An energy-aware design methodology based on kernel optimisations

Mehiar Dabbagh* and Hazem Hajj

Department of Electrical and Computer Engineering,
American University of Beirut,
Beirut, Lebanon
E-mail: mmd43@aub.edu.lb
E-mail: hh63@aub.edu.lb
*Corresponding author

Wassim El-Hajj

Department of Computer Science,
American University of Beirut (AUB),
Beirut, Lebanon
E-mail: we07@aub.edu.lb

Mohammad Mansour, Ayman Kayssi and Ali Chehab

Department of Electrical and Computer Engineering,
American University of Beirut (AUB),
Beirut, Lebanon
E-mail: mmansour@aub.edu.lb
E-mail: ayman@aub.edu.lb
E-mail: chehab@aub.edu.lb

Abstract: The goal of this paper is to present a design methodology for developing energy aware algorithms. The key idea revolves around identifying operations called kernels, which are frequently used operations in the algorithm that can be implemented in hardware. Optimising these kernels for performance or energy would then lead to a major impact in energy saving. We propose a six-step methodology for design of energy aware algorithms. The method includes: high-level algorithm analysis, identifying high frequency kernels, determining the order of computation for each kernel via asymptotic analysis, prioritising kernels in terms of energy impact, proposing alternative implementations to the kernels that cause high energy consumption and investigating further opportunities for energy optimisation specific to the studied algorithm. We further propose a simple and efficient method for estimating a kernel's energy cost. The method was successfully tested with back-propagation (BP) neural network algorithm to identify the kernels targeted for energy optimisation. Based on the findings, we proposed several custom changes to the BP algorithms for lower energy alternatives to kernels, including options that trade off computational accuracy for higher energy saving.

Keywords: DM; data mining; neural networks; back-propagation algorithm; energy aware.

Reference to this paper should be made as follows: Dabbagh, M., Hajj, H., El-Hajj, W., Mansour, M., Kayssi, A. and Chehab, A. (2014) 'An energy-aware design methodology based on kernel optimisations', *Int. J. Autonomous and Adaptive Communications Systems*, Vol. 7, No. 3, pp.271–294.

Biographical notes: Mehiar Dabbagh received his Bachelor degree in EE from the University of Aleppo, Syria in 2010. During his undergraduate study, he received two certificates of academic excellence from the University of Aleppo for his academic performance. He is currently a master student at the American University of Beirut (AUB). He is also a Research Assistant in Intel-KACST Middle East Energy Efficiency Research Centre (MER) at the American University of Beirut (AUB), where he works for developing energy efficient solutions in the application and compiler layers. His research interests include energy-aware computing, data mining, networks and security.

Hazem Hajj is an Assistant Professor with the American University of Beirut (AUB) since 2008. Before 2008, he was a Principal Engineer at Intel Corporation. At Intel, he led research and development for Intel's manufacturing automation, where he received several patents and numerous Intel Achievement Awards. On the academic front, he received his Bachelor degree in Electrical Engineering from AUB in 1987 with distinction and his PhD from the University of Wisconsin-Madison in 1996, where he also received several teaching awards, including the University Teaching Excellence Award. His research interests include data mining and energy-aware computing.

Wassim El-Hajj received his BS from the American University of Beirut in 2000 and MS and PhD in 2002 and 2006, respectively, from Western Michigan University (WMU), all in Computer Science. Currently, he is an Assistant Professor in the Computer Science Department at the American University of Beirut. He is the recipient of numerous recognitions, most notably, the WMU Excellence in Research Award for three years in a row and the Teaching Effectiveness Award which is considered the highest teaching award at Western Michigan University. His research interests include security, network planning and data mining.

Mohammad M. Mansour received his BE with distinction in 1996 and his ME in 1998 both in Computer and Communications Engineering from the American University of Beirut (AUB), Beirut, Lebanon. In August 2002, he received his MS in Mathematics from the University of Illinois at Urbana-Champaign (UIUC), Urbana, Illinois, USA. He received his PhD in Electrical Engineering in May 2003 from UIUC. He is currently an Associate Professor of Electrical and Computer Engineering with the ECE Department at AUB, Beirut, Lebanon. His research interests are VLSI design and implementation for embedded signal processing and wireless communications systems; coding theory and its applications; digital signal processing systems; and parallel computing systems.

Ayman Kayssi was born in Lebanon. He studied Electrical Engineering and received his BE with distinction in 1987 from the American University of Beirut (AUB) and the MSE and PhD from the University of Michigan, Ann Arbor, in 1989 and 1993, respectively. He received the Academic Excellence Award of the AUB Alumni Association in 1987. In 1993, he joined the Department of Electrical and Computer Engineering (ECE) at AUB, where he

is currently a Full Professor. In 1999–2000, he took a leave of absence and joined Transmog Inc. as a Chief Technology Officer. From 2004 to 2007, he served as Chairman of the ECE Department at AUB. He teaches courses in electronics and in networking and has received AUB's Teaching Excellence Award in 2003. His research interests are in information security and in integrated circuit design and test. He has published 140 papers in the areas of VLSI, networking, security and engineering education. He is a senior member of IEEE and a member of ACM, ISOC and the Beirut OEA.

Ali Chehab received his Bachelor degree in EE from the American University of Beirut (AUB) in 1987, Master's degree in EE from Syracuse University in 1989 and PhD in ECE from the University of North Carolina at Charlotte in 2002. From 1989 to 1998, he was a Lecturer in the ECE Department at AUB. He rejoined the ECE Department at AUB as an Assistant Professor in 2002 and became an Associate Professor in 2008. His research interests are VLSI design, VLSI testing and information security and trust.

1 Introduction

The large widespread of mobile devices has made energy an increasing importance in today's technology. Mobile devices are battery dependent and are required to stay charged for a long period of time, while battery technology is developing at a slow pace. Energy is an issue that is not only limited to mobile devices. In fact it has become a major constraint, named the power wall (Kogge, 2011), in improving computer's performance. Reducing the consumed energy in electronic devices is crucial since a part of the consumed energy is transformed into heat which affects the reliability of these devices and requires additional costs for cooling. This has become an important issue especially after Moore's law started hitting against power density barriers, where the heat density on a device could reach heat levels close to what would be found on the surface of the sun. Furthermore, energy costs in large computer centres are increasing from financial and environmental aspects. In fact, it was noted in Pettey (2007) that information and communication technology (ICT) is responsible for 2% of the global emissions, equivalent to aviation. Energy consumption is also financially very important as the energy use of ICT is expected to double by 2020 and triple by 2030. These energy challenges have driven the search for efficient ways to save energy.

Reduction of energy can be achieved by optimising components at the platform level, or examining different computer layers and their interactions, including hardware, architecture, compiler, operating system and application. This paper explores energy awareness and potential optimisations starting from the application level, but with the goal of integrating them within lower layers of the system through low-energy kernels provided from the compiler, operating system, architecture or hardware.

Our proposed methodology consists of six steps that can be summarised as follows:

- *High-level analysis:* The objective of this step is to get a preliminary understanding of the flow of the studied algorithm.
- *Kernel identification:* In this step we determine the algorithm's kernels. We define a kernel as an operation that is frequently executed in the algorithm.

- *Algorithm asymptotic analysis:* The objective of this step is to determine the order of the kernels based on the properties of the algorithm.
- Measuring the energy of the kernels and prioritising them based on their energy impact. To accurately accomplish this step, we propose a simple approach that can be adapted by researchers to measure the energy of kernels using either simulation or physical measurements. Then we prioritise kernels based on their energy contribution to the overall energy cost.
- *Usage of alternative lower energy kernels:* In this step we propose solutions to reduce the energy of the kernels with the highest contribution to energy consumption.
- Investigating other opportunities of energy reduction specific to the algorithm under consideration.

For instance, we inspect how data preprocessing techniques can make computations simpler leading to energy savings. As a case study, we apply our suggested methodology to neural networks back-propagation (BP) (Haykin, 1998; Han and Kamber, 2006) since it is a good example of a compute-intensive application that is widely used in many domains. This paper is an extension to our previous work (Dabbagh et al., 2011), where we have tested the six-step methodology for BP, but the approach was limited to counter estimates. This paper extends the method by providing an approach for measuring actual kernel energy and then applying these measured energies instead of estimated counters. The experiments were repeated with the more accurate energy measurements and supported the success of the method. Our results show that normalising the training set before the learning phase makes the algorithm converge faster with fewer iterations resulting in significant energy savings. More energy saving techniques specific to a given algorithm are discussed in Section 4.

The paper is organised as follows: In Section 2, we review the previous research that was done to optimise and measure energy in computers and show how our work provides new opportunities for energy savings. In Section 3, we explain in details the different steps of our proposed methodology for analysing any algorithm and choose BP as a case study. We also propose in this section a simple and efficient approach to measure the kernel's energy cost as it is a major step in our analysis. In Section 4, we discuss our experimental setup and simulation results using Simplescalar and WATTCH tools which show that a significant amount of energy reduction can be achieved by applying approximation and preprocessing techniques. In Section 5, we conclude the paper and present our future work.

2 Related work

Energy optimisation techniques have been suggested for the various layers of the computer platform. In this section, we overview these techniques in addition to describing the methods used to measure platform energy. We end the section by highlighting how our suggested energy optimisation and energy measurement methods differ from the other works.

2.1 Overview of energy optimisation techniques

The increasing demand for saving energy has made researchers go beyond the low-level circuit layer to explore optimisations in the upper layers including the compiler, the operating system and the application layers.

In the compiler layer, compiler optimisation techniques are usually used to improve performance by reducing the number of cycles that are required to execute a program. These techniques can be also used to save energy. In Daud et al. (2008), Brooks et al. (2000), Sinevriotis and Stouraitis (2002) and Wiratunga and Gebotys (2000), different compiler optimisation techniques (e.g., loop unrolling and instruction rescheduling) were applied on benchmarks in order to save energy. The main limitation of these techniques is that their effect on both performance and energy varies from one code to another. The space of possible compiler optimisation techniques is huge. Consequently, determining the best combination of optimisation techniques using brute-force search is infeasible. Therefore, heuristic methods are used to determine the combination of techniques with the highest energy saving/highest performance for a certain code. An example of such an approach was presented in Desai (2009) where the authors proposed a heuristic method that prunes unpromising optimisation techniques to reduce the search space. Another attempt to overcome the problem of the large search space was presented in Malik (2010), Dubach et al. (2009), Hung et al. (2009) and Cooper et al. (2001). The authors used machine learning (Malik, 2010; Dubach et al., 2009; Hung et al., 2009) and genetic algorithms (Cooper et al., 2001) to predict the best combination and the best sequence of compiler optimisations such that the energy or the delay for running a given code is minimised.

In the operating system layer, researchers worked on optimising and efficiently using the different power management policies that are used to control power. These policies switch idle devices into lower power states if they predict that the full capacity of these devices will not be needed for the coming events. It is worth noting that in OS power policies, the device is not switched into a lower power state whenever it is idle because the energy required to wake the device up is high. Therefore, the device is switched into a lower power state only if we predict that it will remain idle for a long time that is enough to compensate for the transition energy. Additional information about how these predictive techniques and algorithms work is provided in Lu and De Micheli (2001) and Albers (2010). John et al. (2010) presents a comparison between the power policies in Windows 7 and Windows Vista. Results showed that Windows 7 achieved higher energy savings due to more available low power-states and better idleness predictive techniques. Another technique that operating systems use to manage power is dynamic voltage and frequency scaling (DVFS). In DVFS, the operating voltage and frequency for executing a task are reduced in order to save the consumed energy. DVFS leads to a great saving of energy but has a negative effect on performance (Kaxiras and Martonosi, 2008).

In the application layer, sensor network applications (Dutta and Culler, 2005), WiMAX frame construction (Abbas et al., 2011), multimedia (Darwish and Chabukswar, 2009), bioinformatics (Pawaskar and Ali, 2010) and file access (Steigerwald et al., 2008) are some of the applications that underwent energy optimisation attempts. The optimisation methods that were proposed are classified according to Larson (2008) into three main categories: contextual awareness, data efficiency and computational efficiency.

In contextual awareness, applications change their behaviour based on the available energy. A dynamic compilation that adapts battery changes is proposed in Unnikrishnan et al. (2002), where precompiled parts of the code that have low energy are used when the battery is low.

In data efficiency techniques, data is stored, accessed and transferred in an energy efficient way. For instance, in Schall et al. (2010) the authors showed that considerable energy savings can be achieved by using SSD instead of HDD as a storage device.

In computational efficiency, the maximum available system capabilities are used so that the work finishes early, enabling the devices to be switched into idle mode to save energy. Examples of computational efficiency techniques include the use of parallel programming to reduce the execution time or any other technique that enhances performance. The authors in Ge et al. (2010) implemented a framework for profiling the energy of different applications. The authors used parallel programming and examined the number of required parallel nodes such that the overall energy for running different benchmarks is minimised. Examples of computational efficiency also include the work in Chu et al. (2006) and Negrevergne et al. (2010) where the authors proposed parallelising different machine learning and Data Mining (DM) algorithms by distributing calculations on different cores in order to improve runtime performance.

Researchers in the field of DM have focused on improving accuracy and performance of DM algorithms. To the best of our knowledge, no previous work has analysed these algorithms from the energy efficiency point of view. Although, the attempts to enhance performance lead into energy savings indirectly by maximising the idle time of devices, we show in this paper that further opportunities for energy savings are possible in the application layer. Our proposed optimisations can be applied on top of the mentioned energy optimisations at the different layers (compiler, operating system and computational efficiency optimisations) to achieve further energy savings.

The proposed methodology is applicable on any algorithm and aims to find and exploit energy saving opportunities. In our work, we demonstrate the different steps of our methodology by applying them on DM algorithms. The reason behind choosing DM algorithms is twofold. First, DM algorithms are widely used in many domains (bioinformatics, business, social networks, etc.). The second reason is due to the nature of DM algorithms. During the training phase, DM algorithms usually have a segment of code that is repeatedly executed for different tuples. Therefore, optimising the energy of these segments will be highly reflected on the overall energy consumption. We introduce a methodology for analysing any DM algorithm from the energy consumption point of view rather than performance point of view. Then, we apply our proposed methodology to the BP algorithm. Unlike all the optimisation techniques used to reduce energy in general applications [computational efficiency (Ge et al., 2010; Chu et al., 2006; Negrevergne et al., 2010), data efficiency (Schall et al., 2010) and power-aware behaviour (Unnikrishnan et al., 2002)], our technique takes advantage of the algorithm specifics and investigates the opportunities that might lead to energy savings. For instance, we updated the BP algorithm to use approximation techniques via lookup tables and preprocessing techniques via normalisation leading to considerable improvements in both performance and energy.

2.2 Overview of energy measurement techniques

Although the work that attempts to optimise energy consumption is very important, estimating the consumed energy is as important. This is due to the fact that energy estimation methods provide insight into the power consumption of the different parts of the code and into the different components of the architecture. It also allows researchers to assess and compare the energy savings that can be achieved by their proposed optimisation techniques.

All the previously mentioned work used one of two ways to estimate the energy required to run a program on a given architecture:

- Physically measuring the current, frequency and voltage of the different components using ammeters and special acquisition systems. This method has high accuracy but is expensive since it requires special equipment.
- Simulating the behaviour of running the program on the architecture: this method is inexpensive and it allows users to analyse the performance and power behaviour with a cycle level of granularity for the different components. However, it is less accurate than the physical measurements, but it is a good solution when the special equipment and boards of a certain architecture are not available.

In Marcu et al. (2009), the authors proposed different frameworks for providing energy and thermal profiles. In Hu et al. (2005), the authors proposed solutions to reduce the time required to estimate the energy cost of long code segments using clustering techniques. None of the previous works targeted estimating the energy of the kernels for the algorithms. Hence, as will be shown later, studying the kernels and reducing their energy cost plays a major role in reducing the energy of the whole algorithm. This fact led us to propose a simple and efficient approach for estimating the energy cost of these kernels. Our proposed measuring approach can use either physical measurements or simulation tools to estimate the energy of kernels.

In the following section we present the different steps of our proposed methodology to analyse DM algorithms from the energy point of view.

3 Proposed methodology

We explain in the first part of this section our proposed methodology for an energy-aware algorithm. Our proposed methodology consists of six steps. In order to illustrate these steps, we apply the proposed methodology on BP NN algorithm as a case study. In the second part of this section, we explain how to measure the energy of kernels which is an important step in our proposed methodology towards an energy-aware algorithm.

3.1 Methodology for an energy-aware algorithm

The six steps are proposed below, with particular emphasis on kernel-based energy analysis and optimisation. The purpose of these steps is to analyse the program details and identify opportunities for reducing energy consumption. Figure 1 shows the flow of our proposed six-step methodology.

Step 1: High-level program analysis: The program is examined for the major computational structures, including loops, sequential parts and parallel parts of the algorithm. The objective of this step is to get a preliminary understanding of the execution elements of the code so that future steps can enhance the code details and make it more energy efficiency. Figure 2 is a pseudo code for the general flow of the BP algorithm. The algorithm consists of an initialisation step (line 1) followed by a 'for' loop. The body of the loop contains two sequential stages: A forward stage and a backward stage. In the forward stage (line 5), we calculate the output of each neuron in every layer. In the backward stage (line 7), we calculate the error between the obtained output and the desired output from the last to the first layer. Finally, we readjust the weights and biases (line 8) such that the error is minimised.

Figure 1 Proposed six-step methodology for creating an energy-aware algorithm

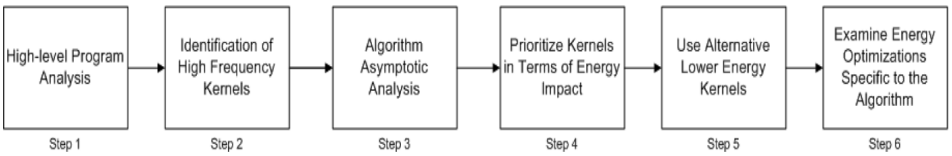


Figure 2 Back-propagation algorithm

Algorithm:
Backpropagation for classification & prediction.
Input:
- *D*, a data set consisting of the training tuples and their associated target value.
- *N*, number of iterations.
- *Network*, a multilayer feed-forward network.

Output: A trained neural network.
Method:
(1) Initialize all weights and biases in *Network*;
(2) **For** i=1 **to***N* {
(3) Pick a training tuple from *D*;
(4) // Forward phase;
(5) Calculate the output of each Neuron From first to last layer in *Network*;
(6) // Backward phase;
(7) Calculate the error from last to first layer in*Network*;
(8) Update the weights and biases of *Network* so that the error between the desired output and *Network* output is minimized;
(9) }

Step 2: Identification of high frequency kernels: Kernels are operations that can be implemented in hardware and are executed repeatedly in each iteration. A typical DM algorithm has repeated mathematical operations, producing a specific set of kernels. Some basic kernels include: addition, multiplication, division, subtraction and count. Other higher level kernels may include: logarithm, exponential, information gain, Euclidean distance and many others. These operations could be executed on vectors, arrays or single elements depending on the algorithm.

From Figure 2, we can observe that reducing the cost of any step in the body of the loop (lines 3–9) will definitely reduce the overall energy since the body of the loop is repeated N times (usually $N \geq 5{,}000$ for BP). Table 1 shows the equations that are executed in the loops of the forward and backward phases and the corresponding kernels. Let S_k^p and y_{jk}^{p-1} be the input and output to the unit k (neuron) in layer p and $p-1$, respectively, w_{jk} be the weight of the connection from unit j to unit k, b_k be the bias of unit k, $e_j(n)$ is the error in unit j at layer n, $d_j(n)$ the desired output of unit j at layer n, $y_j(n)$ be the jth unit output at layer n, Δw_{jk} be the change in weight of the connection from unit j to unit k, γ be the learning rate, δ_k^p be the error in unit k at layer p and , w'_{jk} be the new weight of the connection from unit j to unit k. From Table 1 we can determine the kernels of the BP algorithm: multiplication, division, addition, subtraction and exponential operations.

Table 1 BP equations and kernels

Forward phase	
Equation	*Kernels*
$S_k^p = \sum_j w_{jk} y_{jk}^{p-1} + b_k$	Multiplication
	Addition
$y^p = \frac{1}{1+e^{-sp}}$	Division
	Addition
	Exponential function
Backward phase	
Equation	*Kernels*
$e_j(n) = d_j(n) - y_j(n)$	Subtraction
$\Delta w_{jk} = \gamma \delta_k^p y_j^p$	Multiplication
$w'_{jk} = w_{jk} + w_{jk}$	Addition

Step 3: Algorithm asymptotic analysis for kernel energy and computation growth rates: The purpose of this step is to examine the effect of the different kernels and computations on the energy and performance of the algorithm as the data size grows and as the properties of the algorithm change. This step similar the standard algorithm asymptotic analysis conducted for performance, but applied here for energy.

The analysis consists of:

- Counting the number of times each kernel is executed while accounting for size of arrays or vectors in the operations.
- Calculating the energy cost of each kernel. We present in the following subsection a methodology that can be used to estimate the energy cost of kernels using either physical measurements or simulation tools.
- Calculating the overall energy of the algorithm.
- Determining the energy contribution of each kernel to the overall energy.

For the NN algorithm, the following assumptions and notations are made:

- *IN* represents the number of input neurons in the input layer which is equal to the number of attributes in each input tuple.
- *HN* represents the number of hidden neurons in the hidden layer.
- *ON* represents the number of output neurons in the output layer.
- The non-linear activation function for the hidden layer, the sigmoid function, is given by equation (1), where S^p and y^p are the input and output of the function.

$$y^p = \frac{1}{1+e^{-S^p}}. \tag{1}$$

- The linear activation function given in equation (2) is used for the output layer.

$$y^p = S^p. \tag{2}$$

- The training phase is repeated for N iterations. The number of iterations in the main loop is usually proportional to the data size. So the number N is also representative of data size.
- The variables that are used in the high-level language code of the algorithm are defined in Table 2. Columns two and three represent the number of rows and columns, respectively for array variables.

To conduct the asymptotic analysis, we count the number of times each kernel is executed in every line of the high-level language code for one iteration of the algorithm (Table 3).

For example, the first row in Table 3 breaks the first line of the code $\boldsymbol{s1} = \boldsymbol{x} * \boldsymbol{w1} + \boldsymbol{b1}$ into its basic kernels. The first part $\boldsymbol{x} * \boldsymbol{w1}$ is a matrix multiplication in which we multiply *x*-whose size according to Table 2 is $(1 \times IN)$-by $\boldsymbol{w1}$ which is a matrix of size $(IN \times HN)$. Multiplying $\boldsymbol{x}$ by $\boldsymbol{w1}$ consists of $IN \times HN$ multiplication operations and $(IN - 1) \times$HN additions operation and the resulting matrix has a size of $(1 \times HN)$. The resulting matrix is then added to $\boldsymbol{b1}$ whose size is also $(1 \times HN)$ and this step involves *HN* additions operation. As a result $\boldsymbol{s1} = \boldsymbol{x} * \boldsymbol{w1} + \boldsymbol{b1}$ consists of $IN \times HN$ multiplication operations and $(IN - 1) \times HN$ addition operations for matrix multiplication and *HN* addition operations for summing the resulting matrix to $\boldsymbol{b1}$ as shown in Table 3. The same procedure is applied to the rest of the code.

Table 2 Name and definition of BP code variables

Name	*Num. of rows*	*Num. of col.*	*Description*
X	1	*IN*	Input neurons
w1	*IN*	*HN*	Weights from the input to the hidden neurons
b1	1	*HN*	Bias for the hidden neurons.
w2	*HN*	*ON*	Weights from the hidden neurons to the output neurons
b2	1	*ON*	Bias for the output neurons
s1	1	*HN*	Input to the hidden neurons
y1	1	*HN*	Output of the hidden neurons
So	1	*ON*	Input of the output neurons
Yo	1	*ON*	Final output of the NN
D	1	*ON*	Desired output
E	1	*ON*	Error used to change the weights
delta_o	1	*ON*	Delta for output layer
delta_h	*HN*	1	Delta for hidden layer
Gama	1	1	Learning rate
d_w1	*IN*	*HN*	Delta for the weights of the hidden neurons
d_b1	1	*HN*	Delta for the bias of the hidden neurons
d_w2	*HN*	*ON*	Delta for the weights of the output neurons
d_b2	1	*ON*	Delta for the bias of the output neurons

Table 3 Number of executed kernels for each instruction in one iteration

Line of high-level code	*Matrix multiplication*						*Exp.*
	×	+	×	÷	+	–	
s1 = x * w1 + b1	IN × HN	(IN – 1) × HN	0	0	HN	0	0
y1 = 1/(1 + exp (–s1))	0	0	HN	HN	HN	0	HN
So = y1 * w2 + b2	HN × ON	(HN – 1) × ON	0	0	ON	0	0
e = d – yo	0	0	0	0	0	ON	0
delta_h = w2 * delta_o'. * (y1'. * (1–y1'))	ON × HN	(ON – 1) × HN	2 × HN	0	0	HN	0
d_w1 = gama * x' * delta_h'	IN × HN	0	IN	0	0	0	0
d_b1 = gama * delta_h	0	0	HN	0	0	0	0
d_w2 = gama * y1' * delta_o	ON × HN	0	HN	0	0	0	0
d_b2 = gama * delta_o	0	0	ON	0	0	0	0
w1 = w1 + d_w1	0	0	0	0	IN × HN	0	0
w2 = w2 + d_w2	0	0	0	0	HN × ON	0	0
b1 = b1 + d_b1'	0	0	0	0	HN	0	0
b2 = b2 + d_b2	0	0	0	0	ON	0	0

From Table 3, it is possible to calculate the energy consumption for *NN* with *IN* input neurons, *HN* hidden neurons, *ON* output neurons and *N* iterations. The number of times each kernel is executed per iteration of the main loop can be determined by summing the number of times each kernel in Table 3 is executed for all the lines of the code for one iteration. The results are shown in the following equations:

$$N_{\times} = 2\times IN\times HN + 3\times ON\times HN + 5\times HN \times ON \tag{3}$$

$$N_{+} = 2\times HN\times IN + HN + 3\times HN\times ON + ON \tag{4}$$

$$N_{\div} = HN \tag{5}$$

$$N_{\exp} = HN \tag{6}$$

$$N_{-} = ON + HN \tag{7}$$

where $N_{\times}$, N_{+}, $N_{\div}$, $N_{\exp}$ and N_{-} are the total numbers of multiplications, additions, divisions, exponentials and subtractions per iteration, respectively.

For *N* iterations, the total numbers in equations (3)–(7) are magnified *N* times and therefore we can determine the total order of computation for each kernel as shown in Table 4. Once the asymptotic analysis is complete, we can assess the relative energy consumption for the different kernels and determine which kernels have the highest impact on energy. In Section 4, we show simulation experiments indicating the relative energy impact of using alternative lower-energy kernels.

Table 4 Order of BP kernels

Kernel	*Order*
×	$\theta(N \times [2 \times IN \times HN + 3 \times ON \times HN + 5 \times HN + ON + IN])$
÷	$\theta(N \times [2 \times HN \times IN + HN + 3 \times HN \times ON + ON])$
÷	$\theta(N \times HN)$
–	$\theta(N \times [ON + HN])$
Exp	$\theta(N \times HN)$

Step 4: Prioritise kernels in terms of energy impact: This step helps in knowing which kernels should be targeted for maximising the reduction of energy consumption of the overall algorithm.

In order to prioritise kernels, we study how much saving in the overall cost we can achieve if we reduce each kernel's cost by the same amount. The one that gives the highest overall energy reduction is the best target for energy reduction. In Section 4, we use simulations to get an insight into the priority energy kernels.

Step 5: Use alternative lower energy kernels: At this stage, we use the results of step 4 to determine various ways to reduce the energy cost for priority kernels. Potential approaches include:

- Using approximations to the kernels with less computations and lower energy with a tradeoff of lower accuracy for better performance and energy. In this scenario, it is important to assess the error resulting from the approximation to make sure that the algorithm still yields acceptable results.

- Using alternative hardware implementation of the instruction set with lower energy and without compromising performance. For example a totally new optimised instruction set such as those used in DSP processors can be utilised.

- Parallelising the code to improve the performance of the algorithm. In this scenario, it is important to ensure that the performance improvements lead to energy improvements.

In Section 4 we propose an approximation technique for the exponential kernel using lookup tables.

Step 6: Examine other opportunities of energy reduction specific to the algorithm under consideration: This could be preprocessing the data to simplify algorithm computations. Examples of preprocessing techniques include data normalisation and data reduction. The most important aspect when applying these preprocessing techniques is to make sure that the energy cost of preprocessing data in addition to the energy cost of running the algorithm for the pre-processed dataset is smaller than the energy cost of running the algorithm on the original dataset without preprocessing. We should also make sure that running the algorithm on the preprocessed dataset yields results that are the same as or very similar to those obtained from original dataset. In Section 4 we show that normalising the training set of BP NN algorithm results in notable energy savings.

3.2 Methodology to estimate the energy of the kernels

Once we determine the kernels of the algorithm (Step 3 in our methodology), we need to prioritise kernels in order to get an insight on which kernel we need to target for optimisation; hence we will target the kernels whose optimisations achieve the highest saving in energy (Step 4). Our focus in this subsection is to find an accurate yet efficient way to measure the energy consumed by the kernels and that would help in the prioritisation. The main idea behind our methodology is running a code in which the kernel operation is executed a number of times N and estimating its energy cost. Since the code includes not only the kernel operation but also the initialisation of the variables, we remove the kernel operation from the code and we estimate the energy cost of the base code without the kernel operation. The process is then repeated with the kernel back in the code. Finally, we subtract the energy cost of the code without the kernel from the code with the kernel and divide the obtained result by N ending up with the energy cost of executing one kernel operation.

Furthermore, we look at different ranges of possible parameters for the kernel. The pseudo code shown in Figure 3 is used to determine the energy of the kernel operations that are executed on numbers in the range between *Min_range* and *Max_range*. For example, if we are trying to measure the energy cost of the exponential function kernel, $\exp(x)$, this energy cost is dependent on the value of x. If we know that x in the algorithm is between 0 and 1,000, then we need to estimate the energy cost of calculating the exponential function for the numbers in the range [0, 1,000]. We declare variable x as an array (line 4) and store in it the numbers between 0 and 1,000 inclusive. Since there are infinite numbers in that range, we use a variable 'step' to control the granularity between consecutive numbers. For example, if step is equal to 1 (difference between each two

consecutive values is 1), then we calculate the energy cost of calculating the exponential function of the numbers (0, 1, 2, …., 1,000).

Figure 3 Code used to determine kernels energy cost

```
1)   Collect start-time;
2)   intarray_size = (Max_range - Min_range) / step + 1;
3)   //Declare the variables for a certain range
4)   floatx[array_size]; float y[array_size];
5)   //Give the variables values in the studied range
6)   For(int i=0; i<array_size;i++) {
7)   x[i]=Min_range + i*step;
8)   Y[i]=Min_range + i*step;}
9)   For (int k=0; k<10000;k++)
10)  For (i=0; i<array_size;i++)
11)  For (j=0;j<array_size;j++)
12)  Kernel_operation (x[i] ,y[j]);
13)  Collect end_time;
```

Lines 1 and 13 are used to print the time at the beginning and at the end of the code, respectively. This will help us in knowing the execution time of the code. In line 4 we declare the variables needed to execute the kernel operation. Here we only declare two variables *x* and *y*. There could be more depending on the kernel. The variables are declared as an array where the size of the array (array_size) is equal to the range of the variables that are being studied. In lines 6–8, the declared variables are assigned numerical values. In line 12 the kernel operation is executed for all the possible combinations of the values in the studied range using the 'for' loops in lines 10 and 11. The 'for' loop in line 9 is necessary when physically measuring the voltage and current of the different components because without it, executing the code takes very short time which is not sufficient to collect power measurements. The larger the number of iterations, the more accurate the results but the more time is required for measuring.

The energy cost (refer to it as $E_{w\text{Kernel}}$) of executing the code shown in Figure 3 can be determined using either physical measurements or simulation techniques. For physical measurements, the average CPU power of running the code can be estimated by physically measuring the current and voltage of the major components on the board of the studied architecture. The CPU energy cost of executing the code in Figure 3 can be determined by multiplying the average power consumed by the execution time. If the special equipment required for physical measurements are not available, then simulation tools can be used to estimate the energy cost of executing the code in Figure 3.

Since our objective is to determine the energy cost of the kernel operation, we need to exclude the cost of the lines of the code other than the kernel operation. So we run the same code in Figure 3 after substituting line 12 by ';' and we estimate its energy (refer to it as $E_{wo\text{Kernel}}$). Now the energy cost of one kernel operation (refer to it as E_{Kernel}) can be easily calculated [as shown in equation (8)] by subtracting $E_{w\text{Kernel}}$ from $E_{wo\text{Kernel}}$ and dividing it by the number of times the kernel operation was executed, refer to it as N. N is dependent on the number of iterations of the 'for' loops in lines 9–11.

$$E_{\text{Kernel}} = \frac{E_{w\text{Kernel}} - E_{wo\text{Kernel}}}{N} \tag{8}$$

4 Experiments and results

In this section, we conduct three experiments to evaluate the proposed methodologies for energy analysis. In the first experiment, we perform simulation analysis to assess the energy impacts of the kernels derived in step 3 of Section 3.1 and we use the methodology described in Section 3.2 to estimate the energy of the kernels using Simplescalar (Burger and Austin, 1997) and WATTCH (Brooks et al., 2000) simulation tools. This experiment provides us with information about the kernels with highest energy impacts. In the second experiment, we consider alternatives to the kernels by way of approximation and study the impact of the approximation on the accuracy of BP. In the third experiment, we consider data manipulation as another possibility for energy reduction with BP neural network.

For the experiments below, we assume the NN has two input neurons, three hidden neurons, two output neurons and 10,000 iterations. We can calculate the number of times each kernel is executed for the NN under consideration by simply replacing *IN*, *HN*, *ON*, *N* in Table 4 by 2, 3, 2 and 10,000, respectively.

4.1 Energy impacts of individual kernels on overall algorithm energy

To simulate the execution frequency of each kernel, a separate counter is incremented in the code of Figure 2, each time a kernel is executed. Based on the 10,000 iterations, the number of times each kernel is executed for the NN under consideration is shown in Figure 4. The simulation result of Figure 4 is consistent with the asymptotic analysis conducted in step 3 of Section 3, which showed that the highest numbers of kernels executed are multiplications followed by additions. To estimate the energy cost of BP kernels, we ran the codes of our methodology that is explained in Section 4 on Simplescalar and WATTCH simulation tools. Simplescalar simulates the execution of the code on the RISC architecture and WATTCH is used to give relative energy numbers that determine the energy cost of the different units of the architecture when this code is executed. In our experiments, we used the default configurations of Simplescalar and WATTCH. Details of these configurations are found in Burger and Austin (1997). For the BP kernel, the relative CPU energy costs that were obtained by simulation after using the methodology are shown in Figure 5. The y-axis in Figure 5 represents relative numbers obtained by WATTCH for the CPU energy of executing one kernel operation. We notice in Figure 5 that the exponential kernel is the one with the highest CPU energy cost. The overall energy was calculated by multiplying the number of times each kernel is executed by the corresponding cost of each kernel. The overall energy of the algorithm based on Figure 5 and Table 4 was calculated as shown in equation (9).

$$\begin{aligned}\text{overall energy} &= 152.7 \times 490 + 1{,}41.8 \times 350 + 141.1 \times 50 \\ &\quad + 208.4 \times 30 + 1{,}612.6 \times 30 = 186{,}138 \ (\text{in thousands})\end{aligned} \tag{9}$$

Based on this calculation, we can determine the contribution of each kernel to the overall energy as shown in Figure 6 where we see that although the exponential function is executed few times (as shown in Figure 4), it has a high contribution to the overall energy due to its high relative energy cost. It is also clear from Figure 6, that the multiplication kernel has the highest impact followed by addition and exponential. We also note that the

energy contribution of the exponential kernel is higher than that of the division kernel despite the fact that there are more division computations than there are exponentials.

Figure 4 Number of times each kernel is executed for 10,000 iterations (see online version for colours)

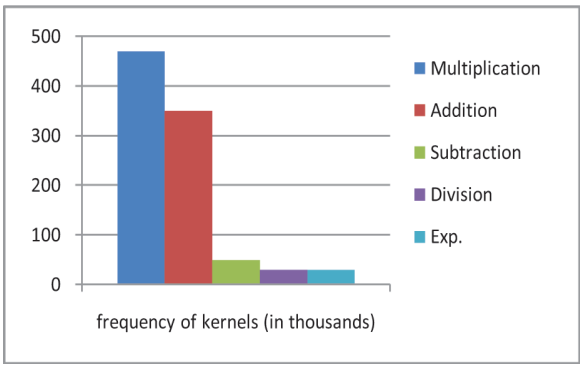


Figure 5 WATTCH results for the CPU energy cost of BP kernels on RISC architecture (see online version for colours)

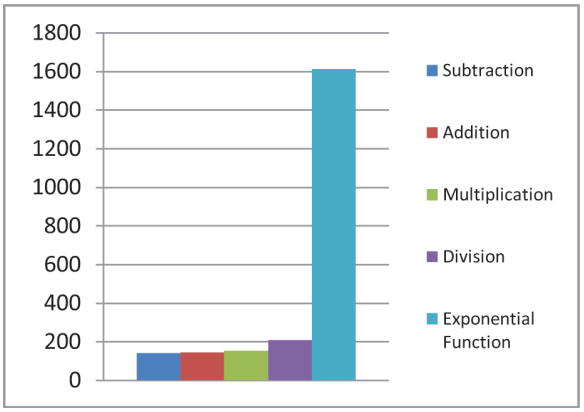
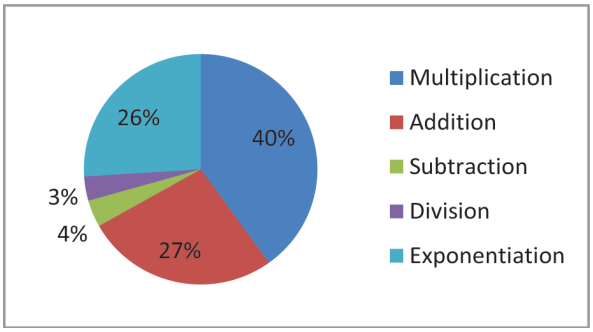
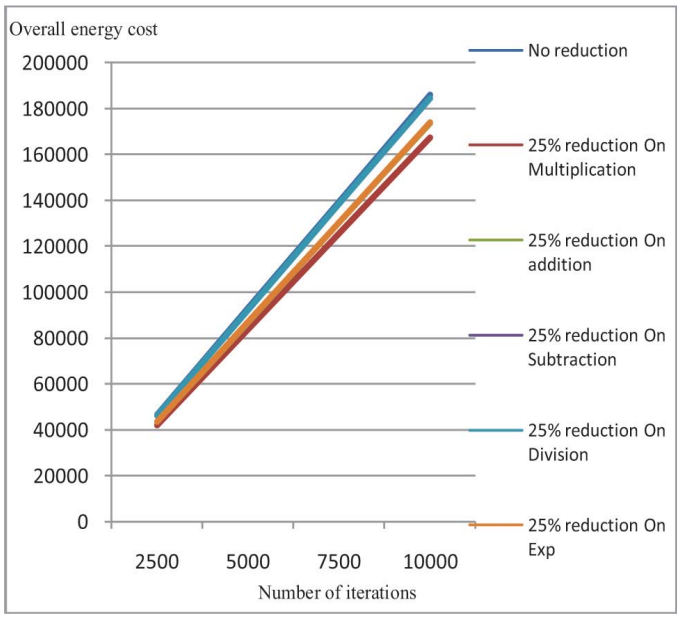


Figure 6 Contribution of each kernel to the overall cost (see online version for colours)



This simulation supports the validity of the proposed methodology to determine kernels with high energy impact, which would need to be further considered for energy reduction. To further confirm these conclusions, we study the savings of the overall energy that is achieved by reducing each kernel's cost by 25%. This helps in reflecting which kernels should be targeted for maximising the reduction of energy. Figure 7 shows the overall energy cost of the algorithm as the number of iterations increases after reducing each kernel's cost by 25%. Results indicate that the largest overall energy reduction is obtained by reducing the cost of multiplication followed by reducing the cost of the exponential function.

Figure 7 The impact of reducing different kernels by 25% on the overall algorithm for different iterations (see online version for colours)



We can also see that reducing the cost of the other kernels (subtraction, division and addition) has an impact on energy reduction, though not as large as multiplication or exponential.

In addition to the high cost of the exponential function, it is also one of the most time-consuming parts of the algorithm. We show next how to improve the performance of the algorithm by using lookup tables instead of calculating the value of the exponential function and show how this improvement also leads to energy saving.

4.2 Using alternative lower energy kernels

In this experiment, we examine alternative options to the high energy consuming exponential operation by using approximation techniques (Yamamoto, 2004). The basic idea behind our optimisation technique is to calculate the exponential function of the numbers in the range 0–1,000 with a step of 1 and store them in a LUT. Now in each iteration of the training phase (usually the number of iterations of BP algorithm is larger

than 5,000), instead of calculating the exponential function of the variables, we fetch the stored result from LUT that is maintained in the cache.

In order to evaluate our proposed method, we need to determine the energy cost of calculating the exponential function of the numbers in the range (0–1,000) and then compare the energy cost of fetching an element from LUT with the energy cost of calculating the exponential function. The energy cost of calculating the exponential function was already determined in Figure 5. So we only need to calculate the energy cost of fetching an element from LUT and compare it to the cost in Figure 5.

But before estimating the cost of a LUT fetch, we need to fill the table with the exponential of the numbers in the range (0–1,000). Hence we need to find the energy cost of populating the LUT. This energy cost of LUT population is only incurred once (refer to it as E_{LUT_initial}) and it can be calculated based on the energy cost of (exp) kernel that was determined using simulation tools as shown in equation (10):

$$E_{LUT\_\text{initial}} = 1,000 \times E_{\exp} = 1,000 \times 1,612.62 \tag{10}$$

where $E_{\exp}$ is the CPU cost of one exponential function operation (Figure 5).

To estimate the energy cost of fetching an element from LUT we measure the energy cost of executing the pseudo code shown in Figure 8 (refer to it as E1). Then, we estimate the energy cost of executing the code without line 8 (refer to it as E2). The energy cost of fetching an element from LUT can be determined by subtracting E2 from E1 and then dividing the obtained result by the number of fetches that were executed in the code.

Figure 8 Code to determine the energy cost of fetching elements from LUT

```
1)  Print time;
2)  //Calculate the exponential function of the numbers in the
    range [0,1000]
3)  For(int i=0; i<1000;i++)
4)       LUT[i]=exp(i);
5)  //Choose a random number and fetch the stored
    exponential function result from LUT
6)  For(i=0; i<1000000;i++)
7)     { pick a random number k;
8)       Fetch the kth element in LUT; }
9)  Printtime;
```

Our simulation results showed that the CPU energy cost of fetching an element from LUT is equal to (88.92). This cost is much smaller than the CPU energy cost for calculating the exponential function which is (1,612.62). In fact, fetching an element from LUT has 94.49% less CPU cost than calculating the exponential function. We should note that using LUT is considered an approximation technique since we are only calculating the exponential function for the variables in the range (0–1,000) with a step of one. Therefore, if we want to fetch the exponential function of (1.3) we will fetch the exponential function of (1) since (1.3) is not stored in the table. However this approximation causes a great saving of energy.

To measure the accuracy of our approximation technique, we calculate the value of exp(s), s being the input of the hidden neuron in equation (1), in two different ways: using the normal exponential function [referred to as expNormal(s)] and using the suggested approximation technique that utilises lookup tables [referred to as expApp(s)]. We then calculate the error which is the difference between these two values, i.e., error(s) = expNormal(s) − expApp(s) and then divide it by expNormal(s) to get a measure of error that is proportional to the normal exponential function, i.e.,

$$relative\_error(s) = \frac{|expNormal(s) - expApp(s)|}{|expNormal(s)|}.$$

Since there exist three inputs (s_1, s_2 and s_3) for the three hidden neurons, we calculate the three corresponding relative errors: *relative_error*(s_1), *relative_error*(s_2) and *relative_error*(s_3) and repeat the calculations for 10,000 iterations.

The average of all relative errors was calculated to be 0.2564, which shows that *error*(s) is very small compared to *expNormal*(s).

Figures 9 and 10 show a comparison between NN output using two lookup tables and using the normal exponential function. In Figure 9, (a) represents the first desired output of NN, (b) represents the first output of NN using normal exponentiation function and (c) represents the first output of NN using LUT. The same comparison is shown in Figure 10 for the second output of NN. Visually, we can see that the output of NN with the LUT approximation technique is very close to the output of NN with no approximation.

Figure 9 (a) 1st desired output, (b) 1st NN output using exp and (c) 1st NN output using lookup table approximation (see online version for colours)

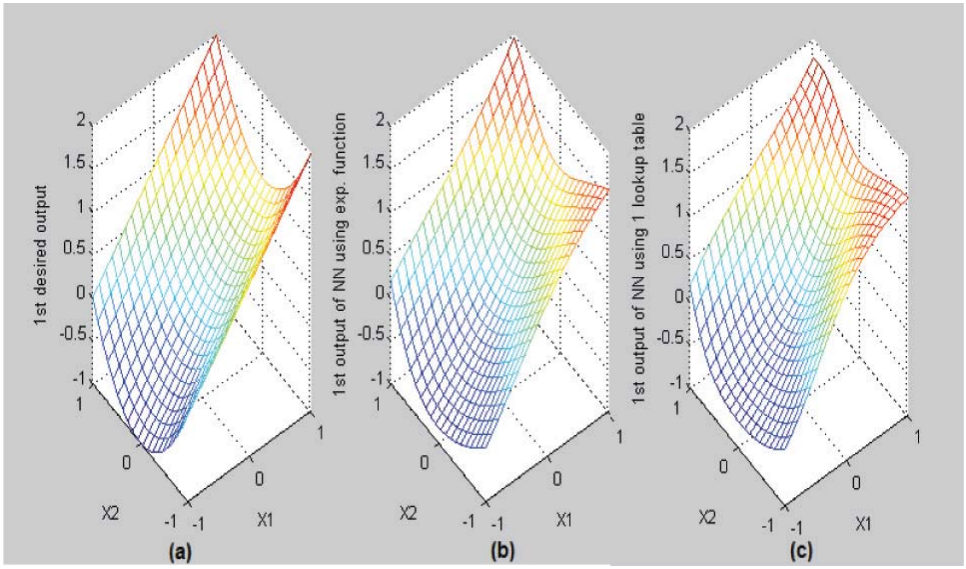
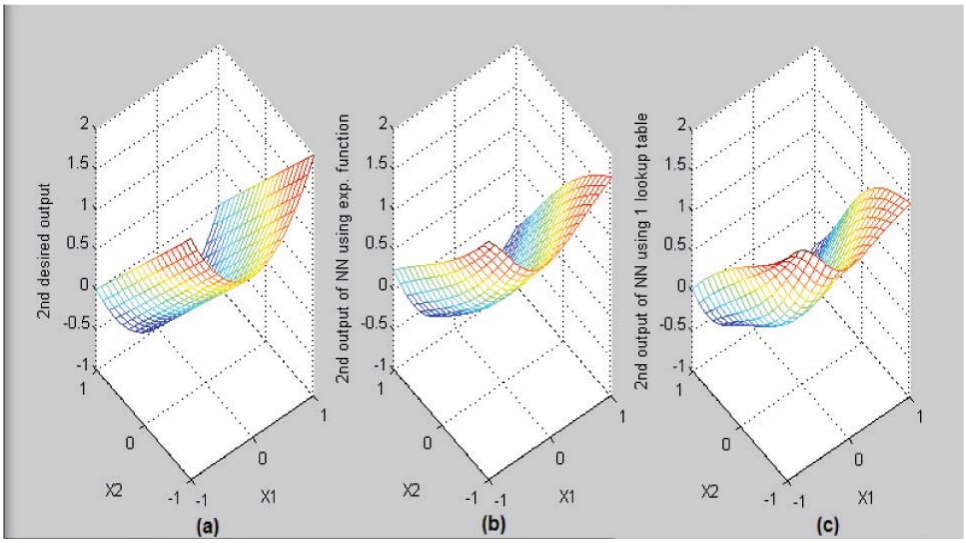


Figure 10 (a) 2nd desired output, (b) 2nd NN output using exp and (c) 2nd NN output using lookup tables approximation (see online version for colours)



4.3 Examining other opportunities of energy reduction specific to the algorithm under consideration

In this experiment, we examine ways that are specific to the algorithm beyond the general examination of kernels. In particular, we look at preprocessing the data to simplify algorithm computations and examine the energy impact.

As stated in Haykin (1998), the input values of the NN training phase can be normalised to the range of (0, 1) for faster convergence of the NN algorithm. To examine the impact of the normalisation on energy efficiency, we examine two scenarios.

In the first scenario, the training set values are not normalised, while in the second scenario, they are normalised to the range (0, 1). In both cases, the learning phase is continued until the mean squared error (MSE) is less than a predefined threshold, usually between 0.1 and 1. Since the normalised training set requires fewer iterations, it consumes less computation energy. But we should take into account the energy cost of the normalisation process when we compare the energy savings between the two scenarios.

Min-max Normalisation was used to normalise the training set into (0, 1) which is shown below:

$$v' = \frac{v - \min_a}{\max_a - \min_a}\left(\text{new\_max}_a - \text{new\_min}_a\right) + \text{new\_min}_a \tag{11}$$

where new_max = 1 and new_min = 0. So the normalisation process requires two operations: subtraction (whose relative cost is 141.1 based on our previous calculations in Figure 5) and division (whose relative cost is 208.4) for each attribute value in each training tuple and the energy cost for normalisation can be determined as shown below:

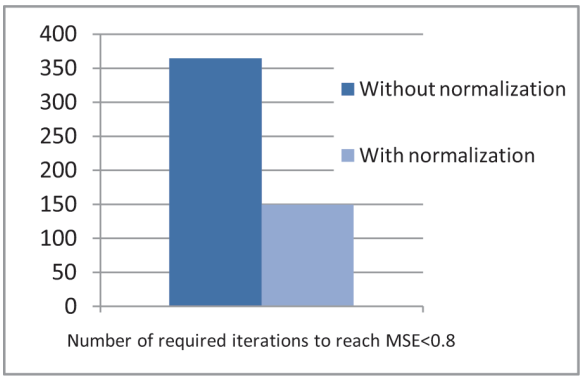
$$C_{\text{nor}} = (141.1 + 208.4) \times N_{\text{tuples}} \times N_{\text{att}} \tag{12}$$

where C_{nor} is the cost of normalisation, N_{tuples} is the number of tuples in the training set and N_{att} is the number of attributes in each tuple.

To demonstrate the savings in energy that are obtained by normalisation we ran an experiment on a training set that contained 400 training tuples, each tuple consisted of two attributes with values between 0 and 2.8. Our stopping criterion was when MSE reached a value less than 0.8.

Figure 11 shows the number of iterations that was required to reach the stopping criterion with and without normalisation. We can see that the larger the values of the input are, the more iterations were required to reach the MSE threshold.

Figure 11 Number of required iterations to reach MSE < 0.8 (see online version for colours)



To determine the execution frequency of each kernel, a separate counter is incremented in the code each time a kernel is executed until we reach the MSE threshold. Then we multiply the execution frequency of each kernel by the relative energy cost (obtained from our energy cost from Simplescalar and WATTCH simulation tools) and we obtain the training cost of the algorithm by adding the costs of all kernels. The cost of normalisation can be calculated from equation (12) by replacing N_{tuples}, N_{att} by 400 and 2, respectively. The overall energy for running the algorithm after normalisation equals the sum of the normalisation cost and the training cost.

Table 5 shows a cost comparison for running the algorithm with and without normalisation. It can be clearly seen that normalising the training set has led to reducing the energy cost. We conclude that normalising the training set before the learning phase makes the algorithm converge faster with fewer iterations and hence resulting savings in energy.

Table 5 A cost comparison with and without normalising the training set

Without normalisation		*With normalisation*	
Overall cost	*Cost of normalisation*	*Training cost*	*Overall cost*
6, 775, 400	279, 600	2, 773, 500	3, 053, 100

5 Conclusions and future work

This paper has introduced a six-step design methodology for energy-aware algorithms, with particular emphasis on new kernel-based evaluation for energy optimisation. Our results show that further opportunities for energy savings are possible in the application layer. Our proposed methodology mainly included: kernel identification, energy-based asymptotic analysis, prioritising kernels based on their energy impact and proposing solution to reduce the cost of kernels that contributed most to the overall energy. We also proposed a simple methodology that allows researchers to estimate the energy of algorithms' kernels using either simulation or physical measurements. The experimental studies demonstrated the effectiveness of our proposed methods. It also helped determine the kernels that contributed most to the overall energy. For the case of the back-propagation algorithm, we showed that exponential approximation and data preprocessing can also have significant impacts. In fact, our experiments showed that fetching an element from LUT consumes 94.49% less energy than calculating the exponential kernel. For future work, we plan to analyse other algorithms and estimate the energy cost of their kernels in addition to examining other opportunities for reducing the energy cost of these kernels. We also plan to pursue further validation by conducting real power measurements using specially instrumented boards for collection of current and voltage measurements in order to obtain more accurate results.

Acknowledgements

This work was funded by Intel's Middle East Energy Efficiency Research (MER) program and the American University of Beirut (AUB) University Research Board (URB).

References

Abbas, N., Hajj, H. and Yassine, A. (2011) 'Optimal WiMAX frame packing for minimum energy consumption', *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, Istanblul, Turkey, pp.1321–1326.

Albers, S. (2010) 'Energy-efficient algorithms', *Communications of the ACM*, May, Vol. 53, No. 5, pp.86–96.

Brooks, D., Tiwari, V. and Martonosi, M. (2000) 'Wattch: a framework for architectural-level power analysis and optimizations', *ACM SIGARCH Computer Architecture News*, Vol. 28, p.94.

Burger, D. and Austin, T. (1997) 'The simplescalar tool set, version 2.0', *ACM SIGARCH Computer Architecture News*, Vol. 25, No. 3, pp.13–25.

Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y. and Olukotun, K. (2006) 'Map-reduce for machine learning on multicore', *Neural Information Processing Systems*, Vancouver, B.C., Canada, pp.281–288.

Cooper, K.D., Subramanian, D. and Torczon, L. (2001) 'Adaptive optimizing compilers for the 21st century', *Proceedings of the 2001 Symposium of the Los Alamos Computer Science Institute*, October, *The Journal of Supercomputing*, Vol. 23, No. 1, pp.7–22.

Dabbagh, M., Hajj, H., Chehab, A., El-Hajj, W., Kayssi, A. and Mansour, M. (2011) 'A design methodology for energy aware neural networks', *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, Istanbul, Turkey, pp.1333–1340.

Darwish, T.H. and Chabukswar, R. (2009) *Intel Hardware Accelerated High Definition Video Playback Power Analysis*, Intel Software & Service Group.

Daud, S., Ahmad, R.B. and Murhty, N.S. (2008) 'The effects of compiler optimizations on embedded system power consumption', *Electronic Design, 2008. ICED 2008. International Conference*, Penang, Malaysia, pp.1–6.

Desai, N.P. (2009) 'A novel technique for orchestration of compiler optimization functions using branch and bound strategy', *Advance Computing Conference, IACC 2009*, Patiala, India, pp.467–472.

Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G. and O'Boyle, M.F.P. (2009) 'Portable compiler optimisation across embedded programs and microarchitectures using machine learning', *Microarchitecture, MICRO-42, 42nd Annual IEEE/ACM International Symposium*, New York, USA, pp.78–88.

Dutta, P.K. and Culler, D.E. (2005) 'System software techniques for low-power operation in wireless sensor networks', *ICCAD '05 Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, November, California, USA, pp.925–932, 6–10.

Ge, R., Feng, X., Song, S.,Chang, H., Li, D. and Cameron, K.W. (2010) 'PowerPack: energy profiling and analysis of high-performance systems and applications', *Parallel and Distributed Systems, IEEE Transaction*, Vol. 21, pp.658–671.

Han, J. and Kamber, M. (2006) *Data Mining: Concepts and Techniques*, 2nd ed., Morgan Kaufmann, January, ISBN 13: 978-1-55860-901-3, ISBN 10: 1-55860-901-6.

Haykin, S. (1998) *Neural Networks: A Comprehensive Foundation*, 2nd ed., Prentice Hall, July, ISBN-10: 0132733501, ISBN-13:9780132733502.

Hu, C., Jimenez, D.A. and Kremer, U. (2005) 'Toward an evaluation infrastructure for power and energy optimizations', *19th IEEE International Symposium on Parallel and Distributed Processing*. Colorado, USA.

Hung, S., Tu, C., Lin, H. and Chen, C. (2009) 'An automatic compiler optimizations selection framework for embedded applications', *Embedded Software and Systems, ICESS '09, International Conference*, Zhejiang, China, pp.381–387.

John, B.P., Agrawal, A., Steigerwald, B. and John, E.B. (2010) 'Impact of operating system behaviour on battery life', Presented at *J. Low Power Electronics*, April, Vol. 6, No. 1, pp.10–17(8).

Kaxiras, S. and Martonosi, M. (2008) 'Computer architecture techniques for power-efficiency', Morgan and Claypool, p.220.

Kogge, P. (2011) 'The tops in flops', *IEEE Spectrum Magazine*, February, p.68.

Larson, P. (2008) *Energy-Efficient Software Guidelines*, Intel Software Solution Group.

Lu, Y. and De Micheli, G. (2001) 'Comparing system level power management policies', *Design & Test of Computers*, IEEE, Vol. 18, pp.10–19.

Lu, Y. and De Micheli, G. (2001) 'Comparing system-level power management policies', *IEEE Design & Test*, March, Vol. 18, No. 2, pp.10–19.

Malik, A.M. (2010) 'Spatial based feature generation for machine learning based optimization compilation', *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference*, Washington DC, USA, pp.925–930.

Marcu, M., Tudor, D., Moldovan, H., Fuicu, S. and Popa, M. (2009) 'Energy characterization of mobile devices and applications using power–thermal benchmarks', *Microelectron. Journal*, Vol. 40, No. 7, pp.1141–1153.

Negrevergne, B., Termier, A., Mé haut, J. and Uno, T. (2010) 'Discovering closed frequent itemsets on multicore: parallelizing computations and optimizing memory accesses', *High Performance Computing and Simulation (HPCS)*, 28 June–2 July, pp.521–528, doi: 10.1109/HPCS.2010.5547082.

Pawaskar, S. and Ali, H.H. (2010) 'A dynamic energy-aware model for scheduling computationally intensive bioinformatics applications', *High Performance Computing and Simulation (HPCS)*, 28 June–2 July, Caen, France, pp.216–223.

Pettey, C. (2007) 'Gartner estimates ICT industry accounts for 2 percent of global CO2 emission', Gartner Press Release, 26 April, available at http://www.gartner.com/it/page.jsp?id=503867.

Schall, D., Hudlet, V. and Harder, T. (2010) 'Enhancing energy efficiency of database applications using SSDs', *Conference on Computer Science and Software Engineering (C3S2E '10)*, ACM, New York, NY, USA, 1–9.

Sinevriotis, G. and Stouraitis, T. (2002) 'A novel list-scheduling algorithm for the low-energy program execution', *Circuits and Systems, ISCAS 2002, IEEE International Symposium*, Arizona, USA, Vol. 4, pp.IV-97–IV-100.

Steigerwald, B., Chabukswar, R., Krishnan, K. and Vega, J.D. (2008) *Creating Energy – Efficient Software*, Intel White Paper.

Unnikrishnan, P., Chen, G., Kandemir, M. and Mudgett, D.R. (2002) 'Dynamic compilation for energy adaptation', *Computer Aided Design, ICCAD 2002, IEEE/ACM International Conference*, California, USA, pp.158–163.

Wiratunga, S. and Gebotys, C. (2000) 'Methodology for minimizing power with DSP code', *Electrical and Computer Engineering*, Vol. 1, pp.293–296.

Yamamoto, A. (2004) 'Computational efficiencies of approximated exponential functions for transport calculations of the characteristics method', *Annals of Nuclear Energy*, June, Vol. 31, No. 9, pp.1027–1037.