load on the industrial side and consumer side can be different. Industrial peak load happens during business hours like car manufacturing, manufacture of electronics etc. On the consumer side, in the summer the peak load is during the afternoon when everyone has turned on their air conditioner. The load also spikes in the evening as many people turn on their television, or at lunch time due to the usage of electrical cooking appliances. But the peaks due to

TV/electronics and cooking are generally smaller than A/C usage.

If peak load is reduced, the electricity providers can have a lighter and more efficient grid. They can install batteries to deal with the peak load or they can use a variable pricing structure which disincentivizes power usage during peak hours. The problem remains that batteries are expensive and time of day pricing has not reduced peak power consumption significantly [19]. Nevertheless, energy suppliers started investing in smart meters. Smart meters can provide insight to the problem by providing fine grained electricity consumption data. Smart meters collect time series data about the power consumption of each individual customer. The data can be analyzed to provide deeper understanding on the power consumption.

The analysis of smart meter data is computationally intensive, and therefore time consuming on general purpose hardware. To speed up the calculation, smart meter data computation can be distributed over a cluster of computers and as a result a large amount of energy will be used to power the cluster. It will be ironic if the analysis of the power consumption data which will be used to reduce energy consumption incurs a high level of power consumption. Previously this was not a problem as software inefficiencies would be masked by newer and more efficient hardware due to the presence of Moore's law. Unfortunately creation of new process technologies has slowed down.Distributed data processing frameworks like Spark and Hadoop are also inefficient compared to centralized ones for data sets that fit inside a single computer. In this case one can try systems which scale up instead of scaling out. In the quest for improving energy efficiency one can even go further and use some form of fixed function special purpose hardware ASICs (application-specific integrated circuit) and other types of silicon like FPGAs (field-programmable gate array).

The problem with such an extreme level of optimization is that the tool chain for such systems (software and hardware ecosystems) are not well developed. It is also extremely labour-intensive to program them. Also if some modification is needed then one has to do significant reprogramming. Hence the goal of this thesis is to explore a programming model which has a mature tool-chains and is purpose built for very high levels of parallelism. Therefore the graphics processing unit (GPU) natural choice. Also, GPUs are commercially available and hence the barrier of entry is much lower in programming them. As far as the software stack is concerned, CUDA is selected for use as it is specifically built for general purpose graphics processing (GPGPU) usage. There are obviously other platforms too like OpenCL and compute shaders in both OpenGL and Direct3D. OpenCL

is an open standard counterpart of CUDA. Since OpenCL is newer than CUDA, parts of the tool-chain are not yet very well supported. It is possible to get into technical problems when using it. It seems that some parts of the features are not supported by the hardware even when it is supposed to work.

So in this thesis CUDA will be used for programming the GPU (NVidia Tesla K40). The performance of analytical computations of smart meter data with GPU will be compared to a CPU implementation. The results will also validate the usage of GPU for smart meter data analysis even though some of the benchmarks do not map well to the CUDA programming model easily.

In the next chapter, GPUs will be explained and expanded upon in regards to the hardware and software stack, and also the prior work related to energy analytics will be discussed.

Chapter 2

Background and Related Work

2.1 Graphics Processing Unit

Computer graphics, or more specifically 3D graphics, was pretty well-known in the commercial and military circles before the advent of dedicated pieces of silicon (also known as GPUs) to process 3D geometry. Derived from IrisGL is the very well-known and one of the most important APIs, namely OpenGL. This allowed software developers to target a single API instead of a multitude of different proprietary APIs. As far as dedicated hardware for graphics is concerned there was no real disruption when GPUs actually came into use. In its early days a large part of the 3D graphics work was done on a CPU then the results of that were sent to the screen. The first true 3D graphics cards were video display controllers. Later on they gained more functionality and were simple 'Image Processors' with very little programming capability. At the turn of the 21st century the GPU became more programmable. The two important programmable stages put there were the vertex shader and the fragment shader (pixel shader in Direct3D). These were written as assembly programs at first and later on in a variant of C. The vertex shader is used to modify the vertices of the triangles that make up the 3D geometry to be projected on the screen. The fragment shader is used to colour the triangles depending upon textures.

As far as the GPGPU (general purpose graphics processing unit) is concerned, the fragment shader however can be re-purposed to read from some textures to do some calculation and then write the result to an off-screen render target. A very early implementation of this idea was with matrix multiplication [23] with bytes (color data only supported byte operations at first). Later on matrix multiplication with floating point numbers [27] was

implemented when the GPUs started supporting them. As matrix multiplication is a very common operation in scientific community, various studies were performed [21] [13] for better understanding of graphics hardware and for improving the performance of the implementations.

These early implementations suffered from problems like low precision [18] [9] and non-compliant hardware (i.e, operations on the GPUs were not IEEE compliant). As a result, many of the implementations were not really direct replacements of the CPU version but complemented it. The low cost of commodity GPUs and the speed at which the results were obtained made the challenge of programming on GPUs worthwhile. The matrix operations also suffered from poor memory utilization [13] as the caching on GPUs was not optimized for the memory access patterns for multiplying matrices.

To solve performance problems related to memory access, an indirection texture is used. A texture is a 2D array but is used to store image data on the GPU for displaying on the screen. These indirection textures function like a hashtable. Let *mat* be a texture and suppose that one needs to access the value of ith row and jth column. Normally what one does is $val = mat[i, j]$. However when an indirection is used, what is being done here is $val = store[indirect1[i, j], indirect2[i, j]]$. The texture *store* keeps the values of the matrix and *indirect1* and *indirect2* are two textures that determines the row and column addresses for *store*. This is used to keep the memory accesses cache friendly. Sparse matrix solving [6] used it. Later other matrix operations also came into light like LU factorization [15] which involved using the rasterizer and switched rows and columns to improve memory performance. As a result, in their early days GPU programming techniques were esoteric and time consuming.

These methods became outdated with the introduction of DirectX 9 and OpenGL 3.0 which had new requirements related to the programmability of shaders. nVidia, a GPU vendor during this time, introduced a C dialect for easier programming called CUDA. This allowed general purpose algorithms to take advantage of the graphics hardware without doing all the difficult index manipulations in the algorithm implementations.

2.2 GPU Software

Nowadays there is a large number of APIs and programming languages which can interact with the GPU. Graphics related APIs like OpenGL and Direct3D (D3D) contains 'Compute Shaders' as a part of the graphics pipeline which can be used for doing purely computational work. Unfortunately these are far too entwined with the graphics pipeline to be sufficiently

useful for non graphical workloads. They are also limited in feature set as these APIs need to work on a wide range of hardware. For example, Compute Shaders 5.0 (CS 5.0) of D3D does not allow the number of threads to be set at runtime, but at compile time. CS 5.0 has not been updated from 2011 and as a result, it is feature incomplete compared to CUDA. OpenGL is not well supported in either Windows 8.1 or Ubuntu Linux 16.04 by the graphics drivers on which the benchmarks in this thesis are run. There is also OpenCL, which is an open counterpart of CUDA, but since some parts of the tool chain remain unpolished, CUDA is being used in this thesis.

CUDA compiler (CUDA version 7.5 at the time of writing) supports nearly all constructs of C++11 (except for standard library). Here CUDA will be expanded upon as CUDA has some non-standard extensions for GPU-specific functionality. To execute any form of operation on the GPU, the programmer will have to perform five operations:

- Allocate memory on the GPU.
- Copy data from the RAM to the VRAM (video RAM). In CUDA terminology copy from host to device.
- Launch 'kernel' (the program which runs on the GPU).
- Copy data back from VRAM of the GPU to CPU.
- Free memory on the GPU.

In CUDA 6.0 and above one does not have to allocate and free memory explicitly but the GPU driver will do so on its own if the programmer uses the CUDA-specific memory allocation API. Unfortunately this results in unstable behaviour and does not play well with the benchmarking process as required by the thesis. Therefore, it has not been used.

The GPU works on the principle of hierarchial parallelism. Threads on a GPU are not launched individually, instead groups of threads called thread block (or blocks in short) are launched. Each block will have the same number of threads. Each thread executes the same program, called a 'kernel'. The Figure 2.1 shows the relationship between blocks and threads. The blocks and threads inside the block can be organized logically in 1D, 2D or 3D grid, though the figure shows only a 2D arrangement of blocks and threads.

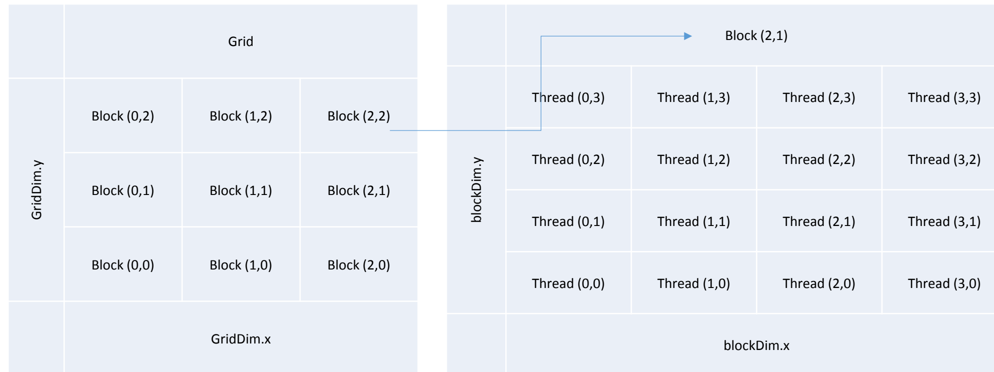


Figure 2.1: CUDA model showing 9 blocks of threads arranged in 2D grid and each block with 12 threads arranged in 2D.

It may be noted that there is no requirement that blocks and threads have to have the same dimensionality. For example blocks can have a 1D arrangement and the threads have a 3D arrangement inside each block. These arrangements are only a software feature, and everything is flattened out in a 1D arrangement when executing on the hardware.

```
1  __device__ float square(float num){ //device function
2          auto res = num*num;
3          return res;}
4  __global__ void computeSquareArrayGPU(float* source,float* target,int size){
   //kernel
5          auto tid = threadIdx.x; //thread index inside the block
6          auto bid = blockIdx.x; //block index
7          auto blockSize = blockDim.x; //Dimension of the block
8          auto globalid = blockSize*bid+tid;
9          if(globalid>=size){return;}
10         target[globalid] = square(source[globalid]);}
11 //CPU side work
12 void cpusidecomputesquareArray(float* source,float* target,int size){
13         auto bsize = 128;     //size of block
14         auto numblocks = ceil(size/bsize); //number of blocks
15         float* dsource; float *dtarget;
16         cudaMalloc(&dsource,sizeof(float)*size);//Allocate memory
17         cudaMalloc(&dtarget,sizeof(float)*size);
18         //Copy to GPU
19         cudaMemcpy(dsource,source,cudaMemcpyHostToDevice);
20         computeSquareArrayGPU<<<numblocks,bsize>>>(dsource,dtarget,size);//
21         cudaDeviceSynchronize();
22         //Copy to CPU
23         cudaMemcpy(target,dtarget,cudaMemcpyDeviceToHost);
24         cudaFree(dsource);cudaFree(dtarget);} //Free memory
```

Figure 2.2: An example of CUDA program executed on the CPU.

The program in Figure 2.2 takes two arrays (source and target) and squares the values of source and puts them in target. The function which will be executed on a GPU is called a 'kernel' and is preceded by the '__global__' keyword and always has the return type 'void'. Any function that needs to be called from the kernel (and executed inside the GPU) is preceded by the '__device__' keyword. The example kernel finds the square of the numbers in an array and puts them in another array. On the CPU side, the values need to be copied to the GPU and then back to get the results. The GPU kernel is run by specifying the number of blocks and the number of threads in each block using angled brackets.

There are some restrictions on the threads and blocks:

- A block can have maximum 1024 threads. This means $blockDim.x \times blockDim.y \times blockDim.z \leq 1024$. Also the blockDim.x and blockDim.y can have range from 0-1023 but blockDim.z can only have range of 0-63.
- The thread index of a thread inside the block will uniquely identify each thread inside a block and is always lesser than the block dimension.
- Similar to threads, the blocks can have block ID range 0 to 2 billion in X axis and 65535 in Y and Z axis.
- The threads in a block can communicate with each other with the use of shared memory and there are mechanisms for them to synchronize the threads inside a block. However there is no way of synchronizing threads between different blocks.

2.3 GPU Hardware

The GPU hardware contains a lot of different parts: some of them fixed function and others are programmable. Here only the programmable parts and parts related to GPGPU will be discussed.

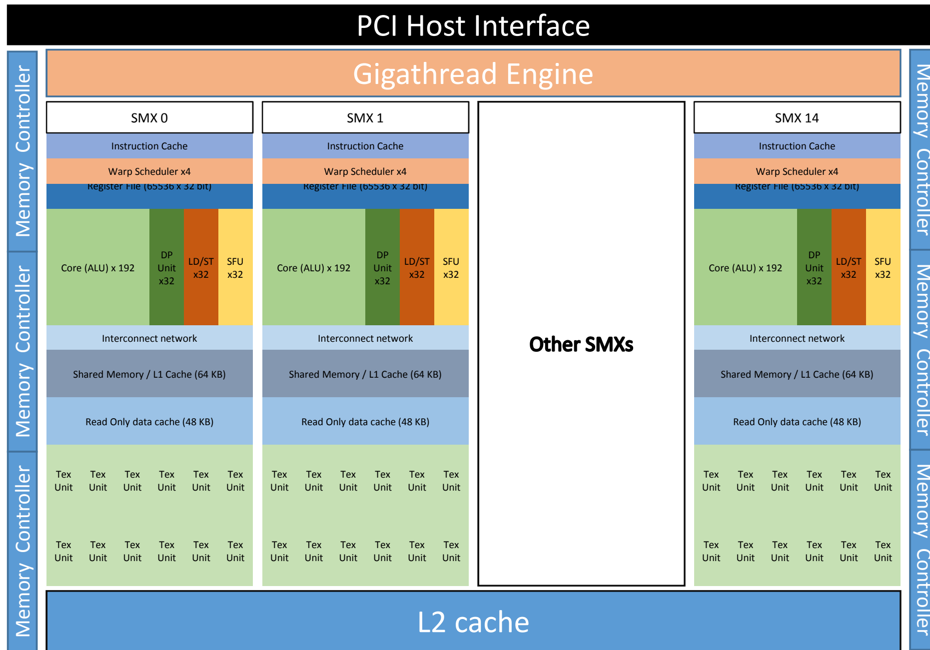


Figure 2.3: Diagram of Kepler hardware (K40) as seen on page 6 of [29].

The GPU hardware works on the principle of hierarchical parallelism. In Figure 2.3 one can see that the GPU is divided into multiple hardware blocks called SMX (Symmetric Multi-processor) in nVidia's Kepler architecture. The SMXs are tied to the memory controllers and L2 cache by the means of a crossbar. There are also the PCIe controller, and other fixed function hardware like the tessellator (not shown in diagram). The GPU has its own VRAM (Video RAM) called GDDR5. Before any form of computation can take place the data is streamed through the PCIe bus to the VRAM. There are 15 SMXs in the K40 GPU.

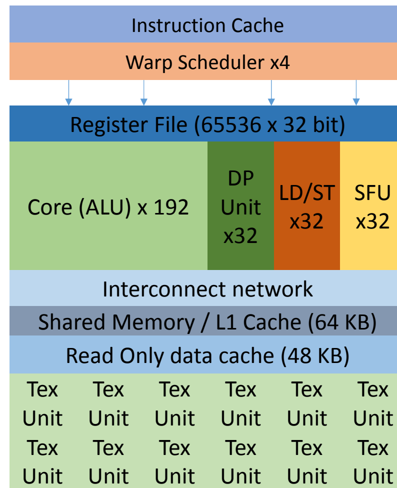


Figure 2.4: Diagram of a single SMX of K40 as seen on page 8 of [29]

Each SMX (Figure 2.4) of K40 contains 192 SIMD lanes (called CUDA cores by nVidia). Each SMX has 48 KB of 'shared' memory (the memory is local to each SMX but is called shared as it is shared by the threads running on the SMX). The SMX contains double precision units and floating point units at a ratio of 1:3. It also contains special function units (SFU). The SFUs are responsible for functions like log, inverse square root, minimum, maximum etc. There are also some fixed function units like the texture units (not used in the benchmarks). LD/ST are the load-store units responsible for address generation. The generated loads are coalesced into blocks of 128B before they are sent to the L1 cache. An important and interesting part of the SMX is the warp scheduler. In a CPU each thread executes on its own and scheduled by the OS, but that is not how it executes on a GPU. The GPU threads are executed in groups of 32 threads called 'warps' in lock-step and are scheduled by the warp scheduler. They are executed in an inorder fashion, i.e., if one of the threads gets stalled then all the threads get stalled. If a thread in a warp becomes stalled then the warp scheduler will dispatch another wave of 32 threads in an effort to hide the latency and improve performance.

One can see a massive register file in the picture of the SMX. Unlike a CPU core, the warp scheduler does not save registers when switching out a thread. Instead, it uses another part of the register file to execute the next wave of threads. If it runs out of register space, then it stalls dispatching new threads. The hardware is limited to 2048 threads so the warp scheduler will only have 64 waves of threads to execute. Any form of limit, be it

register space, shared memory space or maximum number of threads in SMX, will result in stalling of execution of the threads.

The SMX is the GPU's closest analogue to a CPU core and has all the logic in execution of the threads in the core. Unlike a CPU where an OS schedules the threads, the hardware itself does the scheduling. The SMX is also subdivided into SIMT (single instruction multiple thread) cores. Each SIMT core has got 32 ALU lanes, which is why the warp scheduler executes threads in blocks of 32. Thus we have a 1:32 ratio of control logic to the compute logic. Contrary to modern out of order CPUs where a large part of the silicon budget is used for scheduling, register renaming and in re-order buffers, the silicon on GPUs is mostly devoted to ALUs. Also unlike in a CPU, the latency of the operations is long: a single FMA (fused multiply add) on GPU will take about 20 cycles but on a CPU it is only 5 cycles. The memory operations are longer at 300-400 cycles compared to 100-200 of a CPU and also lower clocked at 745 MHz (K40's base clock) core clock compared to 3.9 GHz for Haswell (i7-4770k). These are the reasons why given a simple parallel workload the GPUs consume less power than CPUs when processing the same workload.

2.4 Execution of software on the hardware

A single block of the grid always maps to a single SMX (i.e., a single block cannot execute some threads in one SMX and the other ones in a different SMX). The number of blocks must be greater than the number of SMXs in order to keep the GPU busy. Each block of threads no matter what size is broken down into groups of 32 and is dispatched to the warp scheduler to the execution units. If the number of threads in a block is not a multiple of 32, say 50 threads, then execution units will remain unused. If the number of threads in a block is fewer than 32, there is no need to use any synchronization primitives as the threads will be executed in lockstep. It also means that blocks with less than 32 threads will leave the GPU execution units under-utilized. In case of branch divergence the program counters of the diverging threads are left unmodified and they are executed later.

2.5 Branch and memory divergence

Branch divergence is an important issue in the context of GPUs. The kernel shown in Figure 2.5 is an example of branch divergence. The kernel will run half as fast as a

non-diverging kernel. The slowdown is caused by the fact that GPUs operate in warp granularity as far control flow is concerned but branch divergence means that only a few threads can be executed in a warp.

```
1  __global__ void (int* getarg){
2          auto tid = threadIdx.x;
3          auto bid = blockIdx.x;
4          auto blockSize = blockDim.x;
5          auto globaltid = blockSize*bid+tid;
6          auto gid = globaltid;
7          if(tid % 2 == 0){
8                  getarg[gid] = gid;
9          }else{
10                 getarg[gid] = tid;
11         }
12 }
```

Figure 2.5: An example of 2-way branch divergence.

```
1  __global__ void (int* getarg,int* hashtable,int* source){
2          auto tid = threadIdx.x;
3          auto bid = blockIdx.x;
4          auto blockSize = blockDim.x;
5          auto globaltid = blockSize*bid+tid;
6          auto gid = globaltid;
7          getarg[gid]= source[hashtable[gid]]; // Memory divergence
8  }
```

Figure 2.6: An example of memory divergence.

There are also other form of divergences like memory divergence as seen in figure 2.6. Here consecutive threads access very different parts of memory (if the values of hashtable are far apart) and thus the memory accesses cannot be coalesced together. This results in poor utilization of the memory bus as the memory from the VRAM is accessed in blocks of 128B.

2.6 Related work

Green computing refers to methods that reduce the detrimental effects of electronic devices on the environment. There are two parts to it, one related to disposal and production of electronics and the other related to the power usage of the said electronics. This thesis will focus on the latter. Most of the power reduction research was done with respect to warehouse computing, cluster computing, server farms etc. [4] [3]. Increased power consumption leads to larger amount of cooling required and the reliability of the server also goes down with increased temperature [31] [12]. Most of the techniques are based on scheduling and allocation of computing resources. Later on energy-efficient algorithms [33] are looked at. These are software implementations of algorithms which produce the same answer as conventional optimizations and are done to improve energy efficiency without degrading QoS (quality of service).

Different algorithms have been proposed for analysis of power consumption data [32] [1] [7] [36]. Some of them are related to finding out trends [14], summarising data (like histogram) and prediction [37]. The power consumption data can be high frequency (one data point every second) or at a more sedate rate (at about one data point per hour for smart meter). In this thesis the hourly smart meter data will be analysed.

This thesis marries the two concepts of energy effficient computing and energy analytics. The benchmarks chosen for the thesis are defined in the paper 'Smart Meter Data Analytics: Systems, Algorithms, and Benchmarking' [24]. The four benchmarks introduced in that paper are:

- Histogram: This benchmark calculates the minimum, the maximum and then the histogram of power data for each customer.
- ParX: This benchmark performs periodic auto-regression of the power data. The regression is performed on a per-hour basis. As there are 24 hours in a day the algorithm is used 24 times for each house. To perform the regression a order of three hours is used and the regression coefficients are calculated.
- Similarity search: This benchmark computes the cosine similarity of the time series data of the power consumed in each house. Then for each house the top ten similar houses are output.
- Energy dissaggregation model [5]: This performs disaggregation of power data depending on the temperature. For a given temperature the 10th and the 90th percentile power consumption is selected and regression is performed on them. In most of the

houses what one sees is that both the 10th and 90th percentile values follow a U-shaped curve. At lower temperatures the power consumption is high in the context of using heaters and in higher temperatures the power consumption is also high on account of using air-conditioners. These two gradients at the lower and higher temperatures are called the cooling and heating gradients. The original benchmark performed segmented linear regression (called 3-line). In this benchmark polynomial regression is performed instead. The details are provided in the Energy disaggregation benchmark chapter. To find the gradients the derivatives are taken at low and high temperature regions.

The next chapter discuses the experimental setup and the test environment.

Chapter 3

Experimental Setup

In this section specific hardware and the software configuration will be explained to enable reproduction of the experimental results. The following configurations for benchmarks are used:

System Configuration	
CPU	i7-4770K (Haswell) 3.5 Ghz, 8 MB L3
Motherboard	Gigabyte Z87X-UD3H
RAM	2×8 GB (1600 Mhz)
OS	Ubuntu Linux 16.04
GPU	Nvidia Tesla K40
CUDA version	7.5
Power Meter	Brand Electronics Model 20-1850C

Figure 3.1: Experimental setup.

The system is left in its default configuration. No changes related to power or performance configuration are performed to the system (like disabling turbo-boost etc.).

3.1 Software implementation

For a given benchmark the data is uploaded (as shown in Figure 3.2) into RAM as a single array. The data set is large (10 GB) as it contains 28165 houses with each house having less than a week to year's worth of data. There are also an array of offsets which provides the

index for the starting point and ending point of the data for each house. The information is kept in a packed binary format. Relevant field is extracted (here temperature data is shown) before any form of benchmark is run. There is 1 to 1 mapping between binary data and extracted data (i.e., temperature at array index 3 comes from extracting binary data at array index 3). Thus the array offsets are reused to locate the data for each house when the benchmark is run. All the CPU implementations use threads pinned to specific cores.

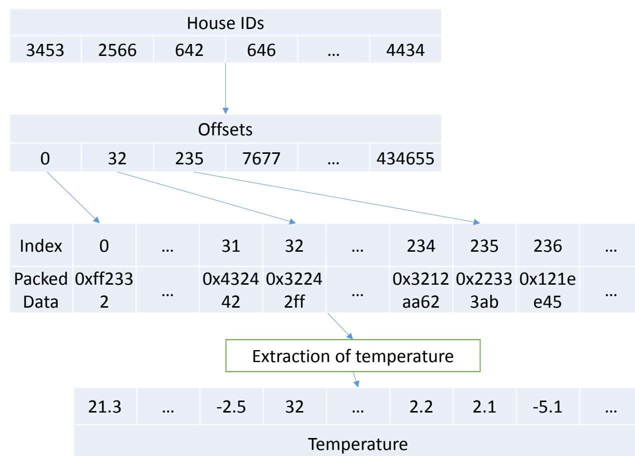


Figure 3.2: Data extraction.

In general GPU based implementation of the benchmark has the following parts:

1. Transfer data to GPU. (Not applicable in CPU implementation.)
2. Extract relevant data.
3. Execute benchmark workload.
4. Retrieve data from GPU. (Not applicable in CPU implementation.)

Similarly one can keep track of the execution times of these parts:

1. Pure benchmark execution (part 3 only).
2. Pure computational execution time. This consists of extraction time + benchmark execution time (2+3). This is not applicable for the CPU implementation.
3. Total execution time. Total execution time taken (i.e. 1+2+3+4). Since part 1 and 4 does not exist for the CPU, computational time and total time is equal in CPU implementation. This makes computational execution time redundant for CPU implementation.

Now GPU comparisons are made against the CPU implementations (single-threaded and octa-threaded) of these parts. Since there is no transfer of data in the CPU implementation, both computational execution time and total execution time on GPU is compared to that of total execution time of CPU.

The extraction performance is tested in Histogram and in Energy Disaggregation benchmarks. This is due to the fact that Histogram, ParX and Similarity extract the same field (power data) for the benchmarks. It is done again in Energy Disaggregation benchmark as it needs two fields (temperature and power data) for its algorithm.

Some benchmarks have a single stage and others have multiple stages of computation. Thus benchmarks like Histogram and ParX have a single kernel while Energy Disaggregation and Similarity have two kernels. Proper breakdowns of the kernels are provided in each benchmark chapter.

3.2 Benchmark validation

To ensure that the benchmarks are providing the right results, the results of the benchmarks are validated against Matlab implementations which was authored by Liu et al. [24]. Since results involving floating point numbers depend upon the order of the operations, the results from the benchmarks and the original Matlab implementations did differ slightly. The CPU and the GPU implementations compute similar results to the originals and are discussed further in each benchmark chapter. The main difference is that Energy Disggregation uses a different regression model for calculating the coefficients. In this case the intermediate results of some of the houses are collected and then regression is performed on Mathematica. This is used to validate the results of the Energy Disaggregation benchmark. Each benchmark is run five times and results are presented with errorbars when the variation of data is large enough to warrant it.

3.3 Power Measurement

The power meter provides a time series data of the power consumption. Thus in order to measure the total energy used the area under the curve of the time series data is calculated. The power meter samples the power consumption data by measuring power data at an interval of 100 milliseconds. According to the manual of the power meter, the error of the power meter is a maximum of:

1. 1 Watt
2. 1.5% of the reading
3. 2 units in the least significant digit (e.g., 0.2 Watts if the reading is 68.5 Watts)

However in the experiments the temporal resolution (100 milliseconds) is a bigger source of error, especially for short benchmarks). The 1.5% error in the reading is the only error affecting the power consumption results as most of the power consumption results are greater than 66W.

3.4 Plots

There are three types of plots used in the thesis: one shows CPU performance, one shows GPU performance and the last one shows power consumption.

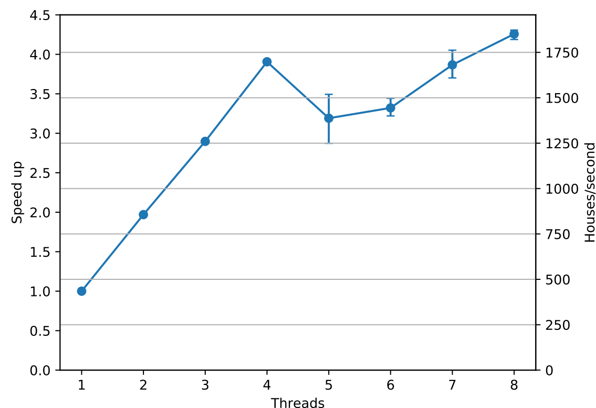


Figure 3.3: Sample ParX CPU plot.

The plot in Figure 3.3 shows the speed up of the algorithm with increasing thread count on a CPU implementation. The left y-axis shows the relative speed up with respect to single threaded implementation and the right y axis shows the absolute speed. The number of houses has been kept constant for all the CPU implementations.

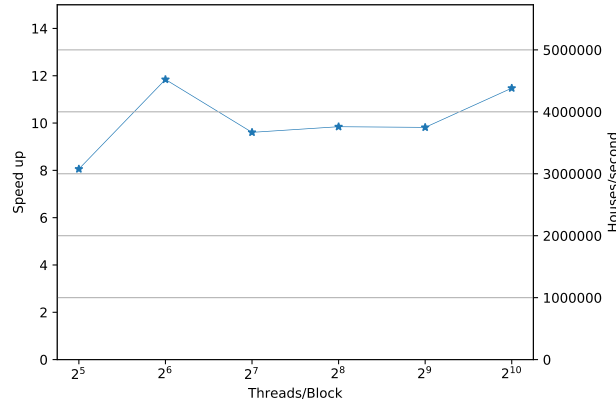


Figure 3.4: Sample GPU plot for Energy Disaggregation.

The plot in Figure 3.4 shows the speed up of the algorithm with increasing number of threads/block with respect to an octa-threaded CPU implementation. The left y-axis shows the relative speed up and the right y axis shows the absolute speed. The lines represent relative speed up and the stars represent absolute speed up. The GPU results are very stable with standard deviation less than 1.5%, so no error bars are shown.

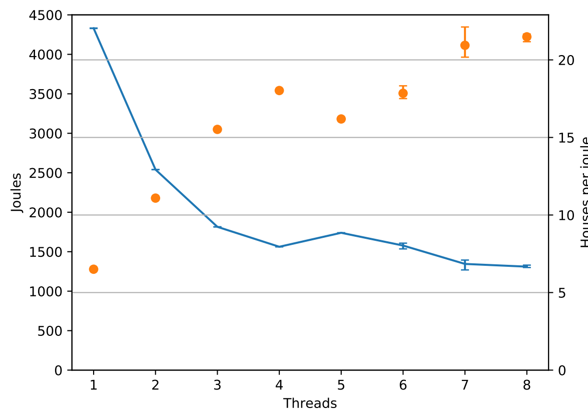


Figure 3.5: Sample power consumption plot for ParX CPU-based implementation.

The plot in Figure 3.5 shows the variation of the power consumption with increasing thread count for a fixed number of houses. The blue line corresponds to the left y-axis and the orange dots correspond to the right y-axis. These are inversely related. The left axis represent the energy consumed and the right y-axis represents the number of houses processed per joule.

Chapter 4

Histogram Benchmark

The histogram is the smallest benchmark of the set of benchmarks. It has been examined many times before [17][22][8]. Here it provides the customer with the distribution of the power consumption values obtained from smart meter data.

The general method of the histogram generation is as follows:

1. The minimum and maximum of the power consumption value are found, which were named *minp* and *maxp*.
2. The space between *minp* and *maxp* are divided into a number of bins *Bins* with each bin has size $binsize = \frac{maxp - minp}{Bins}$. Each bin is assigned zero at first.
3. Depending on the power consumption values the value in each bin is updated.

4.1 Implementation details

The core algorithm for both the CPU and GPU's 1 thread per house implementation is exactly the same. The only difference is that a single CPU thread loops through multiple houses but a GPU thread works on a single house. The core algorithm operates on a single house. As explained in Chapter 3, the *offsets* array is used to locate the position of power data for a given house index. The histogram is calculated in two phases. In the first phase all the values present in the power consumption data are looped over and the minimum

and maximum values of the power data are obtained. The histogram calculation is done in the next phase. In this phase the power consumption values are looped over again and depending on the energy values, the histogram is generated. The histogram is put in an array called *powerhist*.

The histogram block version (one block of threads operates on a single house) is different, each thread in a block calculates the partial minimum and the maximum values which puts them in the shared memory. Then a parallel reduction (see Appendix A) is performed to find out the maximum and minimum values of the power consumption data. For generating the histogram, the threads of the block divide their workload and create partial histograms. The partial histograms are then summed up using the shared memory by looping through the partial histograms. The results are stored in an array.

4.2 Validation

For testing the correctness of the workload, the results are compared against the original Matlab implementation [24]. Since floating point results depend on the order of operartions as explained in Chapter 3, the results obtained are slightly different. To check for differences, the histogram of each house is assumed to be a vector. For each house L_∞ [1] norm of the difference in the vectors of the benchmark and the Matlab implementations were tested. The values of the L_∞ never exceeded one for most of the houses. The L_∞ norm of a few pathological cases did reach two. The discrepancies appear when power consumption values falls on the boundary (or very close to the boundary) of two histogram bins, and the Matlab implementation and the code generated by the CUDA compiler tend to disagree in which bin to put them.

4.3 CPU benchmarks

At first one must touch upon extraction of data. The data extraction is done as all the fields of the dataset is not required for a specific algorithm and hence only the relevant field is extracted. The file where the data is stored is a csv file. To get the power data one has to parse the power data from the text file. This is computationally expensive and is not related to the efficiency of the algorithm. To avoid that issue the csv data is converted into a binary format and then the relevant data is extracted from the binary data before

[1] L_∞ is the maximum of the absolute value of all the components the vector

execution of any algorithm. The extraction of data is orthogonal to the number of houses. The binary data is divided into equal parts and is fed to each CPU thread. The *offsets* array will be used to determine which data represents which house later in the benchmarks. The figure 4.1 shows the extraction speed up for CPU as core count is increased for 28165 houses.

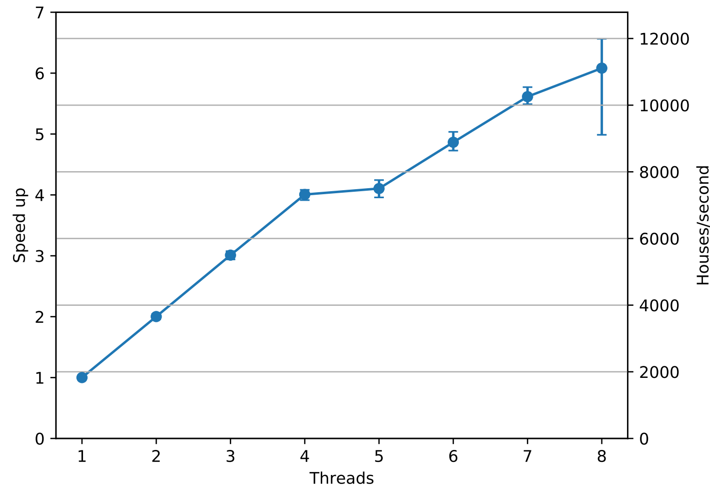


Figure 4.1: Scalability of the extraction process with increasing thread count to a maximum of 8 threads.

Histogram execution time is shown in figure 4.2. One can notice a near perfect scaling from 1 to 4 threads and then sub-linear scaling from 5 to 8 threads. The speed up is shown for 28165 houses.

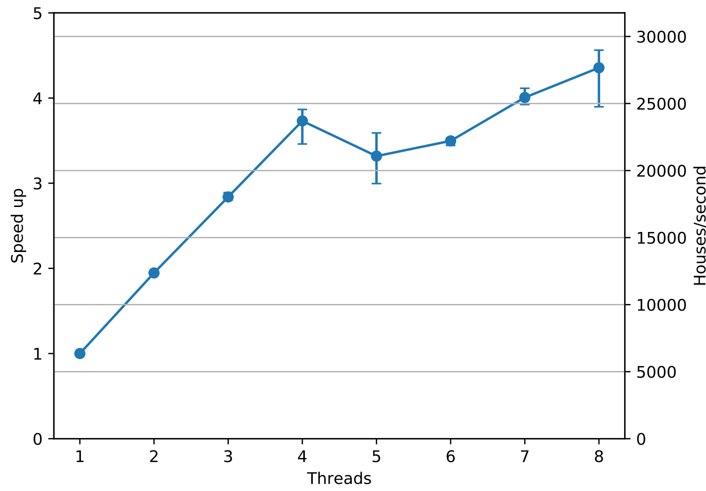


Figure 4.2: Scalabilty of the process of generating a histogram with increasing thread count to a maximum of 8 threads.

In the Figure 4.2 there is perfect scaling up to four cores and then there is a performance regression as more threads are scheduled on the same physical CPU. The threads are pinned to the cores in this benchmark. Unpinning the threads improves performance at higher thread count but brings too much variation with variation up to 50% at times.

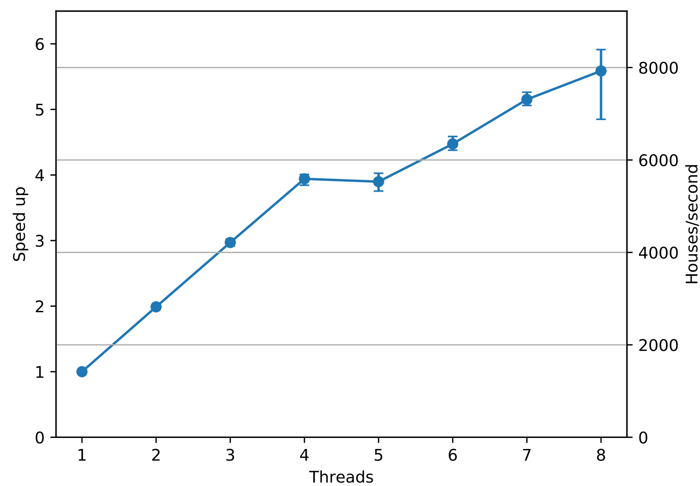


Figure 4.3: Total speed up is shown with the increase of thread count to a maximum of 8 threads.

The figure 4.3 shows the total speed up due to increasing core count. The histogram generation is dominated (see figure 4.21) by the extraction times as the histogram generation is not computationally very heavy.

4.4 GPU benchmarks

There are multiple ways of doing a GPU implementation for Histogram. The algorithm changes depending on the number of bins and the number of physical threads. In this implementation, 1 thread of a GPU is used to calculate the histogram of a single house.

The extraction algorithm is however different from the main algorithm. The extraction used 1 thread per data point.

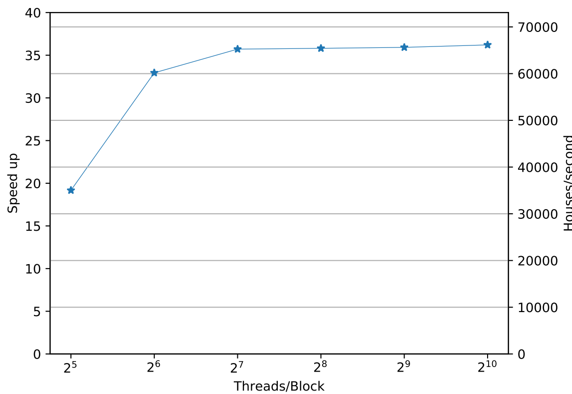


Figure 4.4: Speed up of GPU execution of extraction compared to single threaded CPU implementation.

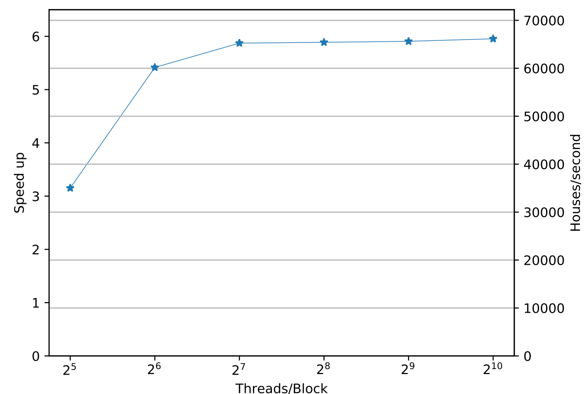


Figure 4.5: Speed up of GPU execution of extraction compared to octa threaded CPU implementation.

Figure 4.4 shows how increasing the thread count per block improves the performance of the extraction and we have 1024 threads per block giving us the maximum performance.

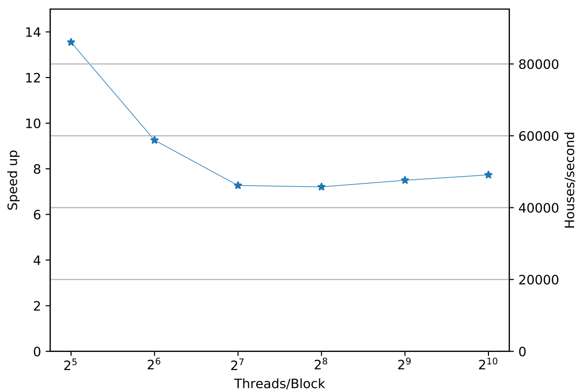


Figure 4.6: Speed up of GPU execution of histogram compared to single threaded CPU implementation.

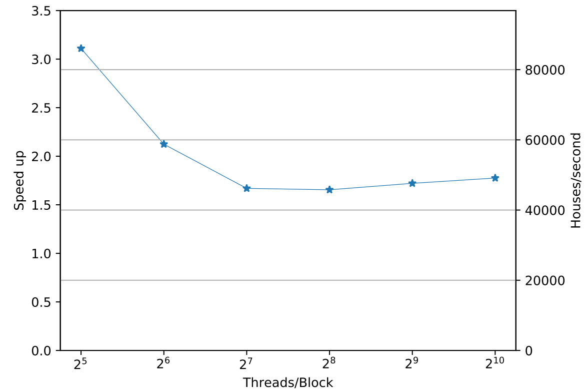


Figure 4.7: Speed up of GPU execution of histogram compared to octa threaded CPU implementation.

The histogram execution is being discussed here. Figure 4.6 shows that 32 threads per block provides best performance. The benchmark is memory bound and decreasing thread count for larger number of houses reduces cache misses. This improves the performance considerably. The benchmark shows memory divergence as each warp of 32 threads accesses memory regions which are far away from each other.

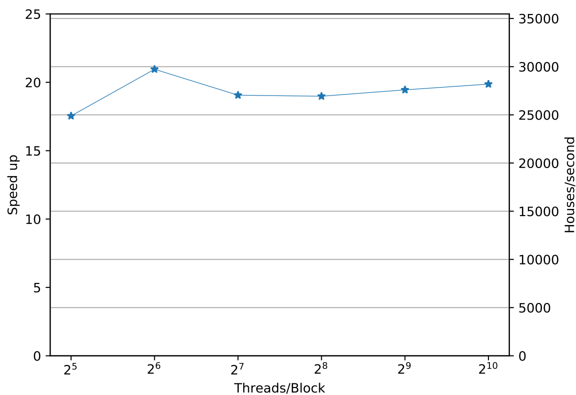


Figure 4.8: Speed up of GPU execution of computational time compared to single threaded CPU implementation.

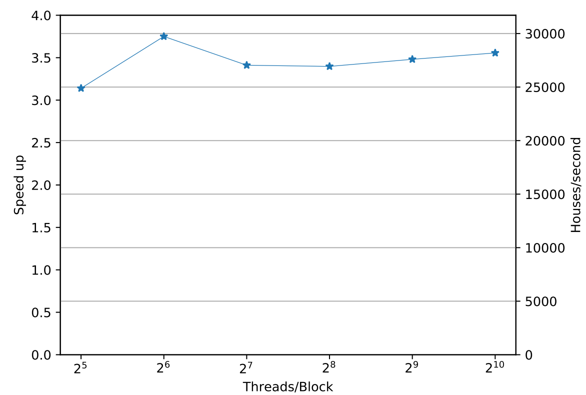


Figure 4.9: Speed up of GPU execution of computational time compared to octa threaded CPU implementation.

Figure 4.8 shows that the extraction time dominates the computational work of the

histogram generation (see figure 4.21). The plot shows that having 64 threads / block will provide the maximum speed up for 28165 houses. However that would not be necessary as one can have different number of threads per block for the different computations (like 1024 threads/block for extraction and 32 threads/block for histogram generation).

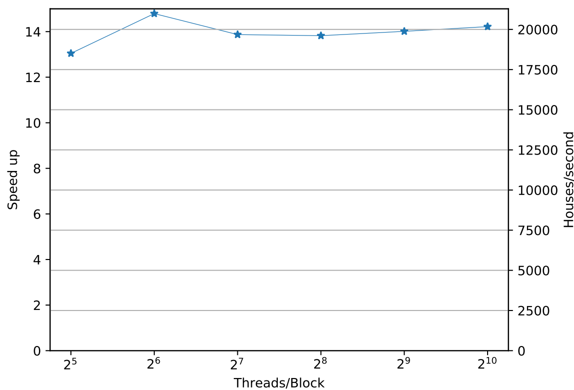


Figure 4.10: Speed up of GPU execution of total time compared to single threaded CPU implementation.

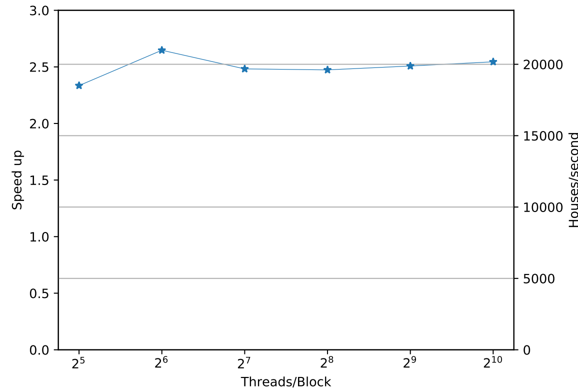


Figure 4.11: Speed up of GPU execution of total time compared to octa threaded CPU implementation.

Figure 4.10 shows the total speed up for performing the operation of the GPU. It is interesting as neither 1024 threads/block nor 32 threads/block shows the maximum speed up. The plot shows that 64 threads per block will provide the best performance.

4.5 GPU benchmark block version

This is another implementation of Histogram but uses the whole block of threads in generating the histogram. Each thread works in its own space of memory and generates a private histogram. The private histograms are summed over at the end to get the final histogram. Shared memory is used to store the private histograms as they need to be accessed in the later stages. Due to increased parallelism, this version of histogram should be faster than the 1 thread/house version.

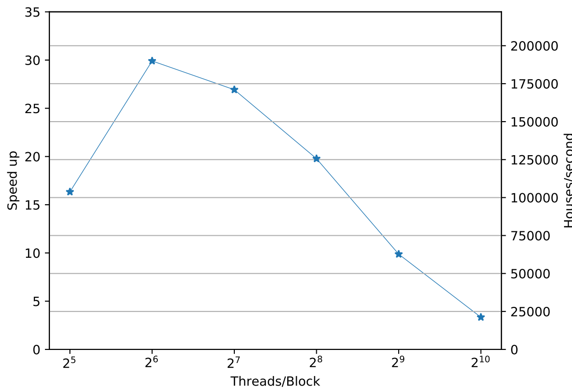


Figure 4.12: Speed up of GPU execution of histogram (block) compared to single threaded CPU implementation.

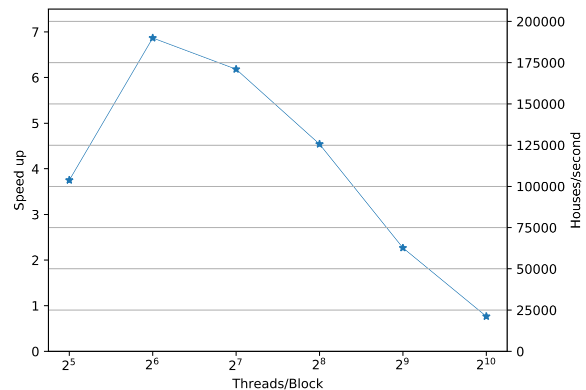


Figure 4.13: Speed up of GPU execution of histogram (block) compared to octa threaded CPU implementation.

Figure 4.12 show that one gets good speed ups at 64 threads per block. Unlike the previous situation, where the 3× speed up was the norm, here one gets nearly 7× speed up with the use of shared memory and increased parallelism. Yet this benchmark is still memory bound and the ratio of the performance of the CPU and the GPU roughly mirrors the ratio of their memory bandwidths.

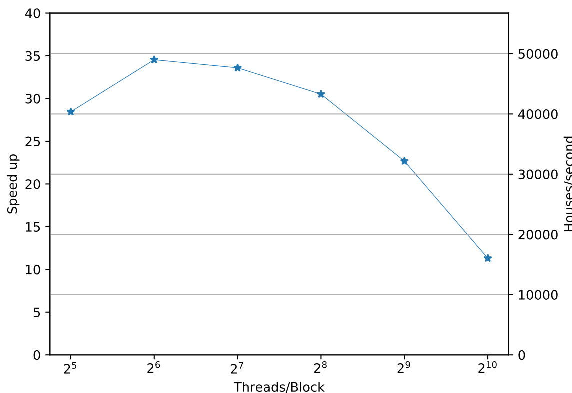


Figure 4.14: Speed up of GPU execution of compute (block) compared to single threaded CPU implementation.

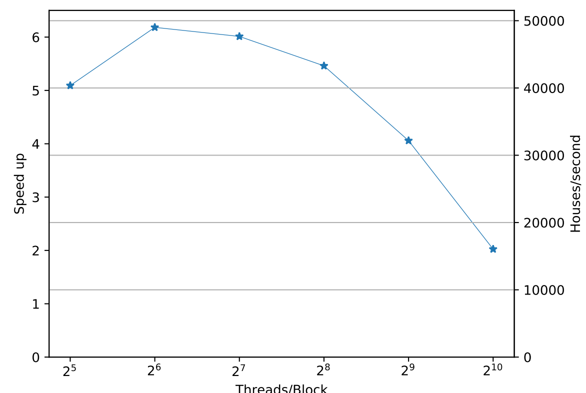


Figure 4.15: Speed up of GPU execution of compute (block) compared to octa threaded CPU implementation.

The computational time as shown in Figure 4.14 has decreased a bit and one get

improved speed for moving the operation of histogram generation on to the shared memory.

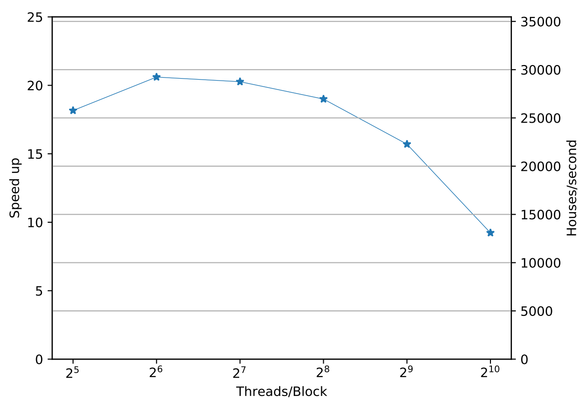


Figure 4.16: Speed up of GPU execution of total (block) compared to single threaded CPU implementation.

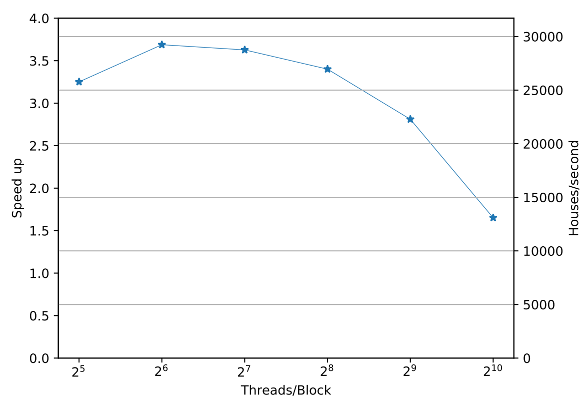


Figure 4.17: Speed up of GPU execution of total (block) compared to octa threaded CPU implementation.

Figure 4.16 shows that the increased usage of the shared memory works and instead of 2.5× speed up , one gets 3.5× speed up . The extraction times still dominate (see Figure 4.21) this benchmark. Making the histogram run faster will not help.

4.6 Power consumption

The power consumption on executing the benchmarks are shown in the following plots.

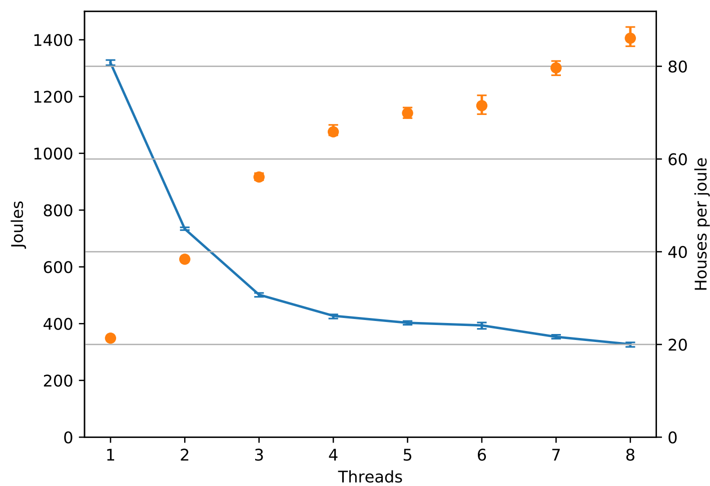


Figure 4.18: Power consumption of the CPU implementation of histogram benchmark.

The energy used in the CPU implementation of the benchmark as shown in Figure 4.18 decreases when increasing the thread count. A nearly 3× improvement in performance is seen.

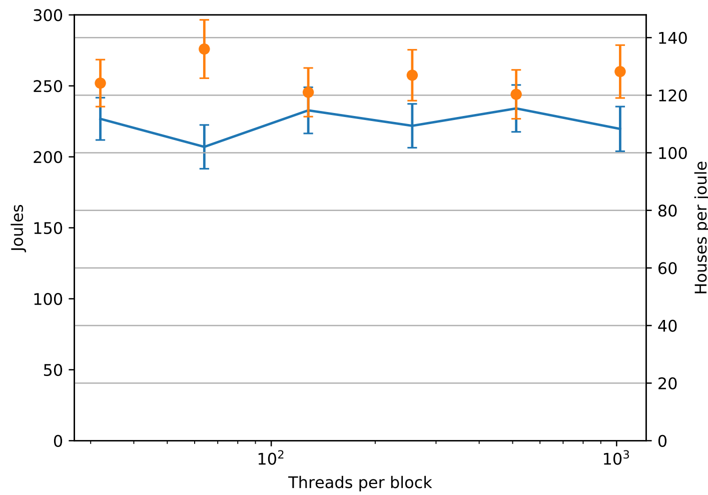


Figure 4.19: Power consumption of the GPU implementation (1 thread/house) of histogram benchmark.

The energy used up in 1 thread/house implementation of histogram is shown in figure 4.19. It shows improvement over the CPU implementation. The fastest implementation (64 threads/block) is also the most power efficient.

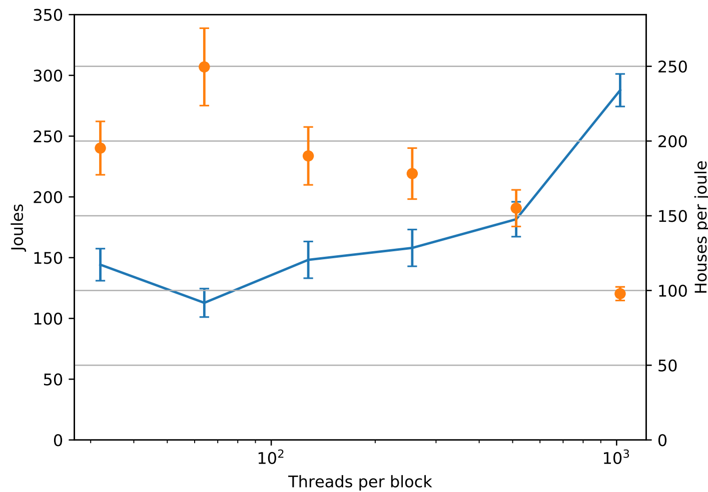


Figure 4.20: Power consumption of the GPU implementation (1 block/house) of histogram benchmark.

The energy used up in 1 block/house version of Histogram is shown in figure 4.20. It decreases by a large amount at 64 threads/block. This is also the fastest version of this benchmark. It does about 250 houses/Joule compared to 20 houses/Joule of the single threaded CPU implementation.

4.7 Summary

Figure 4.21 shows all the different parts of the histogram benchmark.

				CPU				
Threads	Transfer to GPU		Extraction time		Histogram		Transfer to CPU	
1			15.42	76%	4.93	24%		
8			2.50	73%	0.92	27%		
Threads / block				GPU (1 thread/house)				
64	0.39	29%	0.47	35%	0.48	36%	0.00	0%
Threads / block				GPU (1 block/house)				
64	0.39	40%	0.43	44%	0.15	15%	0.00	0%

Figure 4.21: Breakdown of histogram execution times of 28165 houses. Time in seconds.

The histogram being computationally simple caused the execution times to be dominated by other parts (like extraction) of the execution. Extraction of data dominate the execution times on both the CPU and GPU block versions. 32 threads per block shows the maximum speed up in the thread/house version but if the same block size is used for extraction and generating histogram, then 64 threads per block is the sweet spot for fastest results. The block version of the benchmark at 64 threads/block is the fastest implementation. It is 22 times as fast and 12.5 times as energy-efficient compared to the single threaded CPU implementation. However the actual benchmark takes up only 15% of the execution time. It is also the most power efficient implementation of the algorithm. Since the extraction times and data transfer to the GPU are dominating this benchmark, it is better to perform asynchronous transfers so that the memory transfer and the execution of the algorithm are overlapped.

Chapter 5

Similarity Search Benchmark

In this benchmark, one looks for similarities of the time series data of two different customers. The ten most similar customers are then picked out. The following steps are performed for similarity search.

1. As with every benchmark, the power consumption values are extracted from the packed format.

2. Then a dot product is taken with the time series power consumption data of one customer with that of another customer. Then the results are normalized and stored in an array. In this case if two customers have data of different lengths, the smaller length is chosen.

3. The closest 10 matches for each house are then selected out from the calculated cosine similarity.

The similarity benchmark is different from all other benchmarks as its execution times have a quadratic relationship with the number of houses. Assuming N houses, in order to calculate similarity, each house has to find out similarity with (N-1) other houses. That is also true for similarity search as each house has to look through the all the (N-1) other houses for calculation of similarity.

5.1 Implementation details

In this section four different arrays will be considered:

- *offsets*: Stores the offsets of the power consumption data of each of the houses
- *powerconsum*: Stores the power consumption data of each data
- *simval* : Stores the cosine similarity values. This array can be compared to a matrix with each row and each column storing representing a house. The cells of the matrix contains the cosine similarity values for a pair of matrices. Though a matrix is a 2D array, this implementation uses it in a flattened out fashion, a 1D array. This is a temporary array.
- *tenclosest* : This is an array of structs with each struct containing an array capable of storing ten elements.

The core algorithm for calculating cosine similarity for the CPU's version and GPU's 1 thread/(house pair) is exactly same. The core algorithm is a function which takes two house indexes *hid1* and *hid2*. Since the cosine similarity for a pair of houses is symmetric, the cosine calculation proceeds only when $hid1 < hid2$ to avoid redundant work. As explained in Chapter 3, the offset array is used to find the location of the starting point and the ending point of the power consumption data of the pair of houses. As explained before, if the lengths of the power consumption data of the two houses are different, then the smaller value is chosen. Using a loop the dot product and the squares of the norms of the two vectors are simultaneously calculated. In the end the cosine similarity of the pair of houses is calculated and stored in the array *simval*. The main difference in the implementation is that in the CPU version a single thread operates on multiple house pairs but in the GPU a single thread operate on a pair of houses.

The core algorithm of the block version in the GPU implementation is however different. Here one block of threads operates on a pair of houses. In this implementation each thread of the block does a partial dot product and partial squares of the norms of the two vectors. These partial values are stored in the shared memory and then a parallel reduction, as shown in Appendix A, is done to get the dot product and the squares of the vector norms. Finally the calculated cosine similarity is calculated and stored in *simval*.

The core algorithm for calculating the closest 10 matches has been left unchanged in all the implementations. The actual function operates on a per *hid* basis. The algorithm loops through all the results stored in *simval* corresponding to a given *hid* and stores the result in *tenclosest*.

5.2 Validation

This benchmark finds the top ten matches and the order of the matches is not significant. However the algorithms used in both the Matlab implementation and in this benchmark will cause the houses to be put in a sorted fashion. Samples of the benchmark implementation results were tested against the Matlab implementation. Since the floating point operations provide slightly different results, the order of the sorted values tended to swap at times. This happened whenever the two calculated similarity values are very close to each other.

5.3 Similarity CPU Benchmarks

Since the similarity has a quadratic relation to the number of houses the secondary y-axis has Houses^2/second instead of the usual Houses/second. The following CPU benchmarks also show some noticeable dips when the thread count is increased after four threads. The threads are pinned to the cores but the performance regressed when thread count has been above four. This phenomena does not happen with 1 thread or 8 threads (all logical cores are used up). The number of houses used is 880.

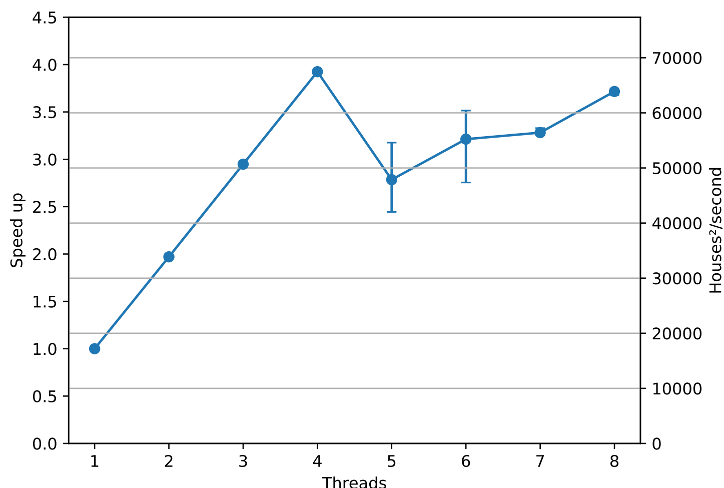


Figure 5.1: Similarity calculation speed up on a CPU with increasing thread count.

The similarity calculation proved (figure 5.1) to be computationally intensive and four threads are enough to saturate the CPU. Increasing thread count does not help and leads

to regression in performance.

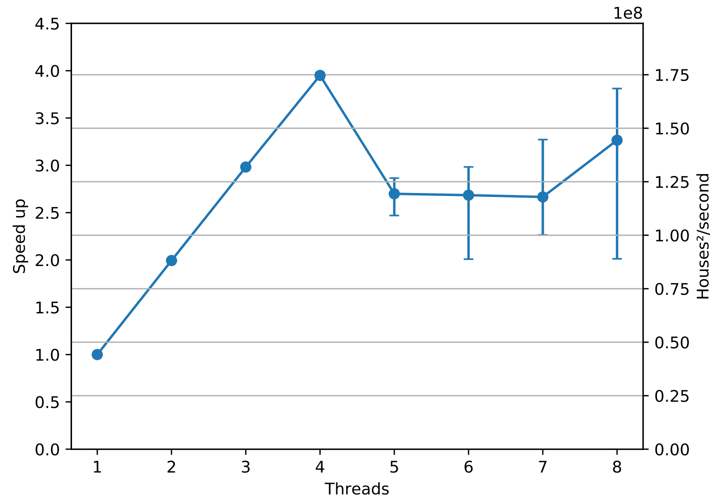


Figure 5.2: Similarity matching speed up on the CPU with increasing thread count.

The similarity matching (Figure 5.2) proved to be rather unstable as it runs really fast (see Figure 5.21) and has very short running times. Thus the overhead of thread scheduling and thread creation dominate the benchmark. The instability does not affect the total execution times as it barely makes a dent in running times.

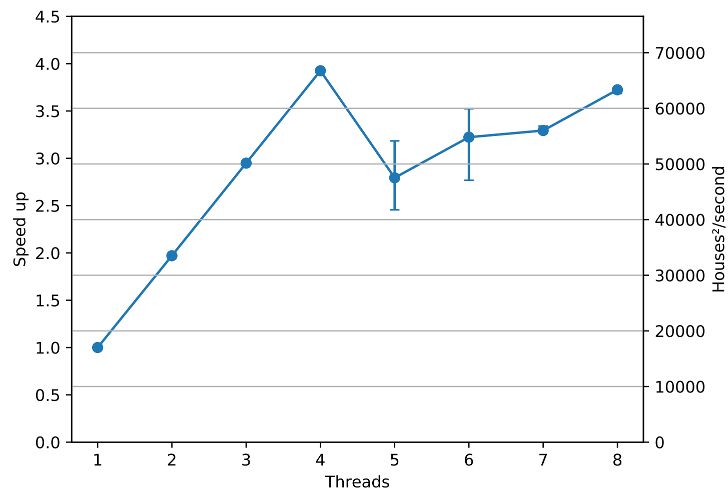


Figure 5.3: Total matching speed up with CPU with increasing thread count.

Figure 5.3 shows the the total speed up. The similarity times as seen in figure 5.21 dominate the execution times. Thus the plots looks similar. This benchmark prefers the four threads pinned to the cores for maximum performance.

5.4 Similarity GPU Benchmarks

In this implementation of the benchmark each GPU thread operates on a pair of houses.

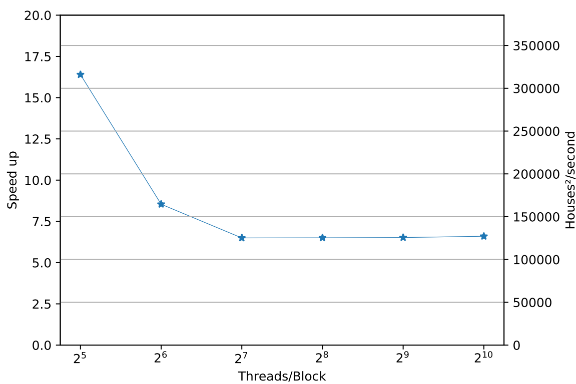


Figure 5.4: Speed up of GPU execution of cosine similarity calculation compared to single threaded CPU implementation.

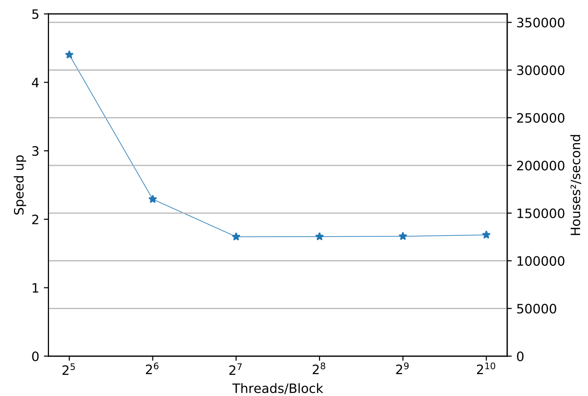


Figure 5.5: Speed up of GPU execution of cosine similarity calculation compared to octa threaded CPU implementation.

The figure 5.5 shows speed up of about 4.5× compared to the CPU version when using 32 threads per block. This is a memory bound benchmark and shows signs of memory divergence as each warp of threads will generate 64 different address locations and they are far away.

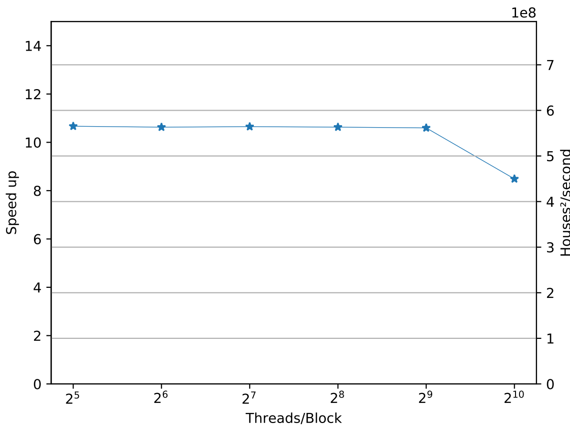


Figure 5.6: Speed up of GPU execution of similarity search compared to single threaded CPU implementation.

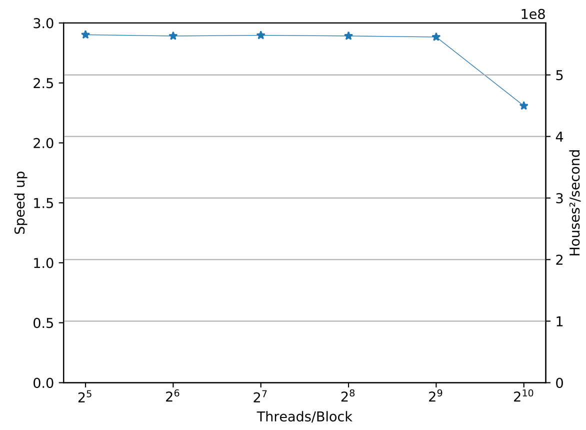


Figure 5.7: Speed up of GPU execution of similarity search compared to octa threaded CPU implementation.

The similarity matching workload as seen in Figure 5.6 is not computationally heavy. Neither is it memory bound. The performance is nearly independent of the number of threads launched.

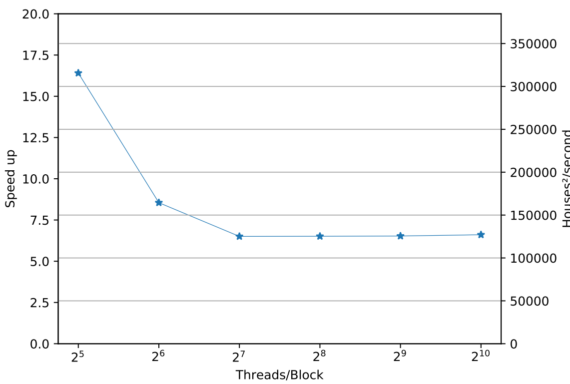


Figure 5.8: Speed up of GPU execution of the computational workload compared to single threaded CPU implementation.

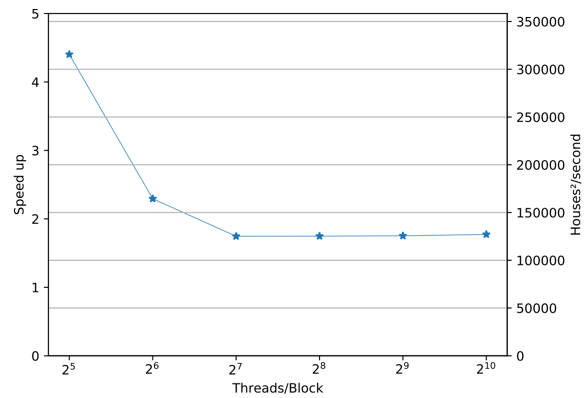


Figure 5.9: Speed up of GPU execution of the computational workload compared to octa threaded CPU implementation.

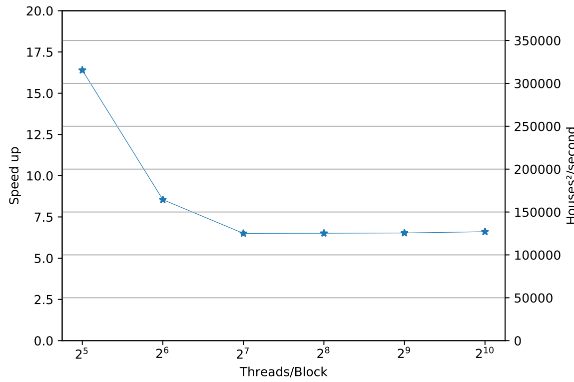


Figure 5.10: Speed up of GPU execution of total workload compared to single threaded CPU implementation.

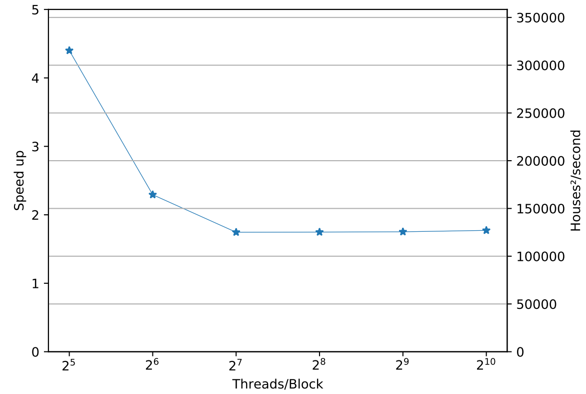


Figure 5.11: Speed up of GPU execution of total workload compared to octa threaded CPU implementation.

Both the computational times (Figure 5.8) and the total times (Figure 5.10) looks the same as the benchmark is dominated by calculating the cosine similarity.

5.5 Similarity Block GPU Benchmarks

Here a whole block is used to compute the cosine similarity of the customers as it is the part with the maximum execution time. In this case parallel reduction is used to speed up the dot product operation. All other parts of the benchmark are left untouched.

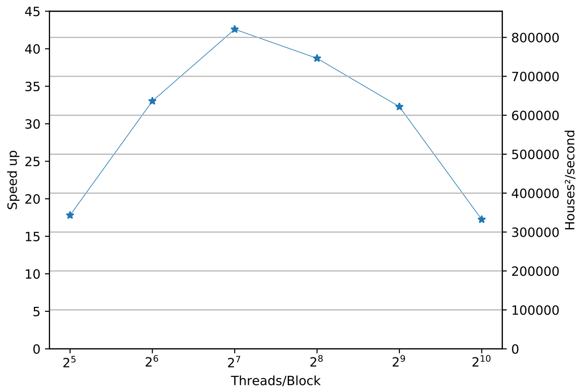


Figure 5.12: Speed up of GPU execution of block version of the similarity compared to single threaded CPU implementation.

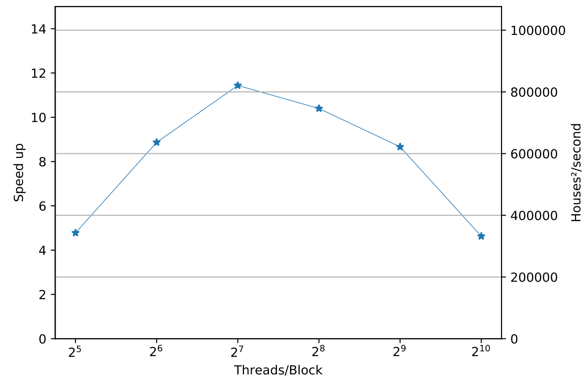


Figure 5.13: Speed up of GPU execution of block version of the similarity compared to octa threaded CPU implementation.

Maximum speed up (figure 5.12) can be observed at 128 threads per block. This is even more than of the previous implementation. The increased parallelism is due to larger thread count and lower memory divergence in this version. All the threads in a warp are working on two houses.

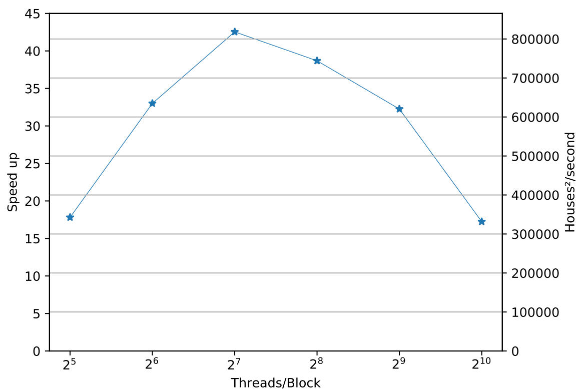


Figure 5.14: Speed up of GPU execution of block version of the computational workload compared to single threaded CPU implementation.

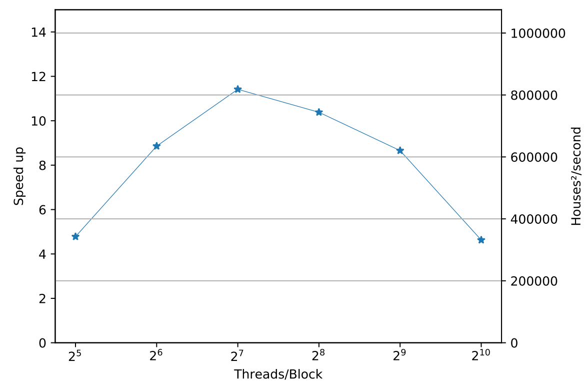


Figure 5.15: Speed up of GPU execution of block version of the computational workload compared to octa threaded CPU implementation.

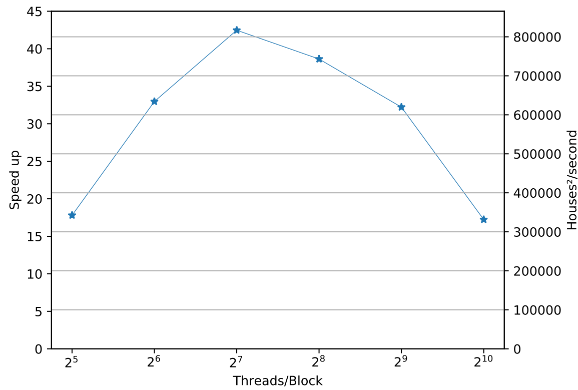


Figure 5.16: Speed up of GPU execution of block version of the total compared to single threaded CPU implementation.

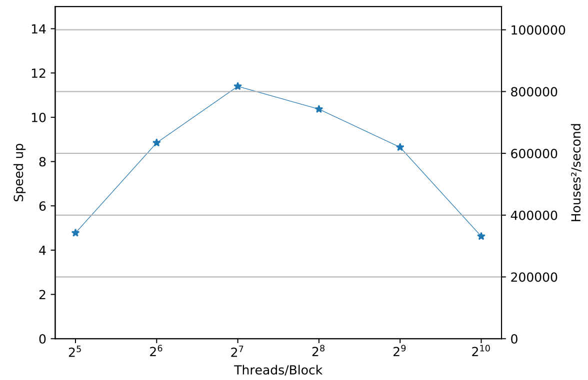


Figure 5.17: Speed up of GPU execution of block version of the total compared to octa threaded CPU implementation.

It can be immediately seen that the speed up obtained in this program is higher, as it is possible to have better utilization of the memory bandwidth. Reality is that this program is memory bound and the higher VRAM bandwidth and its proper utilization is the reason for the speed ups. The similarity calculation is $O(N^2)$ benchmark so the running times for large number of houses will eclipse the running times of all other parts of the computation be it extraction, matching or the time it takes to move data in and out of GPU VRAM.

5.6 Power consumption

The power consumption of the similarity algorithm are provided below. 880 houses are used to measure the power consumption values.

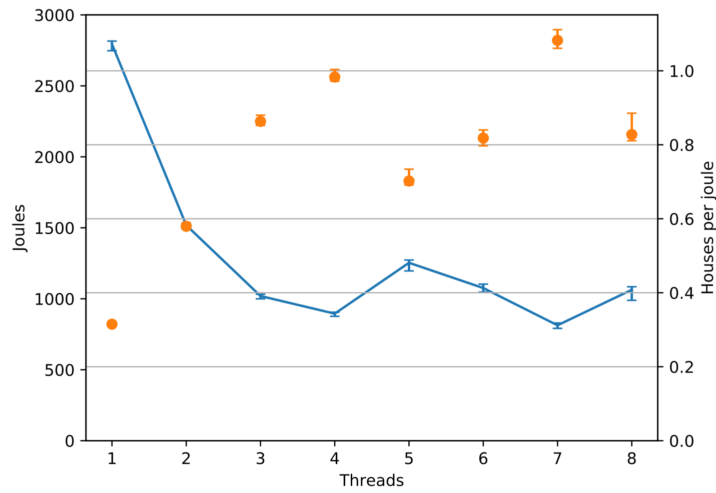


Figure 5.18: Power consumption when executing CPU implementation of similarity algorithm.

The total power consumption of the CPU implementation as seen in Figure 5.18 does not decrease significantly with increased thread count.

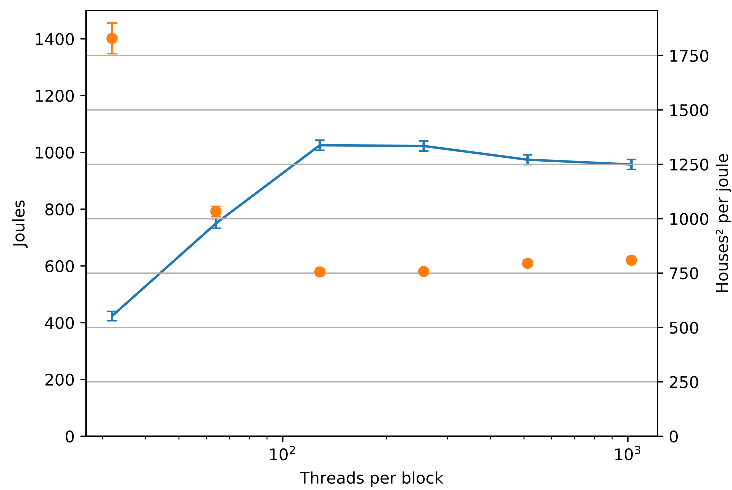


Figure 5.19: Power consumption when executing GPU implementation (1 thread per pair of houses) of similarity algorithm.

The case of 32 threads/block as seen in Figure 5.19 is the most energy efficient and also

the fastest in the implementation with 1 thread/pair of houses.

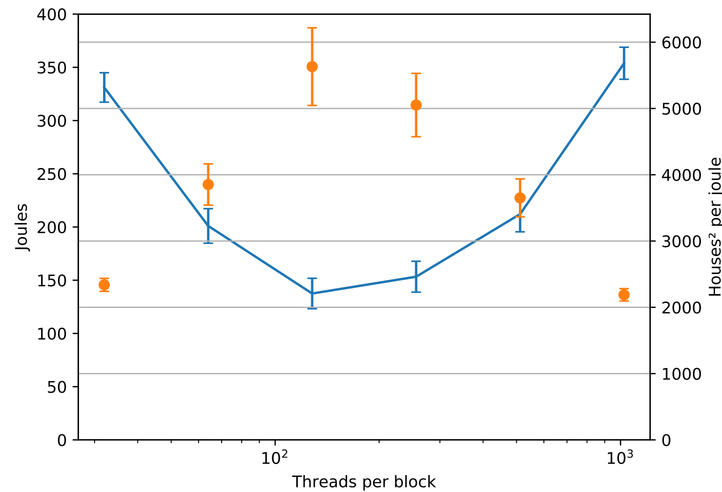


Figure 5.20: Power consumption when executing GPU implementation (1 block per pair of houses) of similarity algorithm.

The case of 128 threads/block as seen in Figure 5.20 is the most energy efficient and also fastest in the implementation of 1 block per pair of houses.

5.7 Summary

Threads	Transfer to GPU		Extraction		Similarity		Similarity match		Transfer to CPU	
					CPU					
1			0.47	1%	45.05	99%	0.02	0%		
8			0.10	1%	12.12	99%	0.01	0%		
Threads / block					GPU (1 thread/house)					
64	0.10	0%	0.11	0%	156.89	100%	0.09	0%	0.00	0%
Threads / block					GPU (1 block/house)					
64	0.10	0%	0.11	0%	144.50	100%	0.08	0%	0.00	0%

Figure 5.21: Breakdown of similarity execution times of 880 houses. Time in seconds.

Figure 5.21 shows the various parts of the benchmark. The cosine similarity calculation takes the maximum amount of time irrespective of the type of hardware or implementation. The case of the similarity calculation which has 128 threads/block is the fastest ($11\times$) and the most power efficient ($8.2\times$). The similarity benchmark are memory bound and thus the speed ups shown against the 8 threaded CPU implementation is roughly a ratio of VRAM to RAM bandwidth. Further improvements to this benchmarks do not seem to be possible due to the VRAM bottleneck.

Chapter 6

ParX Benchmark

The Parx model stands for periodic auto regression with exogenous variables. In the benchmark however only the energy consumption has been used for regression. This model has 24 'seasons' corresponding to 24 hours in a day. All the benchmarks below have the order P=3 for the auto-regressive model. This means that the value of the current hour is determined based on the last 3 hours.

Both the CPU and GPU reuses the same code but the granularity of parallelization is different. The core algorithm is implemented as follows. For example let us asssume that there are 12000 data points for a given house. Now there are 24 hours and hence for each hour there will be 500 data points. Since the current value of the power data is dependent on last three hours we end up with a matrix equation of the form $A_{500\times3}x_{3\times1} = b_{500\times1}$. Multiplying both sides with A^T gives us the following matrix equation $(A^TA)_{3\times3}x_{3\times1} = A^Tb_{3\times1}$. Now the value of x is determined using the conjugate gradient method. For each house one then gets 72 coefficients.

6.1 Implementation details

The CPU and the GPU implementations uses the same algorithm. The core function which executes the algorithm works tat the granularity of a single hour of each house. The only difference is that a single CPU thread loops through multiple houses but a GPU thread works on a single house. As with other algorithms, *offsets* is used to locate the power consumption data from the array *powerconsump* for a given house. The matrix (A^TA) and vector A^Tb are calculated directly with a loop going over the the

power consumption data with a stride length of 24. Let us assume it is calculating for a given hour h and the *offset* is of. For the A matrix the loop tries to access $powerconsump[of + h - 1], powerconsump[of + h - 2], powerconsump[of + h - 3]$ in one cycle and in the next cycle accesses $powerconsump[of + h - 1 + 24], powerconsump[of + h - 2 + 24], powerconsump[of + h - 3 + 24]$ etc. For the b vector the values accessed are $powerconsump[of + h], powerconsump[of + h + 24]$ and so on. Once both the matrix and the vector have been calculated then the regression is performed with the help of the conjugate gradient. Since this is only a 3×3 matrix, a very simple implementation of conjugate gradient is used to get the regression values.

The GPU version of the benchmark has a parameter which controls the looping strategy for each house. This allows the GPU version to vary the number of threads that can operate on a given house. Thus, one thread can operate on all the 24 hours, only one hour for a given house, or anywhere in between.

6.2 Validation

For testing the correctness of the implementations, the results of the workload is compared against the original Matlab implementations. Since there are differences in the floating point results as explained in Chapter 3, the results obtained are slightly different. To determine the magnitude, the coefficients generated by the regression for the sampled house are flattened out as a vector. Then the L_∞ norm of the relative differences of the results generated of the two implementations is looked into. The results differ less than 0.01% for the sampled houses. Sometimes the results difference is large at about 4%. This is caused by the fact that for some houses the matrix $(A^T A)$ is singular. Thus finding the solution to the given matrix is numerically unstable operation. These are not show stopper problems as the power consumption values themselves are very coarse grained (small values with precision up to a single decimal digit) and so any prediction is not going to differ significantly from the Matlab implementation.

6.3 CPU Performance

The following charts will show the speed up one gets when the thread count is increased for the ParX algorithm running on the CPUs. The number of houses used is 28165. The threads are pinned to the cores. The next two charts will show the speed up only when using ParX and also the total (ParX+extraction) speed up.

In the CPU implementation if there are say N houses and T threads then each thread performs the execution on K = ceiling(N/T) houses. Each thread has a loops over K houses and an inner loop iterates over the 24 seasons for each house.

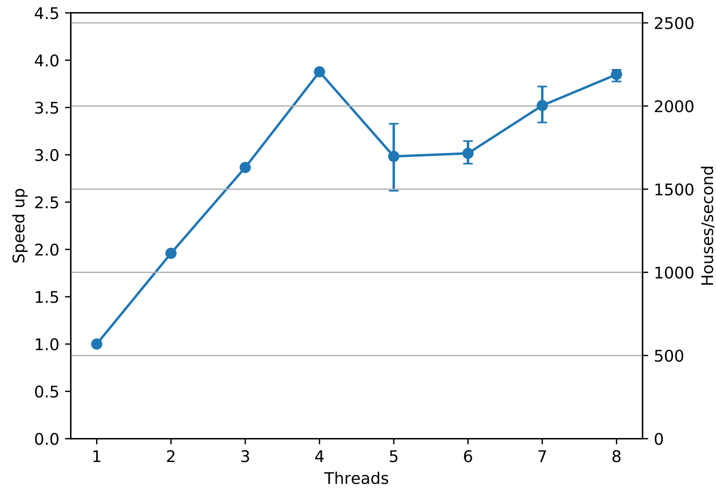


Figure 6.1: ParX speed up with increasing thread count.

The ParX plot in Figure 6.1 shows that the speed up is not consistent. The threads are pinned to the cores but as the thread count is increased over four, small irregularities seem to pop up.

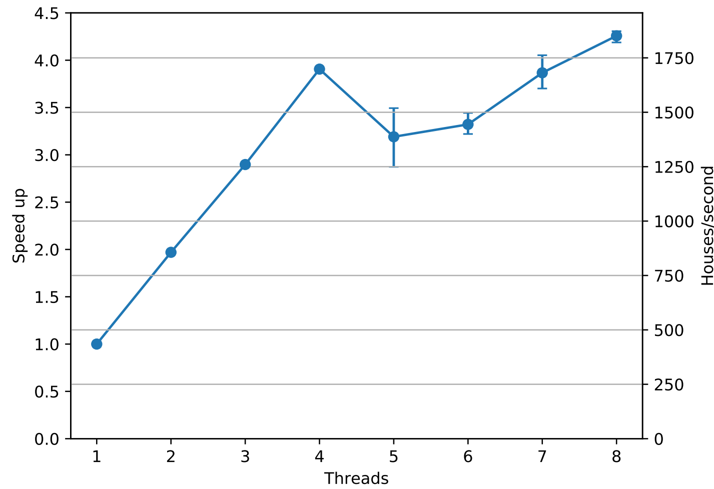


Figure 6.2: Total speed up with increasing thread count.

The total speed up plot in Figure 6.2 shows a higher speed up than just the ParX speed up when core count is increased. This is due to the fact that extraction of power data exhibits a greater speed up and benifits from more effective parallelization than the ParX algorithm. The table shows the breakdown of the various components of the execution. The fact that the extraction times go from 24% to 15% as seen in Figure 6.30 when thread count is increased is why the speed up is higher.

6.4 GPU Performance

The GPU performance relative to the CPU is analyzed in six things

1. ParX algorithm speed up compared to 1 CPU thread doing only ParX.
2. ParX algorithm speed up compared to 8 CPU threads doing only ParX.
3. Total compute (ParX+extraction) speed up compared to 1 CPU thread.
4. Total compute (ParX+extraction) speed up compared to 8 CPU threads.
5. Total (ParX+extraction+memory transfer) speed up compared to 1 CPU thread.
6. Total (ParX+extraction+memory transfer) speed up compared to 8 CPU threads.

6.4.1 1 thread per house

Here only 1 GPU thread per house will be used to compute the regression coefficients of that house. Thus the number of threads equal to the number of houses.

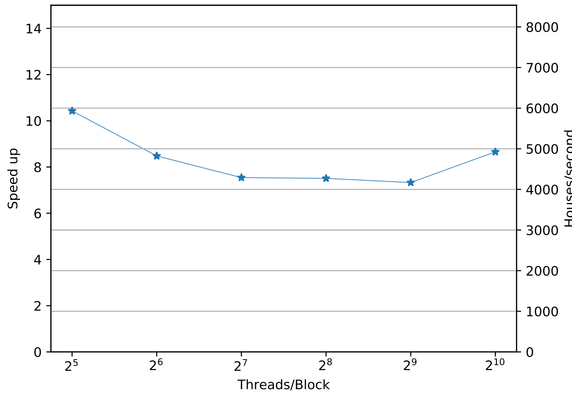


Figure 6.3: Speed up of ParX (1 thread per house) compared to the single threaded CPU implementation.

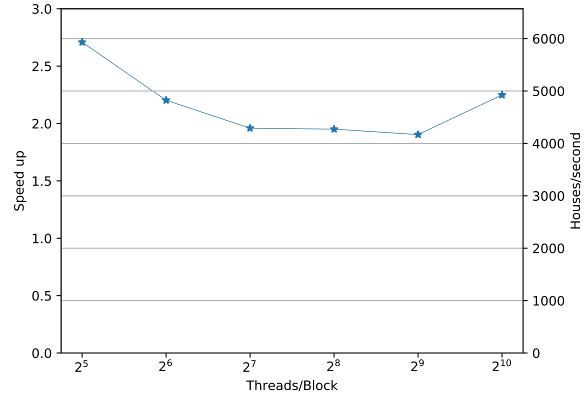


Figure 6.4: Speed up of ParX (1 thread per house) compared to the octa threaded CPU implementation.

The ParX plot as seen in Figure 6.3 shows that the maximum speed up is obtained by using 32 threads per block. Whenever smaller number of threads per block shows higher speed up on a GPU it means that the algorithm is memory bound. This algorithm also shows the effects of memory divergence as each thread will access different parts of the memory for each house.

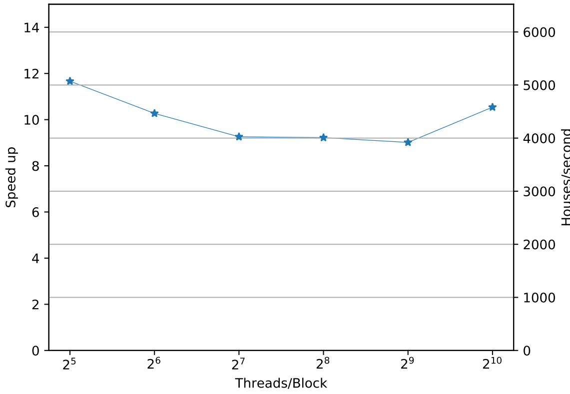


Figure 6.5: Speed up of compute (1 thread per house) compared to the single threaded CPU implementation.

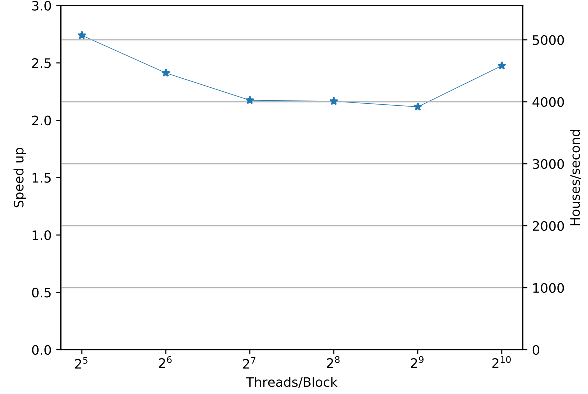


Figure 6.6: Speed up of compute (1 thread per house) compared to the octa threaded CPU implementation.

The pure computational plot as seen in Figure 6.5 shows higher speed up at about 10x. This is caused by the fact that extraction of the data undergoes a significant speed up when executed on the GPU.

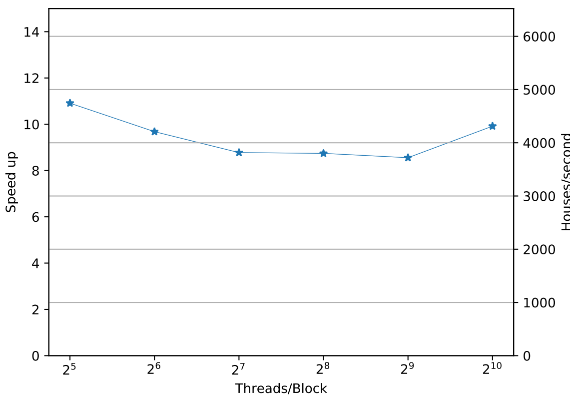


Figure 6.7: Speed up of total execution time (1 thread per house) compared to the single threaded CPU implementation.

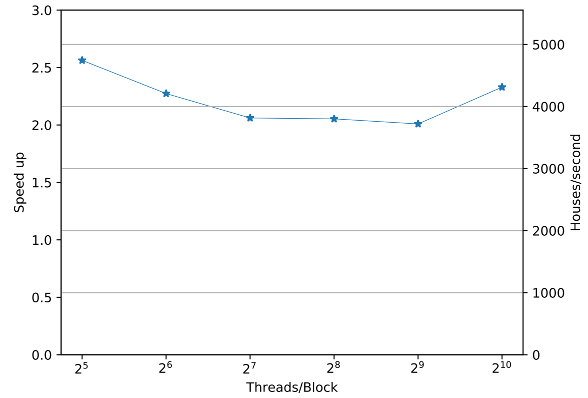


Figure 6.8: Speed up of total execution time (1 thread per house) compared to the octa threaded CPU implementation.

The total execution time Figure 6.7 follows similar trend but the speed up is obviously smaller as the transfer of data to and from the GPU has been taken into account. The

Figure 6.30 shows that the 20% of the time is spend on extraction and transferring data to the GPU. However the slowest part is still the ParX algorithm.

6.4.2 24 threads per house

Unlike Histogram or Similarity, the ParX algorithm does not have an implementation that uses block per house. The ParX algorithm does not map well to such style of GPU parallelism. Here only 24 GPU threads per house will be used to compute the regression coefficients of that house. Each thread will correspond to each hour of a day for each house.

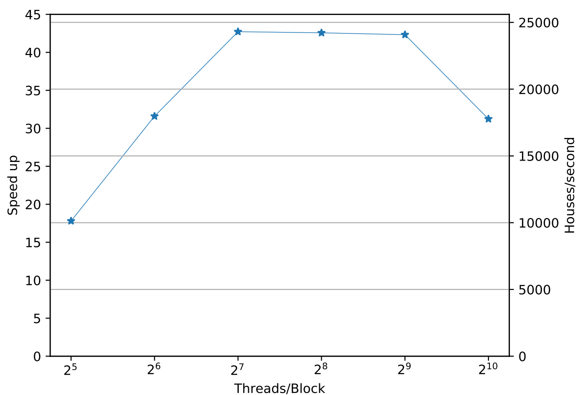


Figure 6.9: Speed up of ParX (24 threads per house) compared to the single threaded CPU implementation.

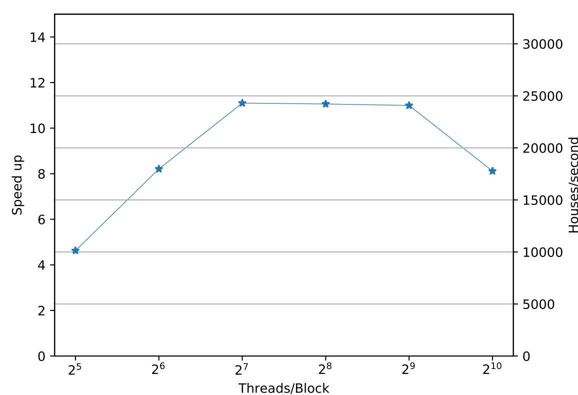


Figure 6.10: Speed up of ParX (24 threads per house) compared to the octa threaded CPU implementation.

The ParX plot in Figure 6.9 shows the improved speed up due to the 24× increase in parallelism. Also since groups of 24 threads access contiguous parts of the memory, the effects of memory divergence have decreased. The improved memory utilization meant that the new sweet spot for maximum speed up is 128 threads per block and 42× faster than the single threaded CPU implementation.

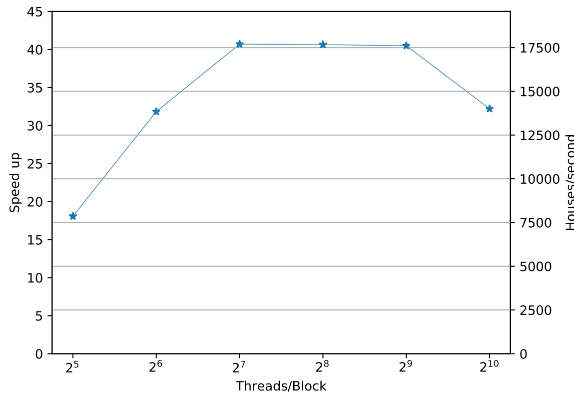


Figure 6.11: Speed up of compute (24 threads per house) compared to the single threaded CPU implementation.

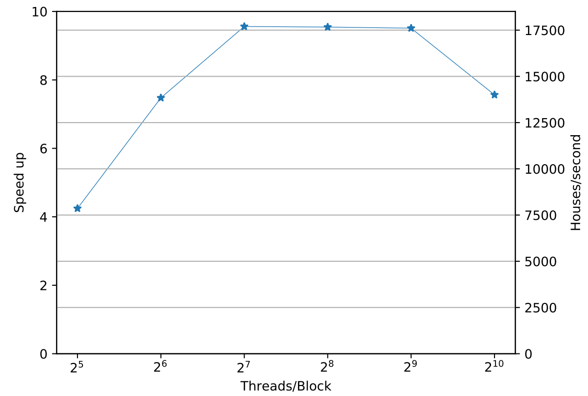


Figure 6.12: Speed up of compute (24 threads per house) compared to the octa threaded CPU implementation.

The compute plot in Figure 6.11 shows a similar speed up as the pure ParX plot. Even though the ParX computation has improved, the other parts of the execution like transfer of data to the GPU and extraction of the data taking up major amount (41%) (see Figure 6.30) of execution time.

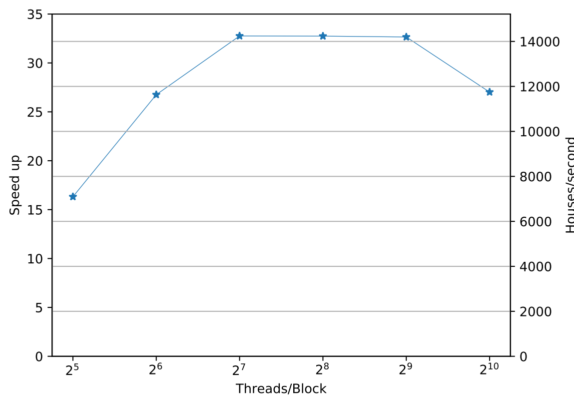


Figure 6.13: Speed up of total execution time (24 threads per house) compared to the single threaded CPU implementation.

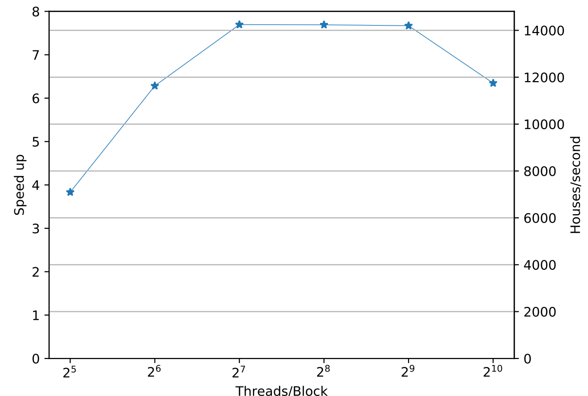


Figure 6.14: Speed up of total execution time (24 threads per house) compared to the octa threaded CPU implementation.

The total execution time plot in Figure 6.13 follows a similar trend but the speed up is obviously smaller as the transfer of data to and from the GPU has been taken into account.

The table shows that the 59% of the time is spend on ParX algorithm on the GPU. Thus the slowest part is still the ParX algorithm.

6.4.3 Different threads per house

Here only n GPU threads per house will be used to compute the regression coefficients of that house, where n is a number from 1 to 24. The threads will compute in a strided fashion. Say if n=7 then thread 0 will compute hour 0,7,21 and thread 6 will compute 6 and 13. In this section two different number of houses will tried out: 28165 houses in larger case and and 3520 houses in the smaller case. The smaller number of houses will show lower scalability as the smaller number of threads meant poor GPU utilization.

Small number of houses

The legends in the plots in this section show the number of threads per block.

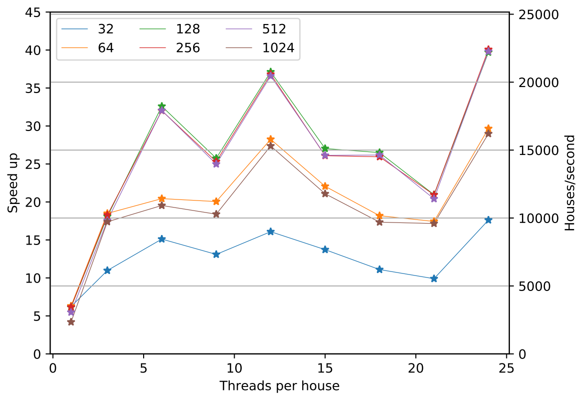


Figure 6.15: Speed up of ParX (3520 houses, multiple threads per house) compared to the single threaded CPU implementation.

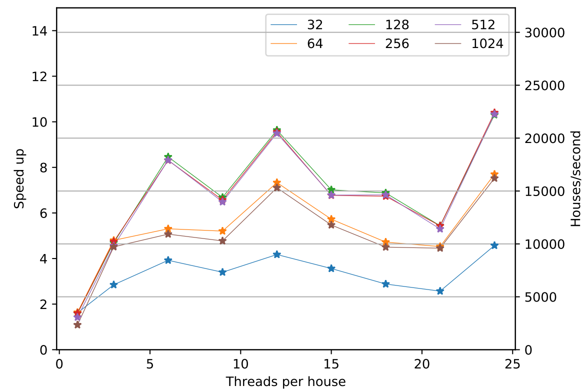


Figure 6.16: Speed up of ParX (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation.

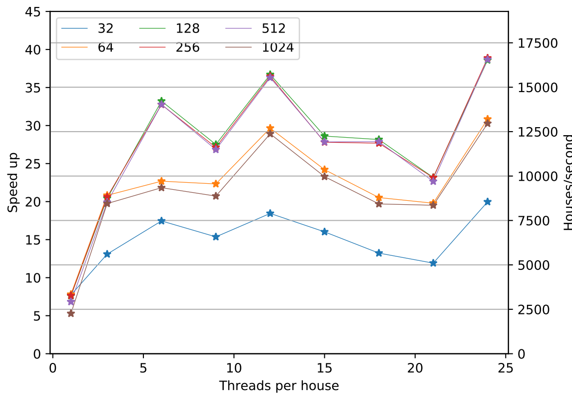


Figure 6.17: Speed up of compute (3520 houses, multiple threads per house) compared to the single threaded CPU implementation.

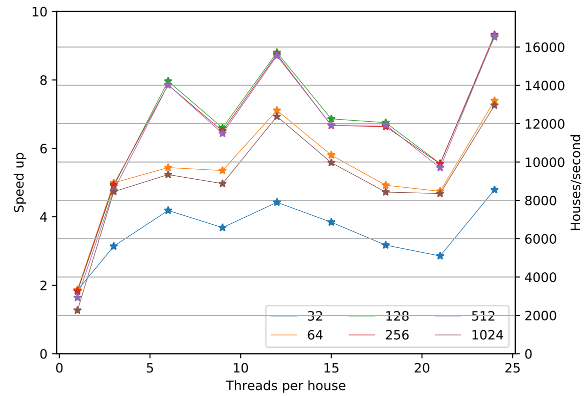


Figure 6.18: Speed up of compute (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation.

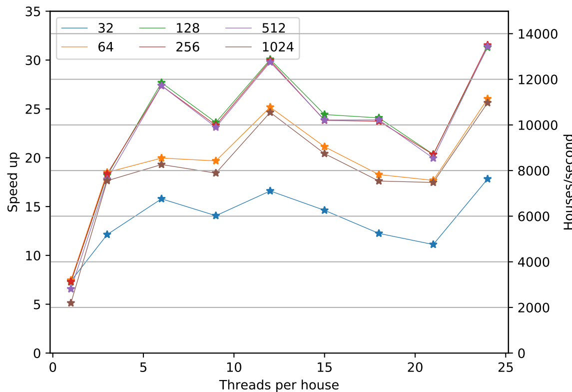


Figure 6.19: Speed up of total execution time (3520 houses, multiple threads per house) compared to the single threaded CPU implementation.

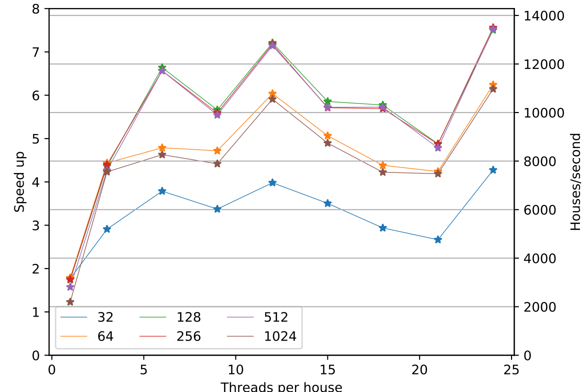


Figure 6.20: Speed up of total execution time (3520 houses, multiple threads per house) compared to the octa threaded CPU implementation.

All the plots in Figures 6.15, 6.17 and 6.19 show how increasing the thread count increases the performance but there is a large dip when using 21 threads per house. Also if 6, 12 or 24 threads per house is used, one sees the greatest speedup. Unfortunately the profiler was crashing so the exact reason for the dip is unknown.

Larger number of houses

The legends in the plots in this section show the number of threads per block.

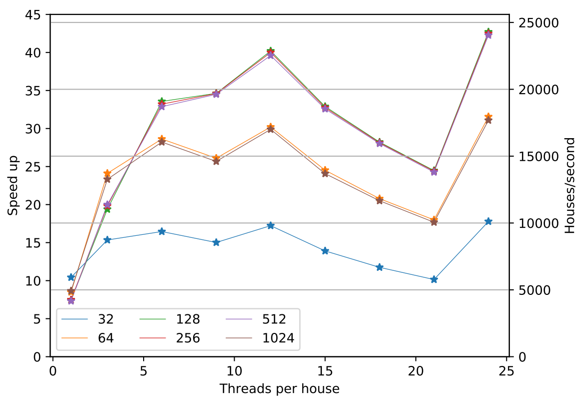


Figure 6.21: Speed up of ParX (28165 houses, multiple threads per house) compared to the single threaded CPU implementation.

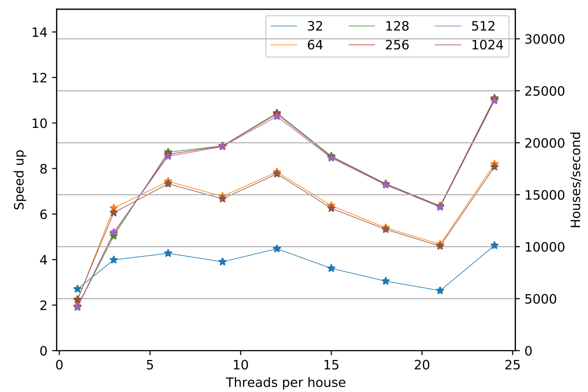


Figure 6.22: Speed up of ParX (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation.

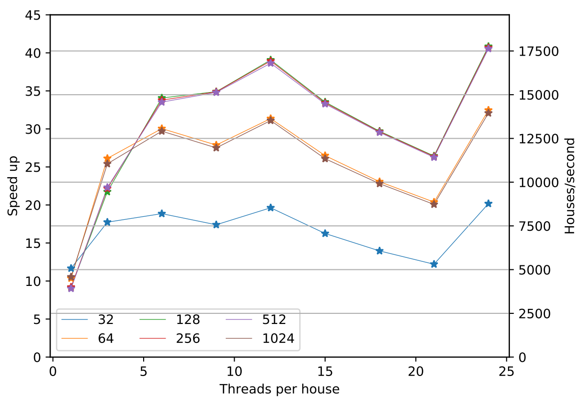


Figure 6.23: Speed up of compute (28165 houses, multiple threads per house) compared to the single threaded CPU implementation.

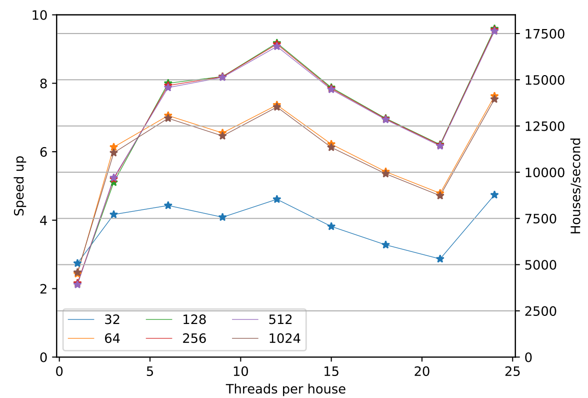


Figure 6.24: Speed up of compute (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation.

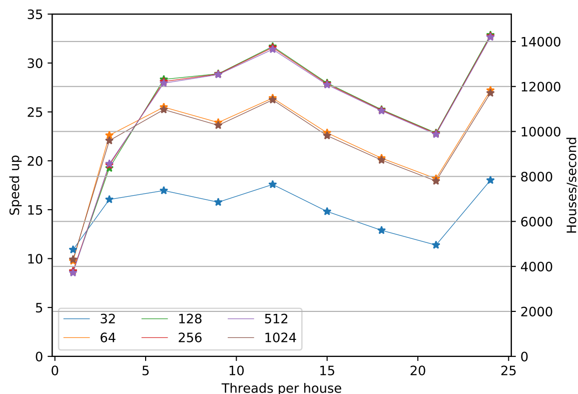


Figure 6.25: Speed up of total execution time (28165 houses, multiple threads per house) compared to the single threaded CPU implementation.

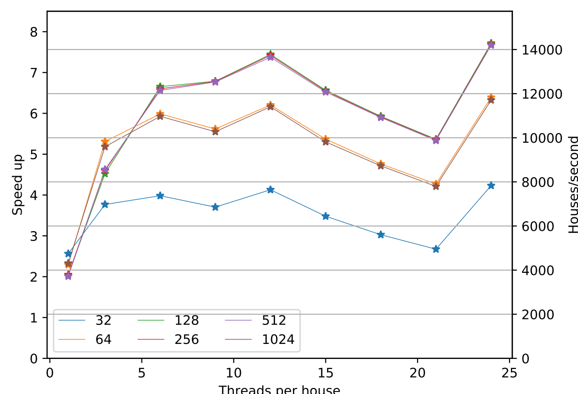


Figure 6.26: Speed up of total execution time (28165 houses, multiple threads per house) compared to the octa threaded CPU implementation.

All the plots in Figures 6.21 6.23 6.25 show how an increasing the thread count increases the performance but there is large dip when using 21 threads per house. Also if 6, 12 or 24 threads per house is used, one sees the greatest speedup. Unfortunately the profiler was crashing so the exact reason for the dip is unknown. Compared to lower number of houses however an increase in speed up as larger portion of the GPU is utilized. Compared to the smaller number of houses, the speed ups seen here have increased.

6.5 Power consumption

The power consumption in Figure 6.27 follows a predictable pattern where increasing the thread count improves the power consumption. As far as GPU is concerned both the Figures 6.28 and 6.29 the variations in the power implementation closely mirror the speed ups. The faster the operation, the more power efficient it is. This is tested for 28165 houses.

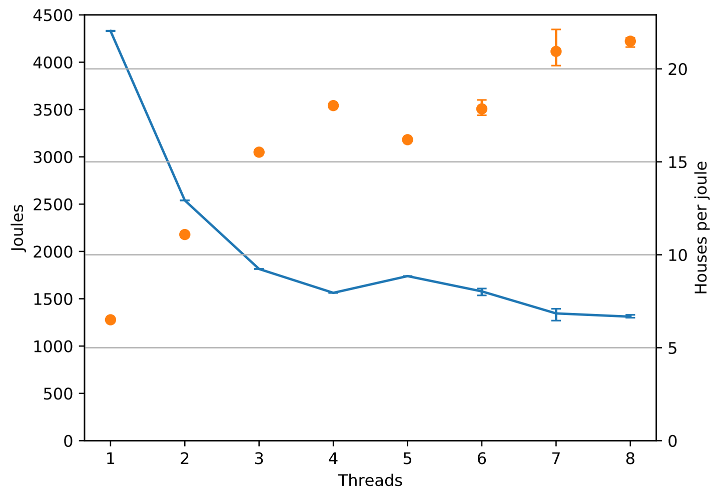


Figure 6.27: ParX CPU power consumption with increasing thread count.

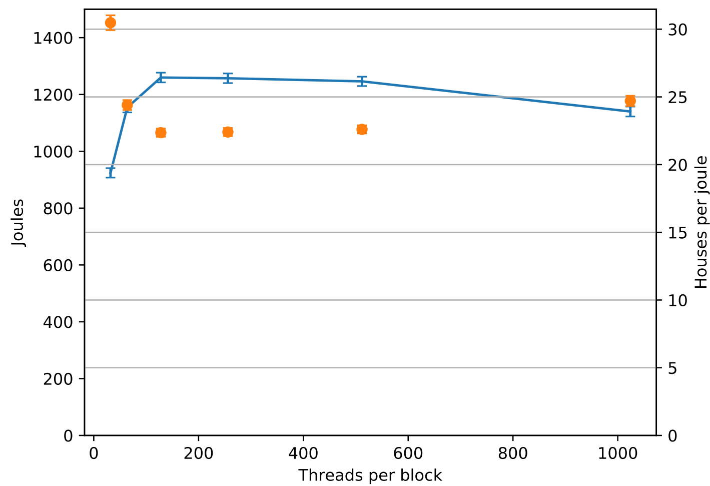


Figure 6.28: ParX GPU power consumption (1 thread/house) with increasing thread count.

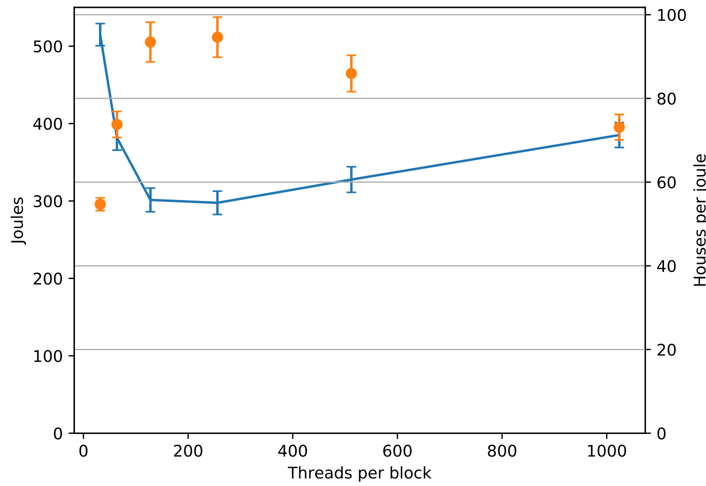


Figure 6.29: ParX GPU power consumption (24 thread/house) with increasing thread count.

6.6 Summary

Threads	Transfer to GPU		Extraction time		ParX		Transfer to CPU	
			CPU					
1			15.27	24%	49.51	76%	0.00	0%
8			2.35	15%	12.86	85%	0.00	0%
Threads / block			GPU (1 thread/house)					
32	0.38	6%	0.80	14%	4.75	80%	0.00	0%
Threads / block			GPU (24 threads/house)					
128	0.38	19%	0.43	22%	1.16	59%	0.00	0%

Figure 6.30: Breakdown of ParX execution times of 28165 houses. Time in seconds.

ParX, unlike histogram or similarity, does not have a sufficient amount of intrinsic parallelism that allow us to map a whole block of threads to a given house. Thus, one is stuck with 24 threads per house for maximum parallelism on a GPU. This implementation is both faster and more efficient from an energy point of view compared to CPU or GPU

(1 thread per house) implementation. There is some parallelism left over in solving the matrices when performing regression, however the matrix size is only 3×3, meaning that its parallelism is wasted. GPUs need a lot more parallelism with matrix sizes of 128×128 or above to be useful. According to Figure 6.30, extraction and transfer of data takes up a significant amount of execution time, so an asynchronous memory transfer can be helpful. Asynchronous memory transfer will however make the benchmark difficult to reproduce and instrument as the GPU driver will do some of the heavy lifting behind the scenes.

The plot in Figure 6.21 and others shows a drop in performance at 21 threads/house. The reason for such behaviour is unknown as the profiler crashed. A potential source of bad performance may be cache evictions as the GPUs uses a direct mapped cache and performance falls off a cliff if there is poor memory uitlization. Still, the best GPU implementation of ParX is 7× as fast and 5.2× more energy efficicient compared to the octa-threaded CPU implementation.

Chapter 7

Energy Disaggregation Model Benchmark

The energy disaggregation model [5] performs regression of the smart meter power data against the temperature for a given household. Unfortunately at any temperature one will get 10s to 100s of different power consumption data with large variations. This is caused by the fact that a given temperature can be present at different times of year (or different times in the same day) resulting in a widely fluctuating power consumption. Thus to remove any large variations, the 10th and the 90th percentile power consumption readings for a given temperature are used for regression. The 10th percentile represents the base load and the 90th percentile is the activity load.

Most of the households show a very common U-curve in power consumption. People use heaters at low temperatures and ACs at high temperatures which causes the increased power consumption at both ends of the spectrum. The slope of the two ends of the U-curve are heating and cooling gradient. Previously cubic spline [26] is used with manually defined boundaries. In this benchmark regression is done assuming that the power data has a cubic dependence on the temperature . The derivatives of the cubic polynomial (at the minimum and maximum temperatures) so obtained will provide the necessary cooling and heating gradients. The benchmark algorithm has two parts, sorting and regression.

This implementation differs from the original implementation as the original implementation uses segmented linear regression but here a polynomial (cubic) regression is performed. The segmented regression is a brute force approach to find the correct fit. However, if one is interested only in the heating and cooling gradient then polynomial

regression can be used as it is faster and easier to implement than segmented regression.

7.1 Implementation details

The core function of both sorting and the regression phase operates on a per house basis. These functions are common to both the CPU and GPU implementations. The only difference is that a single CPU thread loops through multiple houses but a GPU thread works on a single house. Similarly to all the previous algorithms, the array *offsets* will be used to locate both the temperature and power consumption data for a given house. The temperature range used in this implementation ranges from -10°C to 35°C. This temperature range is divided into 40 bins.

The sorting phase goes in two steps. The first step is the actual reallocating of power consumption data according to the corresponding temperature and adding it to the right bin. Now in the next step the energy data in each temperature bin is sorted. In this benchmark, a non-recursive version of merge sort is used for sorting. Once the sorting is complete, the 10th and the 90th percentile values for each bin are collected and put into another array named *tennine*. The *tennine* array is an array of structs that itself contains the index of the house *hid* and two arrays *tenvalues* and *ninevalues*. The 10th and the 90th percentile values are put in *tenvalues* and *ninevalues* arrays respectively.

The regression phase performs cubic regression of the energy data stored in the *tenvalues* and *ninevalues* arrays for each house with the temperature corresponding to each bin. The regression is done with the help of conjugate gradient. There is no extra parallelism used for this part of the benchmark as the matrices used are small (4×4). A standard implementation of the conjugate gradient algorithm is being used for solving and finding the polynomial coefficients.

7.2 Validation

This benchmark uses a different regression model than the the original benchmark. Hence the values generated by this workload will not match up with the Matlab implementation. In this case a few houses were sampled and regression was performed in Mathematica to ensure that the implementation is producing correct results. As usual, since Mathematica uses a different precision internally (defined by a parameter called 'Machine Precision') than the one used in this benchmark (single precision float), the values generated are

slightly different. Relative differences in the values are less than 0.01%. This validates the functional correctness of the benchmark.

7.3 CPU benchmarks

The benchmarks show the speed up with increasing thread count for 28165 houses. It contains the benchmark results of the different phases as explained before. It also includes the total speed up, which has the speed up of all the phases put together (extraction+sorting+coefficient generation). The threads are pinned to the CPU cores.

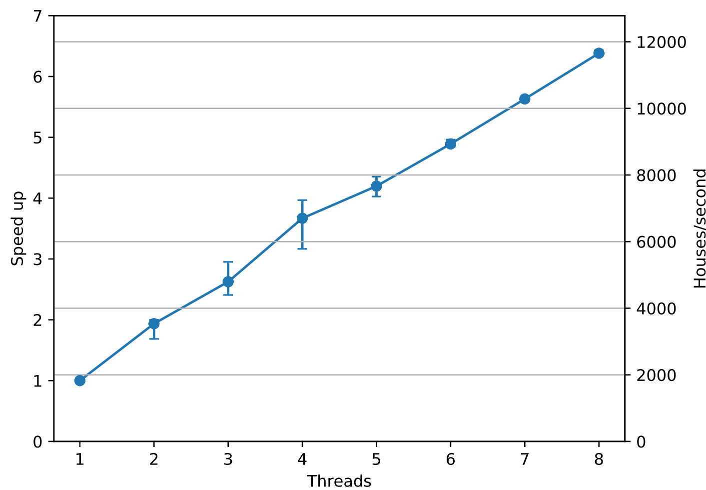


Figure 7.1: Extraction speed up.

Extraction as seen in Figure 7.1 has low ILP (instruction level parallelism) and thus shows increased speed up with more threads than physical cores. It shows near linear speeds up to three threads with the maximum speed up of 6 with 8 threads.

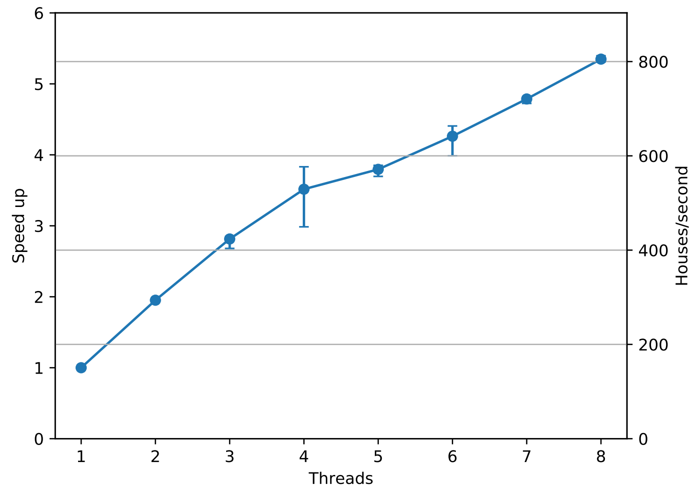


Figure 7.2: Sorting speed up.

This benchmark as seen in Figure 7.2 shows small variations even when the threads are pinned. Sorting has low ILP and is a very branchy workload as it uses comparison based sort. Hence increased thread count shows increased performance.

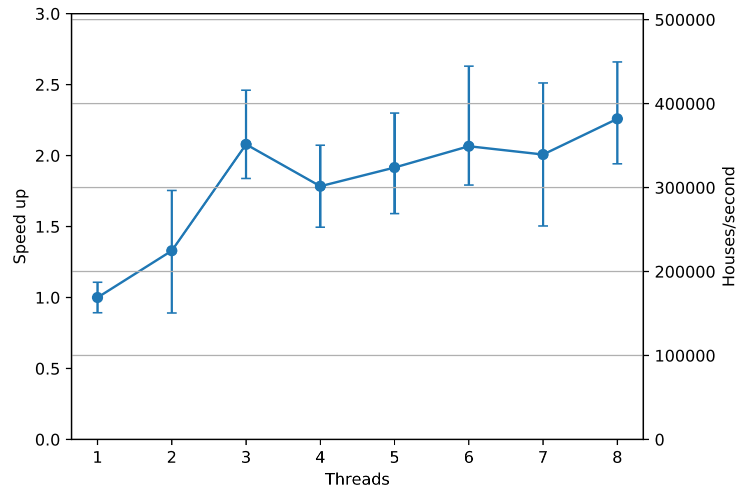


Figure 7.3: Coefficient generation speed up.

This benchmark as seen in Figure 7.3 high has variability in spite of pinned threads as the benchmark times are very short. The overhead of launching threads shows up in

larger thread counts and thus prevents enough speed up from occurring. This part of the benchmark takes up little time (Figure 7.17) and hence has little effect on the final results.

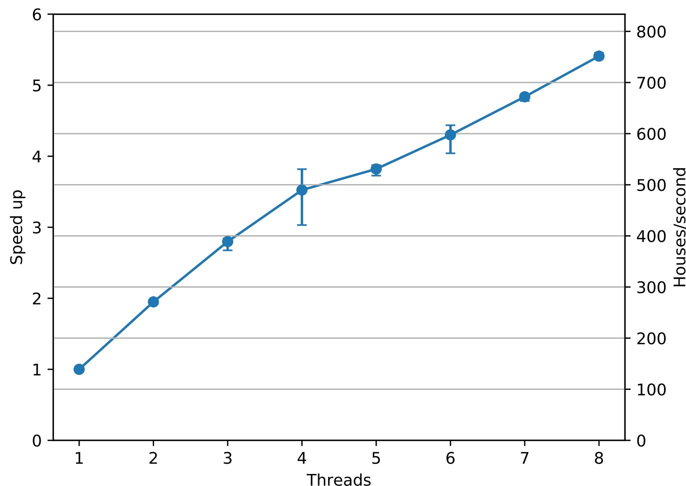


Figure 7.4: Total speed up, labels indicate the number of threads.

The total speed up as seen in Figure 7.4 closely mirrors the sorting plot as the sorting takes up majority of the benchmark time.

7.4 GPU benchmarks

The GPU implementation uses one GPU thread per house. The compute times and total times of the GPU are compared to the total time of the CPU's 1 and 8 threaded cases. The GPU benchmarks, like the CPU benchmarks, are done with 28165 houses.

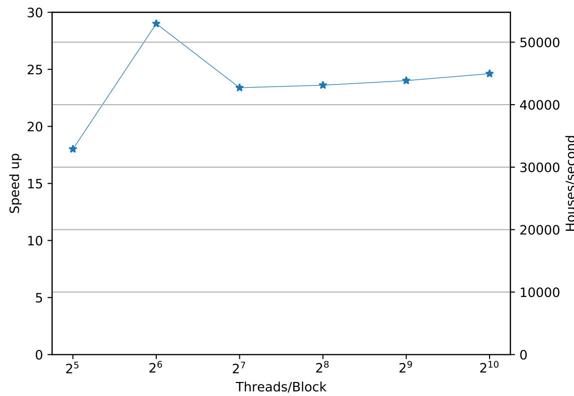


Figure 7.5: Speed up of GPU execution of extraction compared to single threaded CPU implementation.

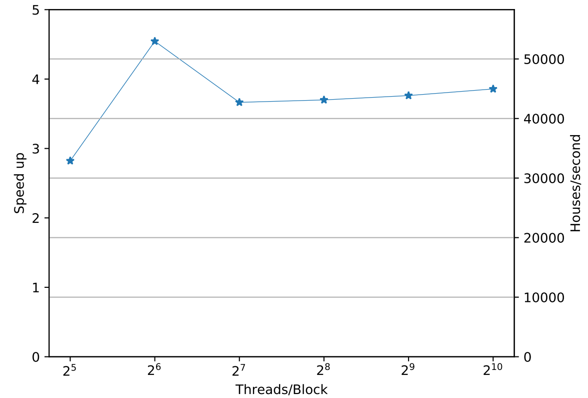
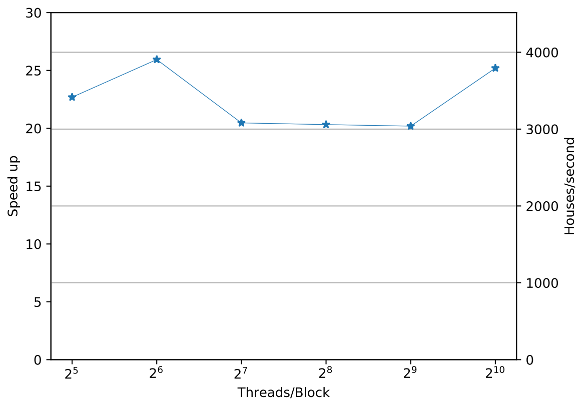


Figure 7.6: Speed up of GPU execution of extraction compared to octa threaded CPU implementation.

Immediately we see that in Figure 7.5 the difference due to increased memory pressure of the extraction. The new sweet spot is now 64 threads per block compared to extraction in previous benchmarks (Figure 4.4 of Histogram) where sweet spot was 1024 threads per block. The previous extraction times only extracted the power consumption data but here the data retrieved is both power consumption and temperature. Increase in the thread count causes more cache evictions and thus counter-intuitively means that lowering the thread count increases the performance.



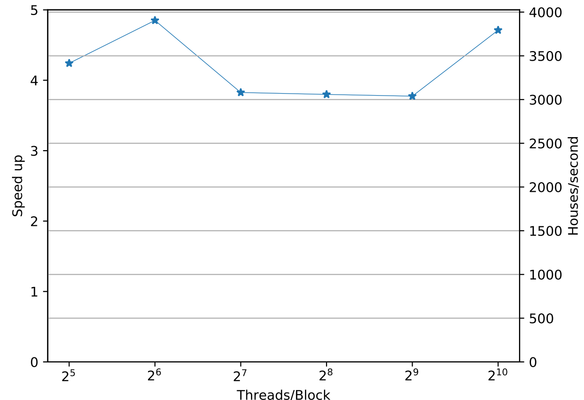
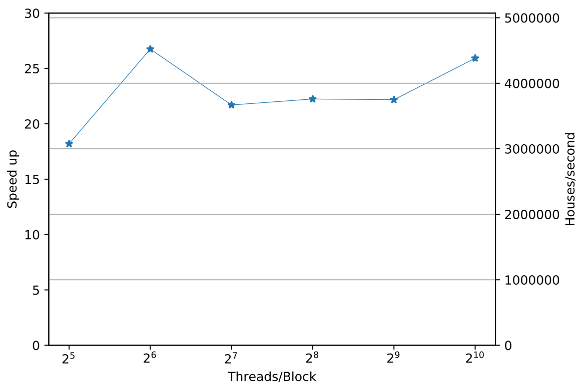


Figure 7.7: Speed up of GPU execution of sorting compared to single threaded CPU implementation.

Figure 7.8: Speed up of GPU execution of sorting compared to octa threaded CPU implementation.

This Figure 7.7 shows that 64 threads per block gives maximum performance when there is a large number of houses. Branch divergence is the main reason for the small speed ups. The branch divergence is caused by sorting (merge sort), which is comparison based.



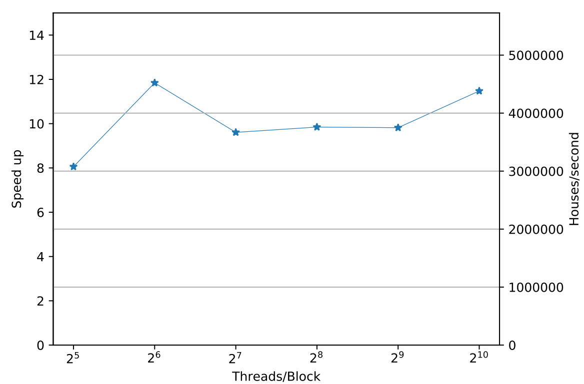


Figure 7.9: Speed up of GPU execution of coefficient generation compared to single threaded CPU implementation.

Figure 7.10: Speed up of GPU execution of coefficient generation compared to octa threaded CPU implementation.

Coefficient generation as seen in Figure 7.9 is compute heavy rather than memory

bound so increasing the thread count helps here as it allows us to do more computation. 64 threads per block is the sweet spot for this operation.

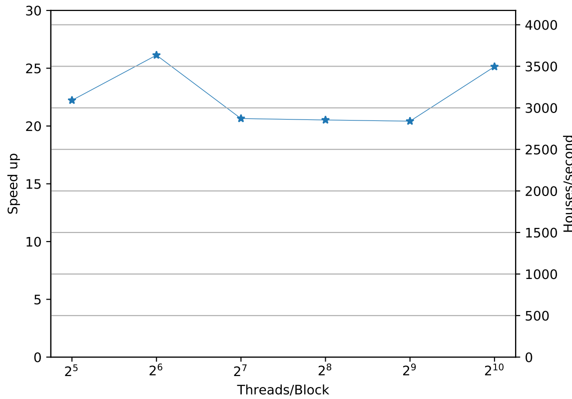


Figure 7.11: Speed up of GPU execution of compute compared to single threaded CPU implementation.

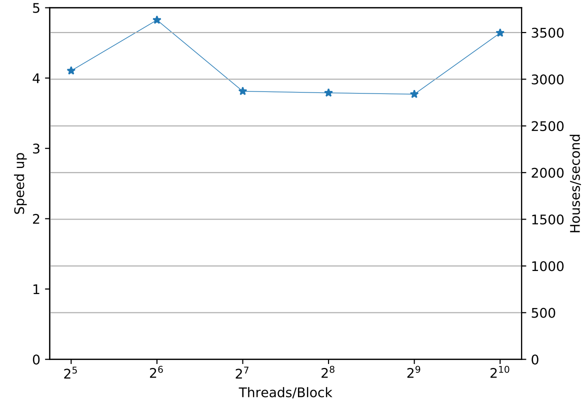


Figure 7.12: Speed up of GPU execution of compute compared to octa threaded CPU implementation.

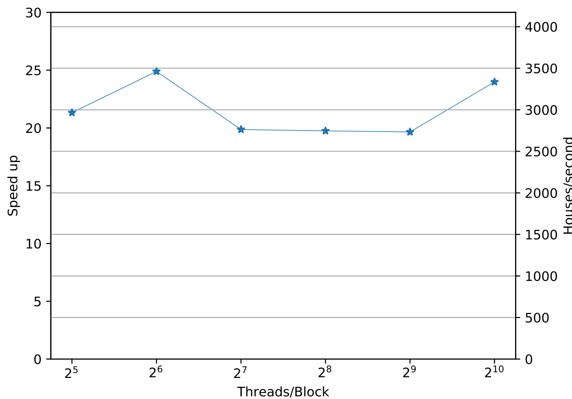


Figure 7.13: Speed up of GPU execution of total execution compared to single threaded CPU implementation.

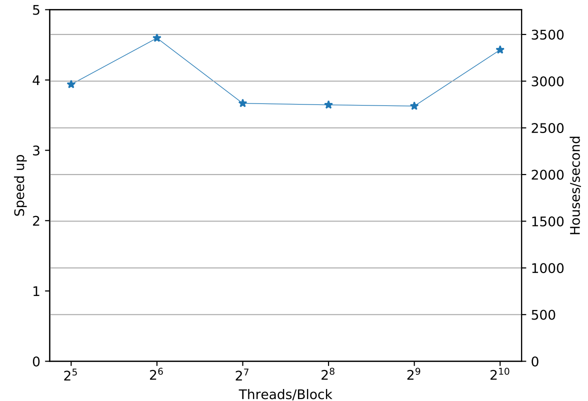


Figure 7.14: Speed up of GPU execution of total execution compared to octa threaded CPU implementation.

Both the compute (Figure 7.11) and total speed up (Figure 7.13) rather closely mirror the speed up of sorting as sorting takes up a large proportion (Figure 7.17) of execution time.

7.5 Power consumption

The CPU as seen in Figure 7.15 shows increased energy efficiency with larger thread count (3.5× efficient). All the different threads per block in the GPU as seen in Figure 7.16 show higher efficiency than any CPU implementation but the 64 threads/block shows maximum efficiency.

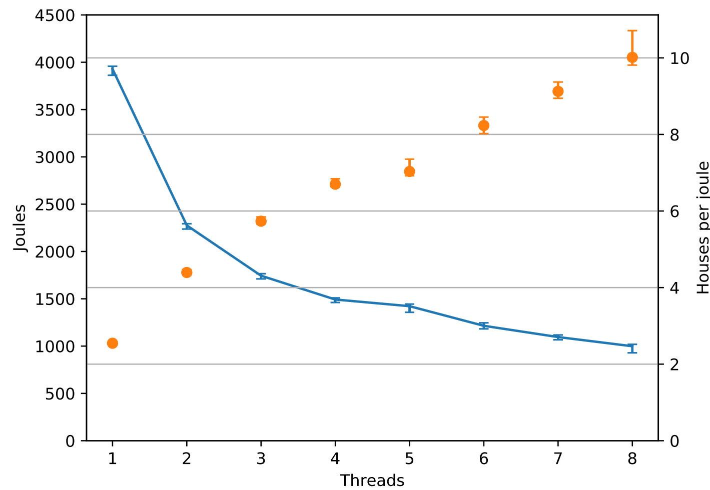


Figure 7.15: Power consumption of CPU in energy dissagregation.

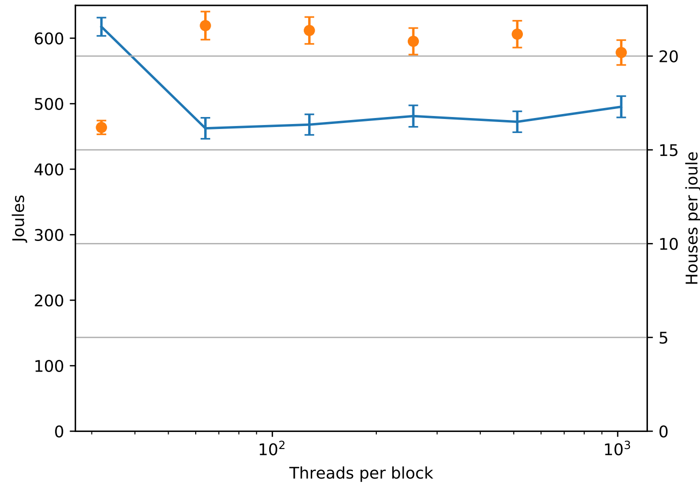


Figure 7.16: Power consumption of GPU in energy dissagregation.

7.6 Summary and discussion

	CPU									
Threads	Transfer to GPU		Extraction time		Sorting		Coefficient		Transfer to CPU	
1			15.46	9%	162.70	91%	0.15	0%		
8			2.53	7%	32.37	93%	0.05	0%		
Threads / block	GPU (1 thread/house)									
64	0.38	5%	0.53	7%	7.21	93%	0.01	0%	0.00	0%

Figure 7.17: Breakdown of Energy Dissagregation execution times of 28165 houses.Time in seconds.

A breakdown of the various parts of the benchmark in table 7.17 is shown. The benchmark is dominated by sorting the data. This benchmark is not very GPU friendly as the branch divergence of the sorting dominates the execution times. The interesting part is that the individual parts of the sorting phase like allocating the power data to the correct bin according to temperature and performing non-recursive merge sort are very much parallelizable. Unfortunately the parallelization is rather irregular with each operating

on different sizes of data which makes it very hard to map to the GPU programming model. Still one gets improved speed ($4.25\times$) and power efficiency ($2.25\times$) against the octa-threaded CPU version, which makes using the GPU-based implementations worthwhile.

Chapter 8

Discussion

This chapter completes the thesis. In broad strokes it can be seen that the chosen energy analytics benchmarks are mostly memory bound. Some of them show high memory divergence. Here some of the important observations are summerized as follows.

- Histogram: The histogram benchmark is very short and large parts of the execution times are dominated by transferring data and extraction of data. The GPU implementation which maps 1 block to a house improves the running time but other parts of the benchmark drag it down as far as speed ups are concerned. Since this is a very short and simple benchmark, I believe that no substantial improvements can be made. This benchmark is a memory bound benchmark.

- Similarity Search: The simple version of the GPU implementation which operates on 1 thread per house is very memory divergent and hence suffers from poor performance (though it is improved compared to CPU implementation). The GPU version which operates at 1 block/house shows large improvement due to better cache utilization as memory divergence is lowered. The similarity calculation takes up a large proportion of running times and hence the block version of the GPU is a viable choice for doing fast calculations. The other parts of the benchmark process like sending data to the GPU, extraction and finding the closest 10 matches take up negligible time. Substantial improvements to this benchmark are likely infeasible as it is completely memory bound.

- ParX: The previous two benchmarks had two variants of GPU implementation, 1 thread/house and 1 block/house. The latter type of implementation is not feasible

in ParX as this benchmark shows limited parallelism and is proved to be difficult to implement. The 1 thread/house implementation shows memory divergence like previous benchmarks. The maximum performance is obtained by using a 24 threads/house with each thread performing regression on each hour of the day. Changing the number of threads per house initially shows better performance due to improved parallelism and memory utilization, reaching a peak at 12 threads /house. Then it drops performance if the number of threads do not divide 24 evenly with performance reaching lowest values at 21 threads/house and goes back up again at 24 threads per house. This phenomenon needs to be investigated further as the profiler crashed when using large data sets. Improvement in this benchmark is possible especially in solving the 3×3 matrix and it may be possible to get a $3\times$ parallelism (as the elements of the solution vector can be computed in parallel), but extracting the possible parallelism seems very hard.

- Energy disaggregation: On a GPU this benchmark shows very different behavior compared to the previous benchmarks. It is not fully memory bound but suffers from branch divergence due to comparison based sort. This benchmark unlike the previous benchmarks depends on both temperature and power consumption data. Definite improvements may be possible in this benchmark by using a non-comparison based sort like radix sort but since the power data are floating point numbers some non-trivial changes needed to be made. This benchmark has parts which can be subjected to parallelism but on a whole, the parallelism is too irregular and it seems difficult to map it to the CUDA programming model. Improvements are also possible if specific parts of parallelism are targeted but it would be difficult. The sorting uses non-recursive merge sort which behaves in a top down fashion. Using an inplace version of quicksort was tried out earlier but the GPU suffered from stack overflow errors. Increasing the stack size stopped the problem but in that case the GPU ran slower than the CPU version!

All the GPU implementations discussed in the thesis followed a general rule of having 1 block/house faster than 1 thread/house which itself is faster than the 8 threaded CPU implementation. The power consumption data followed the same pattern. The single-threaded version performed poorest in all the benchmarks. Thus, increasing parallelism improved performance and energy efficiency of the benchmark but one has to carefully fine tune the number of threads per given block to get the best results. Determining the optimal value of the parameter is difficult and has to be found out in a trial and error basis.The need to tune such parameters can be considered a disadvantage in GPU computing. Another issue encountered when performing benchmarks has been compiler bugs,

which caused incorrect GPU assembly code to be produced. Profiler bugs are also present as using large data sets or using kernels which are a bit complicated (ParX and Energy disaggregation) will cause the profiler to crash.

In the end the energy analysis benchmarks ended up being sufficiently GPU friendly in spite of all the hitches mentioned. In the best case scenario, the GPU version of Histogram, Similarity, ParX and Energy disaggregation are 2.5×, 10×, 6× and 3.25× times faster than the octa-threaded CPU version. The power consumption of the GPU version of Histogram, Similarity, ParX and Energy disaggregaton are 3.6×, 8.2×, 5.2×, 2.75× smaller compared to the octa-threaded CPU version. Hence the work done in this thesis validates the usage of GPU for analysis of smart meter data.

References

[1] A. Albert and R. Rajagopal. Building dynamic thermal profiles of energy consumption for individuals and neighborhoods. In *Proceedings of IEEE International Conference on Big Data*, pages 723–728, 2013.

[2] R. C. Axtmann. Environmental impact of a geothermal power plant. *Science*, 187(4179):795–803, 1975.

[3] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012.

[4] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 577–578, 2010.

[5] B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands. Disaggregating categories of electrical energy end-use from whole-house hourly data. *Energy and Buildings*, 50:93–102, 2012.

[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG)*, 22(3):917–924, 2003.

[7] G. Chicco, R. Napoli, and F. Piglione. Comparisons among clustering techniques for electricity customer classification. *IEEE Transactions on Power Systems*, 21(2):933–940, 2006.

[8] J. Chwastowski, K. Korcyl, J. Płażek, and P. Poznański. Selected issues on histogramming on GPUs. *Computing & Informatics*, 35(2), 2016.

[9] P.H. Colberg and F. Höfling. Highly accelerated simulations of glassy dynamics using GPUs: Caveats on limited floating-point precision. *Computer Physics Communications*, 182(5):1120–1129, 2011.

[10] R. M. Cushman. Review of ecological effects of rapidly varying flows downstream from hydroelectric facilities. *North American journal of fisheries Management*, 5(3A):330–339, 1985.

[11] A. De Bono, P. Peduzzi, S. Kluser, and G. Giuliani. Impacts of summer 2003 heat wave in Europe. 2004.

[12] Jon G. Elerath and Sandeep Shah. Server class disk drives: how reliable are they? In *Proceedings of the Annual Symposium-RAMS on Reliability and Maintainability*, pages 151–156. IEEE, 2004.

[13] 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*, pages 133–137, 2004.

[14] K. Firth, S.and Lomas, A. Wright, and R. Wall. Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 40(5):926–936, 2008.

[15] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 3–3, 2005.

[16] P. C. Garnsworthy. The environmental impact of fertility in dairy cows: a modelling approach to predict methane and ammonia emissions. *Animal Feed Science and Technology*, 112(1):211–223, 2004.

[17] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil. An optimized approach to histogram computation on GPU. *Machine Vision and Applications*, 24(5):899–908, 2013.

[18] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, 2002.

[19] W. J. Hausman and J. L. Neufeld. Time-of-day pricing in the US electric power industry at the turn of the century. *The RAND Journal of Economics*, 15(1):116–126, 1984.

[20] P. D. Jones, K. E. Trenberth, P. Ambenje, R. Bojariu, D. Easterling, T. Klein, D. Parker, J. Renwick, M. Rusticucci, B. Soden, et al. Observations: surface and atmospheric climate change. *Climate change 2007: the physical science basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*, pages 235–336, 2007.

[21] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[22] A. Kubias, M. Deinzer, F.and Kreiser, and D. Paulus. Efficient Computation of Histograms on the GPU. In *Proceedings of the 23rd Spring Conference on Computer Graphics*, pages 207–212. ACM, 2007.

[23] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 55–55, 2001.

[24] X. Liu, L. Golab, W. Golab, I. F. Ilyas, and S. Jin. Smart Meter Data Analytics: Systems, Algorithms, and Benchmarking. *ACM Transactions on Database Systems (TODS)*, 42(1):2:1–2:39, 2016.

[25] G. A. Meehl and C. Tebaldi. More intense, more frequent, and longer lasting heat waves in the 21st century. *Science*, 305(5686):994–997, 2004.

[26] R. Miller. Analysis of weather, time, and mandatory time-of-use pricing effects on aggregate residential electricity demand. Master's thesis, University of Waterloo, 2015.

[27] A. Moravánszky. Dense matrix algebra on the GPU. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:352–380, 2003.

[28] A. Morrow and T. Cardoso. Why does Ontarios electricity cost so much? A reality check. *The Globe and Mail*.

[29] NVidia. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[30] A. L. Page, A. A. Elseewi, and I. R. Straughan. *Physical and chemical properties of fly ash from coal-fired power plants with reference to environmental impacts*. Springer, 1979.

[31] M. K. Patterson. Temperature on energy efficiency. In *Proceedings of the 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems.*

[32] M. Poess and R. O. Nambiar. Energy cost, the key challenge of today's data centers: a power consumption analysis of TPC-C results. *PVLDB*, 1(2):1229–1240, 2008.

[33] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, 2007.

[34] C. Rosenzweig, A. Iglesias, X. B. Yang, P. R. Epstein, and E. Chivian. Climate change and extreme weather events; implications for food production, plant diseases, and pests. *Global change & human health*, 2(2):90–104, 2001.

[35] R. Saidur, N. A. Rahim, M. R. Islam, and K. H. Solangi. Environmental impact of wind energy. *Renewable and Sustainable Energy Reviews*, 15(5):2423–2430, 2011.

[36] G. J. Tsekouras, N. D. Hatziargyriou, and E. N. Dialynas. Two-stage pattern recognition of load curves for classification of electricity customers. *IEEE Transactions on Power Systems*, 22(3):1120–1128, 2007.

[37] G. K. F. Tso and K. Yau. Predicting electricity energy consumption: A comparison of regression analysis, decision tree and neural networks. *Energy*, 32(9):1761–1768, 2007.

APPENDICES

Appendix A

Parallel Reduction

This is a very common technique to get the sum (or any binary operation which is associative and commutative) of all the array elements in a GPU with the use of shared memory. The table below shows the various stages of a summing up the elements of an array arr[] = 1,2,3,4,5,6,7,8. Since the array has 8 elements it takes $\log_2(8) = 3$ stages in completing the sum.

	1	2	3	4	5	6	7	8
Cycle 1	arr[a]=arr[a]+arr[a+4] when a < 4							
	6	8	10	12	5	6	7	8
Cycle 2	arr[a]=arr[a]+arr[a+2] when a < 2							
	16	20	10	12	5	6	7	8
Cycle 3	arr[a]=arr[a]+arr[a+1] when a < 1							
	36	20	10	12	5	6	7	8

Figure A.1: Parallel reduction stages

Now the 0th location of array arr[] will contain the sum of all the elements. Though this operation is shown with addition as it is the operation which has been used in these algorithms, any binary operator which is associative and commutative can be used for performing reduction. A few things to take note of:

- The first cycle needs 4 operations, the next cycle 2 operations and the last cycle 1 operation which makes a total of 7 operations. So it is very work efficient and performs the same number of operations as the sequential sibling.

- If we consider a single thread performing each operation, then after every cycle the threads need to synchronize. This means that the given algorithm may not map really well to CPU threads as the cost of synchronization is very high. It also means that for a GPU all the threads should be inside a block as only threads in a block can synchronize amongst themselves and inter block synchronization is not available.
- Since there can be a maximum 1024 threads for a given block, the maximum number of elements can be operated on in a GPU is 2048. However one can actually do reduction on larger number of elements by performing a sequential partial reduction (i.e. If there are 4096 elements then each thread can do sequential reduction of 2 elements at a time and then do parallel reduction on the 2048 elements).
- If the number of elements is not a power of 2, like an array with 27 elements then one can do the parallel reduction assuming the array has 32 elements (i.e. the next power of 2) and perform the required reduction carefully taking into account the boundary conditions.