FPGA Fastfood - A High Speed Systolic Implementation of a Large Scale Online Kernel Method

ABSTRACT

In this paper, we describe a systolic Field Programmable Gate Array (FPGA) implementation of the Fastfood algorithm that is optimised to run at a high frequency. The Fastfood algorithm supports online learning for large scale kernel methods. Empirical results show that 500 MHz clock rates can be sustained for an architecture that can solve problems with input dimensions that are 10^3 times larger than previously reported. Unlike many recent deep learning publications, this design implements both training and prediction. This enables the use of kernel methods in applications requiring a rare combination of capacity, adaption and speed.

1 INTRODUCTION

Kernel methods are a popular class of machine learning algorithm capable of solving many problems, ranging from classification and regression to novelty detection and feature extraction. However, they are often limited to small datasets because their memory and computation requirements scale linearly with the number of input examples. To address this problem, Le et al. [8] proposed an efficient algorithm, called Fastfood, that builds an approximation to the kernel using combinations of random diagonal matrices and Hadamard transforms. This is advantageous because diagonal matrices require little storage and only involve elementwise multiplications, and the Hadamard transform can be computed in log linear time via the Fast Walsh Hadamard Transform (FWHT). Fastfood reduces $O(nd)$ storage and computation requirements to $O(n)$ and $O(n \log_2 d)$ respectively, where d is the length of the input vectors and n represents the number of basis functions used to preserve the statistical properties of the kernel function. To summarise, Fastfood only suffers minimal degradation in prediction accuracy and can solve problems which were previously intractable using exact kernel methods.

We describe the first known implementation of Fastfood using Field Programmable Gate Arrays (FPGAs). Our design, given in Section 3, is a systolic array architecture that creates local connections by grouping processing elements into designated blocks. This gives us the flexibility of compiling different configurations of Fastfood, and achieve a 500 MHz operating frequency. In fact, simply preserving module partitions during synthesis is enough to sustain high clock rates with our design. In this article, we highlight that Fastfood is particularly suited for a hardware implementation because: 1.) the computation requires simple arithmetic operations involving mainly addition and subtraction, 2) memory requirements are minimal allowing large parameter spaces to be stored in embedded memory, 3.) the algorithm can be translated into a datapath that has a regular structure, is highly parallelisable, and benefits from high-speed local interconnections, and 4.) the butterfly structure for computing the FWHT has an efficient hardware implementation.

Fastfood implements a form of model compression for kernel methods, however, unlike related deep learning algorithms that take advantage of reduced precision and sparsity [3, 6], Fastfood also works during the learning phase. In addition, kernel methods are attractive in comparison to Neural Networks because they allow domain knowledge to be imparted in a statistical manner. Neural Networks have powerful generalisation properties because they are free to learn any data representation, but this can obfuscate the training process and yield results which are often less interpretable. The key contributions of this paper are:

- The first FPGA implementation of a large-scale online kernel method. Our design can solve problems with an input dimensionality up to 3 orders of magnitude larger than previous FPGA implementations of kernel methods, and achieves 245× speed up over a single-core Central Processing Unit (CPU).
- A novel hierarchical systolic architecture for sustaining high clock rates on FPGAs. Blocks of processing elements constrain the majority of computation to small physical spaces, and allow resource reuse. In particular, the design efficiently implements the computational bottleneck (FWHT) using the same processing elements that compute the rest of the Fastfood algorithm.

2 BACKGROUND

2.1 Machine Learning Regression Using Kernel Methods

Let $x_i, i = 1...m$ represent m input vectors of a given dimensionality d ($x_i \in \mathbb{R}^d$), and let y_i represent the corresponding desired output values for each input vector ($y_i \in \mathbb{R}$). In standard machine learning regression problems, the goal is to find a function $f(x)$ that results in the minimal prediction error (i.e. $\min \sum_{i=1}^{m} (y_i - f(x_i))^2$). For non-linear problems, the input must first be transformed into a high dimensional space where patterns in x_i are linearly separable. The *kernel trick* does this implicitly, and thus, it is regularly employed to inexpensively evaluate $f(x)$, as described by (1).

$$f(x_i) = \sum_{j=1}^{N} \alpha_j \kappa(x_i, v_j) \tag{1}$$

Here, $\kappa(x_i, v_j)$ is the kernel function, and α_i are parameters to be optimised. The only caveat is that a length N subset of the input data must be stored in memory, also known as the *dictionary* or *support vectors* and denoted by v. The machine learning algorithm must update both the dictionary and parameters α_j to minimise the predictive error. On small datasets (less than 10^4 examples), (1) can be computed efficiently. However, this is not possible on large datasets because N tends to grow linearly with the number of input examples, m, [2]. To address this problem, Rahimi and Recht [11] proposed Random Kitchen Sinks (RKS) for shift-invariant kernels ($\kappa(x, x') = \kappa(x - x', 0)$). The main idea is to create a function, $z(x) = \frac{1}{\sqrt{n}} cos(Wx_i)$, such that the inner products $\langle z(x), z(x') \rangle$ are

approximately equal to the high-dimensional features extracted from $k(x, x')$. Here, n is the number of basis functions used to approximate N dictionary elements, and W is a $n \times d$ matrix randomly sampled from the Fourier transform of $k(x, x')$. Importantly, n is fixed and means that Random Kitchen Sinks can be trained independent of the dataset size, making them much faster on problems with many inputs.

2.2 Fastfood

The Fastfood algorithm was introduced by Le et al. [8] to reduce the computational complexity of Random Kitchen Sinks. The main idea is that a random projection, given by Wx_i, can be approximated with a projection Vx_i, as described by (2), which requires much fewer operations to compute. [1]

$$Vx_i = [V_1 x_i, ..., V_h x_i], \text{ where } V_q x_i = SHGPHBx_i \tag{2}$$

We first break the computation down into $h = \lceil n/d \rceil$ separate stages, each working with d basis functions. By working from right to left, we can complete each intermediate computation of $V_q x_i$ by a sequence of matrix-vector operations, without storing the matrix V_q. Importantly, the matrix-vector operations are designed to involve minimal computation. Firstly, B, G and S are $d \times d$ *diagonal* matrices, where B_{ii} and G_{ii} are drawn i.i.d. from $\{-1, 1\}$ and $N(0, 1)$ distributions respectively, whilst S_{ii} depends on the choice of kernel function. In this work we assume a Gaussian RBF kernel, meaning S_{ii} is drawn from a chi-squared distribution [8]. Secondly, $P \in \{0, 1\}^{d \times d}$ is a permutation matrix which can be efficiently implemented in hardware. Finally, $H \in \{-1, 1\}^{d \times d}$ is a Hadamard matrix, as defined by (3). Matrix-vector multiplication with a Hadamard matrix can be done efficiently using the Fast Walsh Hadamard Transform (FWHT). For example, using the radix-2 FWHT algorithm, this reduces the number of operations from $O(d^2)$ to $O(d \log_2 d)$. The Hadamard matrix in (3) requires the input dimension, d, to be a power of 2 ($d = 2^l$, where $l \in \mathbb{N}$). To meet this condition, the input vectors can be padded with zeros.

$$H = H_d = \begin{bmatrix} H_{d/2} & H_{d/2} \\ H_{d/2} & -H_{d/2} \end{bmatrix} \text{ where } H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3}$$

The prediction is then formed by passing $Vx_i \in \mathbb{R}^n$ through a sinusoidal function, $\psi(x_i) = \frac{1}{\sqrt{n}} \cos(Vx_i)$, and taking a weighted sum as shown by (4).

$$f(x_i) = \sum_{j=1}^{n} \alpha_j \psi(x_i) \tag{4}$$

In this work, we train $f(x)$ for regression by solving the least squares problem $\min_\alpha \|y - \alpha^T \psi(x)\|_2^2$, where $y \in \mathbb{R}$ is the expected output. More specifically, we apply an online stochastic gradient descent algorithm, which incrementally updates α for each new (x_i, y_i), as given by (5). Alternatively, a batch-based training method could be used. However, this requires prior access to a large subset of the input data which is not available in real-time applications.

$$\alpha_{t+1} = \alpha_t + \eta[y_t - \alpha_t^T \psi(x_t)]\psi(x_t) \tag{5}$$

[1] For brevity, we limit the review to the key computation steps. Refer to [8] for a complete analysis, including derivations, proofs of convergence, and error bounds.

As with Random Kitchen Sinks, when $n = 16,384$, Fastfood achieves an accuracy on the CIFAR-10 dataset [7] which is among the top two for shift-invariant kernel representations [8].

2.3 Kernel Methods on FPGA

Efficient FPGA implementations of kernel methods have previously been studied and several architectures for performing simultaneous prediction and training have been reported. Fraser et al. [5] describe a floating point implementation of the KNLMS algorithm. They achieve very high efficiency using a fully-pipelined architecture and time-multiplexing independent parameter sets. This removes the data dependency in the update equation (similar to (5)) significantly improving overall throughput. Multiple parameter sets and deep pipelining increase the latency on a per input basis. Tridgell et al. [12] address this issue using a technique called "Braiding", which resolves data dependencies in pipelined architectures. Their design (in fixed point) operates at well over 10 Gb/s and achieves a latency around 10 cycles for $d = 8$ and a fixed $N = 200$, this coming at the cost of frequency and DSP resources. Neither explore re-utilisation of resources. Pang et al. [10] describe a micro-coded vector processor for the SW-KRLS algorithm which time-multiplexes resources to improve scalability. The works reviewed target a specific part of the design space, however, none can scale the input dimensionality, d, and dictionary size, N, to support the type of big-data applications dominating the machine learning community today. Our implementation of Fastfood fills this void.

3 ARCHITECTURE AND DESIGN

3.1 High-Level Description

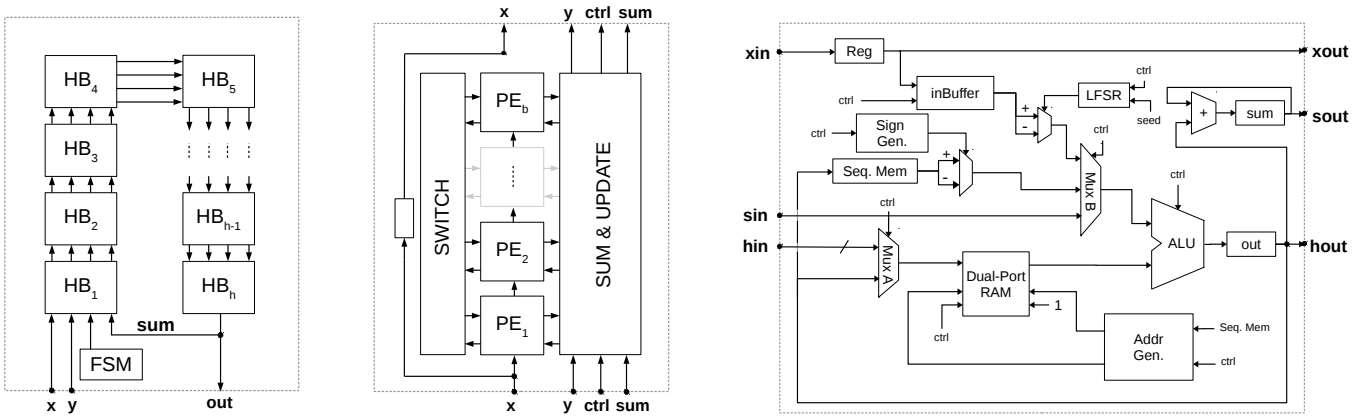
From Equation (2), it is evident that $V_q x_i$ can be processed independently. Our architecture separates and localises their computation using blocks of Processing Elements (PEs), called Hadamard Blocks (HBs). A hierarchical diagram of the Fastfood processor is shown in Figure 1, and the following equation shows how the computation of $f(x)$ (4) is unrolled into (6).

$$f(x) = \sum_{j=1}^{n} \alpha_j \psi(x) = \sum^{h} \sum^{b} \sum_{j=1}^{k} \alpha_j \psi(x) \tag{6}$$

Here, n is the number of basis functions or expansion dimensions, k is the basis functions per PE, b is the PEs per HB, and h is the number of HBs. The other constraints are the input dimensionality, d, and total number of PEs, p.

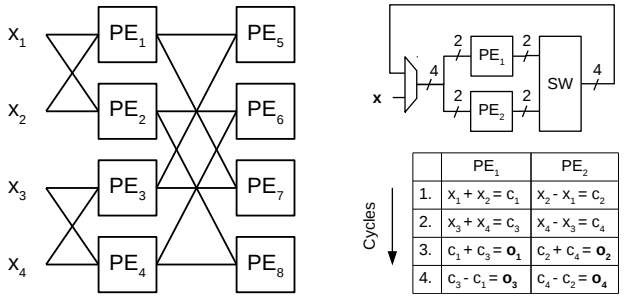
We adopted this architecture because 1) massive scalability can be achieved when PE compute resources are reused, 2) PEs with mostly local connections are more amenable to high-frequency implementations, and 3) blocks of PEs create the local connectivities needed to implement a fused and distributed Fast Walsh Hadamard Transform (FWHT). The last point gives our architecture some unique characteristics in the trade-off between resources, parallelism and frequency, and is discussed more in Section 3.3.

Parameters p, b and k should be chosen carefully, and depend on the FPGA device, task, and performance requirements: 1) p controls the parallelism and latency of the design; 2) $k = n/p$ is a portion of the algorithm distributed evenly across the PEs. k controls the computable size of n and scales the size of each PE; 3) $b = d/k$ is the



(a) Top-Level (b) Hadamard Block (HB) (c) Processing Element (PE)

Figure 1: A hierarchical block diagram of the Fastfood processor



(a) no iterative reuse, width=4 (b) full iterative reuse, width=2

Figure 2: Two systolic array configurations for a 4-point FWHT, based on Milder et al. [9].

number of PEs allocated to each HB. b controls the locality of connections between PEs; and in combination with k, also determines the computable size of d.

3.2 Top-Level Module

The top level diagram of our architecture is illustrated in Figure 1a (ignoring PCIe and memory bus interfaces). It mainly consists of an h-length array of HBs connected in a ring topology. Each HB accepts control logic from a finite state machine (FSM) and a stream of (x_i, y_i) pairs as input data. The Fastfood processor is an iterative architecture which does not have overlapping compute stages. This means that the inputs arrive with an initiation interval (II) equal to the number of cycles taken to compute a given sized input vector. Each HB computes a partial result for Fastfood, and an output is produced by summing each partial result as per the outer loop of (6). The HBs are connected in a ring array for two main reasons: 1.) to minimise routing between HBs for computing the sum, and 2.) to efficiently broadcast the input and FSM control logic across a large area of the chip.

3.3 Hadamard Block

Each HB contains an array of b PEs, a switch, and a module for computing the HB sum and partial update. This is shown in Figure 1b. We created HBs with the aim of developing local structure between PEs for implementing the FWHT at high clock rates. A special emphasis is placed on the FWHT because its parallel implementation has data dependencies which require communication between multiple PEs. Below, we discuss design considerations for the FWHT and *Sum & Update* unit which maximise the frequency and scalability of Fastfood.

The standard architecture for a radix-2 4-point FWHT is the fully pipelined version given in Figure 2a. It is constructed by cascading $\log_2 d$ stages of d PEs and connecting them in a butterfly structure to directly implement the required switching behaviour. This design executes the FWHT in a minimum number of cycles, but requires large amounts of additional hardware that are not reused. Given that the rest of Fastfood can be computed using a linear systolic array, we chose to implement the d-point FWHT by reusing one stage of b PEs over multiple cycles, like Figure 2b. This design has a compact and regular structure, which makes it well suited to processing large amounts of data at a high frequency. The only overhead is the switching network which is used to send intermediate results between $\log_2 b$ PEs. We implemented the switch using a multiplexer in each PE, and connected the PEs directly. For increasing b, the switch size also increases. We experimented with $b = \{4, 8, 16, 32\}$, above which we observed a degradation in frequency due to routing congestion.

The *Sum & Update* unit routes the FSM control logic and also includes some logic to implement several instructions not allocated to the PEs. Figure 3 gives a dataflow diagram of the operations involved. We use $\log_2 b$ adders to compute the middle loop of (6). Each HB requires the results from every other HB to know the update (outer loop of (6)). These are passed around the ring array and accumulated here. The *Sum & Update* module then includes one subtract and multiply unit for computing part of update equation (5). The result is written back to each PE where the weights are ultimately updated. This creates a small resource inefficiency and

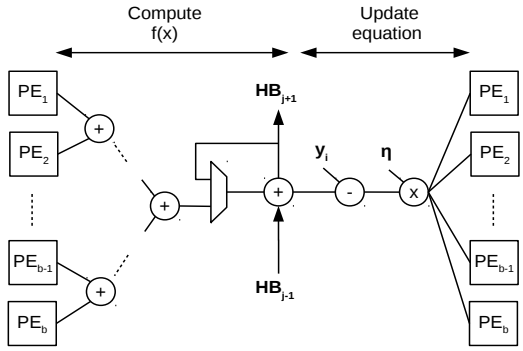


Figure 3: Dataflow diagram of the HB sum and update module (data moves from left to right)

Instr.	ALU	Active Cycles	Instr.	Active Cycles
Proc. Elem. (PE)			Had. Block (HB)	
Mul B	N	k	Adder-tree	$log_2 b$
Mul	Y	$3k$	Add	h
Perm.	N	k	Sub $(y - f(x))$	1
Add	Y	$2k \log_2 d + k$	Mul (η)	1
Cosine	N	k		
Mul-Add	Y	$k + 1$		

Table 1: Fastfood instruction count for resources constrained in each PE and HB

means the PEs are idle for a short period of time. However, it only equates to an extra 1×DSP and between 2-5% in LUTs and FFs, while the PEs are idle for only 1% of total processing time.

3.4 Processing Element

The basic structure of each PE is given in Figure 1c. It consists of one Arithmetic Logic Unit (ALU), two scalar operands, and a control mechanism for reading and writing to memory. Computation can only begin once a full input vector has been loaded into PE memory and the previous computation is finished. The *inBuffer* shift register is used to buffer the input and allow the loading and computation stages to be overlapped. Below, we discuss other important elements of the Fastfood PE:

Table 1 summarises the compute requirements of each PE as a list of instructions. As per the table, not all the functionality is implemented with resources confined to the ALU. For instance, the first Fastfood operation is a binary multiplication involving *inBuffer* and a random variable $B \in \{-1, 1\}$ (in Eq. 2). We implemented this as a sign change using 1-bit from a 16-bit Linear Feedback Shift Register (LFSR). The Mul and Add instructions are implemented using 1×DSP slice and 1×adder in the ALU, and for Mul-Add we use an additional adder for the accumulation stage (i.e. *sum*). The majority of Fastfood compute time is spent iteratively reusing the ALU adder. In fact, approximately 70% of total cycles are dedicated to Adds. The cosine function is implemented using a 256-point look-up table stored in a shared *Dual-Port RAM*. We read a result, i.e. $cos(x)$, using an address generated from x, where x is an intermediate result from the *Seq. Mem* shift register.

The FWHT involves $2k \log_2 d$ add and subtract operations per PE. We reuse the ALU adder and perform sign switching on the top operand using the *Sign Generator* block. The *hin* and *hout* I/O ports connect multiple PEs together, and *MuxA* switches between them. For local permutations (or switching), the *Address Generator* develops the memory access patterns required. The size of *MuxA* scales as $O(\log_2 b)$ which for $b = \{4, 8, 16, 32\}$ can be implemented efficiently with one layer of LUTs. The address and sign permutations are the same for each PE and can be generated using several counters and bit operations. This only contributes an additional 30 LUTS and 26 FFs for the *Address Generator* and *Sign Generator* modules combined. Given the small overhead, we have included these control units in every PE. This reduces the compute density of each PE, but removes a large fan-out problem which would likely require multiple stages of pipeline registers. Fastfood involves a data transformation from $\mathbb{R}^d \rightarrow \mathbb{R}^n$, where $n > d$. Our architecture distributes this computation across p PE's so that each PE works on $k = n/p$ dimensions, also known as basis functions. This requires k-length blocks of memory for intermediate results and Fastfood parameters G, S and α. We write intermediate results back to *Sequential Memory* every cycle, and load the Fastfood parameters to the *Dual-Port RAM* once at startup. For the FWHT switch, we read and write to a double buffer in *Dual-Port RAM* to avoid pipeline stalls. Therefore, including the *inBuffer* and 256-point cosine look-up table, each PE requires $(k + k + 3k + 2k + k + 256) \times bitwidth$ of memory. The sequential memories are either implemented using two LUT-based shift registers (i.e. SRL) or the *Seq. Mem* shift register is also mapped to a BRAM. In Section 4, we present results for both configurations. The choice depends on whether we prefer a LUT-constrained or BRAM-constrained design. The PEs are fully pipelined. There are 4 register stages for reading and writing to memory, between 3 and 6 on the ALU, 1 on the ALU operands, 1 on the output, and 1 on the *MuxB* inputs. This minimises the number of logic levels between registered signals and keeps the frequency high.

3.5 Scalability: I/O and Latency

The top-level module takes 1× 18-bit feature as input per cycle. The data is loaded through the array via a d-length shift-register, where d is the input dimensionality. Given that load and compute stages operate in parallel, one result can be retrieved every tc cycles, where tc is the total number of compute cycles given below:

$$tc = 8k + 2k \log_2 d + \log_2 b + 2h + 9 \quad (7)$$

The above equation is taken from Table 1 for prediction and training, except a few extra cycles are added because the HB Control unit is not fully pipelined. The latency is thus $tc + d$ and the I/O required is $18d/8$ bytes every $max(d, tc)$ cycles.

For large d and $d > tc$, the throughput is limited by loading of the input data. This equates to 18-bits per cycle, or 9Gb/s, assuming the memory bandwidth is not the bottleneck. This can be improved by broadcasting more of the input data in parallel. For fast designs, p should be large so a small k can be distributed to each PE.

	LUT Total (203k)	LUT Mem. (112k)	FFs (406k)	BRAM (1080)	DSP (1700)	Max. PEs (p)
FF-L						
k=64	323	98	524	1	1	610
k=128	398	170	531	1	1	494
k=256	547	314	540	2	1	355
FF-B						
k=64	282	70	590	2	1	540
k=128	327	107	603	2	1	540
k=256	405	180	615	3	1	360

Table 2: PE resource utilisation for LUT and BRAM constrained 18-bit designs on a Kintex XCU035

4 RESULTS AND EVALUATION

Our designs were written in Chisel HDL [1] and will be made available on our github repository. They were synthesised and implemented using Xilinx Vivado 2017.2, targeting a Kintex Ultrascale XCKU035-FBVA676-2-e FPGA. We present results for two designs: one which is LUT-constrained (FF-L) and the other BRAM-constrained (FF-B). In the FF-L design, both shift registers are mapped to LUT SRL primitives, and in the FF-B design, *Seq. Mem* is implemented with a BRAM.

4.1 Resource Utilisation

Table 2 shows resource utilisation of FF-L and FF-B for an 18-bit design. Area scales with k because of two k-length shift-registers, *inBuffer* and *Seq. Mem.* The last column shows the maximum number of PEs that can be placed on the targeted device. These numbers are based on 97% of total LUTs, 99.8% of memory configurable LUTs, and all available BRAMs. The remaining resources are attributed to the FSM and HB modules, which contribute an additional 141 and h×617 LUTs respectively. Our designs are restricted to three BRAMs per PE, one of which is only required for $k = 256$. Larger k can be supported, but this requires doubling memory resources since k scales as a power of two. In terms of parallelism, FF-L can support the most PEs for $k \leq 64$, FF-B is preferred for $k = 128$, and the designs are similar for $k = 256$.

4.2 Problem Size

The maximum problem size of the present design is compared with other online kernel methods in Table 5. For FF-B and FF-L, this translates to a 10^3 times increase in input dimensionality, and modelling capacity up to $90K$ basis functions. The massive modelling capcity allows our design to be used in problems with many input examples, m, because they approximate the kernel function for large dictionary sizes, N. Exact kernel methods such as KNLMS, KRLS and NORMA are intractable for datasets such as CIFAR-10 [7] because N is very large. In contrast, Fastfood with $n = 16,384$ basis functions achieves an accuracy in the top two for RBF kernel representations [8]. This suggests that our architecture with $n = 90K$ basis functions can solve problems even larger than CIFAR-10, which has $m = 50,000$ images and $d = 3,072$ dimensions.

		Designs achieving Freq=500 MHz			For max. resources in Table 2	
		LUT %	BRAM %	PEs (p)	PEs (p)	Freq. (MHz)
k=64	b=32	91.6	50.4	544	576	483
	b=64	67.9	35.6	384	576	416
k=128	b=32	92.0	41.5	448	480	487
	b=64	54.8	23.7	256	448	465
k=256	b=32	89.2	59.3	320	320	508
	b=64	53.3	35.6	192	320	476

Table 3: Clock frequency for an 18-bit FF-L design - *Seq. Mem* implemented using a LUT SRL primitive

		Designs achieving Freq=500 MHz			For max. resources in Table 2	
		LUT %	BRAM %	PEs (p)	PEs (p)	Freq. (MHz)
k=64	b=32	82.1	94.8	512	512	501
	b=64	66.0	71.1	384	512	465
k=128	b=32	88.3	88.9	480	512	487
	b=64	59.7	59.3	320	512	455
k=256	b=32	76.6	97.8	352	352	501
	b=64	43.6	53.3	192	320	468

Table 4: Clock frequency for an 18-bit FF-B design - *Seq. Mem* implemented using a BRAM

4.3 Clock Frequency

Table 3 and 4 show the effect of design configurations on the clock frequency. Importantly, both tables show that a high operating frequency can be achieved for designs very close to the resource constraints given in Table 2. For $b = 32$, clock rates at or above 500 MHz can be sustained right up to $p = 544$, $p = 448$ and $p = 320$ for FF-L, and $p = 512$, $p = 480$ and $p = 320$ for FF-B. This corresponds to utilisation rates between 89-98% of available resources. Furthermore, the frequency is only slightly reduced to 483-495 MHz for PEs which are even closer to the resource limit. Both tables highlight the degradation in clock frequency observed for any $b > 32$. This occurs because 1.) routing congestion increases between the PEs and *Sum & Update* unit, and 2.) the size of each HB doubles in size which de-localises connections between them. Additional pipeline registers could manage both these issues, but on large designs, this creates even more congestion between CLBs. Instead we chose an architecture, where $b <= 32$, which guarantees an operating frequency close to 500 MHz for varying problem sizes. All of our designs were synthesised with cross module LUT optimisation turned off. This preserves the local structure in our architecture, and results are up to 20% faster.

4.4 Speed-Up and Accuracy

Table 5 gives the speed-up of our FPGA implementation over a CPU. The execution time for the CPU version comes from the original

	(d,n,B)	Lat (cyc)	Fmax (MHz)	Exec (ns)	Tput (Gb/s)
[5] (V7)	(8,16,32)	207	314	3.18	80.4
[12] (V7)	(8,200,18)	10	127	7.87	18.3
[10] (SV)	(-,128,32)	4396	157	28000	-
[8] (CPU)	(1024,16.4K,32)	-	-	580000	0.06
Ours (V7)	(1024,16.4K,18)	1893	432	23700	7.77
Ours (KU)	(8192,90.1K,18)	16930	508	17200	8.57
Ours (V7)	(8192,98.3K,18)	16934	413	21167	6.97

Table 5: Comparison of our implementation (FF-B only) with other online kernel methods (B=bit width, Lat=latency, Tput=throughput, Exec=execution time, V7=Virtex 7, SV=Stratix V, and KU=Kintex Ultrascale)

Fastfood publication by Le et al. [8] and is only for prediction. The authors' code is written in C++ and uses the Spiral library for the FWHT. While details of the CPU are unclear from Le et al. [8], Spiral provides cache optimised C/C++ code which uses SSE vector instructions and up to four processor cores [4]. For a fair comparison, we chose a previous generation Virtex 7 FPGA as the target for our design. The problem has $d = 1,024$ and $n = 16,384$, therefore, our implementation consists of 16 HBs, each of which compute a 1024-pt FWHT. For an FF-L design, the device operates at 432 MHz and has capacity for $p = 985$ PEs. However, only $p = 512$ could be used because p scales exponentially on fixed problem sizes. For $b = 32$ and $k = 32$, the number of compute cycles for prediction is $tc = 869$ (from Eq. 7). Therefore, our design is I/O constrained (i.e. $tc < d$), and one result can only be obtained every d cycles. The final result is a 245× speed-up although only 52% of available resources are used. We compared the mean square error (MSE) of floating point and fixed point versions, on the Mackey Glass regression benchmark [13] using a model consisting of $n = 16,384$ and $d = 1,024$. Competitive learning accuracies were achieved for floating point (MSE= 0.07), using 24-bit (MSE= 0.09) and 18-bit (MSE= 0.12) fixed point, although convergence of 18-bit is slower on average. This is only a minor problem since the additional latency gets hidden over time during online training.

4.5 Summary

Our implementation of Fastfood occupies a unique part in the design space of online kernel methods. This is observed in Table 5 [2]. The reported Fastfood configurations achieve an excellent combination of problem capacity and throughput for an 18-bit implementation. This yields 3 orders of magnitude increase in input dimensionality, 8.57 Gb/s of throughput, and a large number of basis functions. The last point gives us the ability to approximate much larger dictionary sizes than can otherwise be supported in [5][12][10].

Compared with Random Kitchen Sinks that have $O(nd)$ memory complexity, our Fastfood design only requires storage for $3n$ parameters. This means that basis functions up to 1.54 GB (i.e. $n \times d \times 18/8$) can be approximated using only 0.58 MB, equating to a compression factor of 2655×. Future work will investigate how this result can be reinterpreted for DNNs, where state of the art compression factors are around 10× for full precision weights [6]. The main difference being that basis functions are learned in DNNs, whereas in Fastfood, they are randomly sampled from a distribution which closely approximates a kernel function.

[2] 1.) Input latency is included for Fastfood and KRLS [10] but not KNLMS [5] and NORMA [12], 2.) n denotes both the number of basis functions and dictionary size, whereas in the text, N denotes the dictionary size and is applied only in the context of KNLMS, NORMA and KRLS

5 CONCLUSION

This paper demonstrated the utility of employing the Fastfood algorithm for non-linear regression problems. Such an approach can be used to reduce the hardware requirements of kernel methods in applications demanding energy efficiency and real-time learning. A novel hierarchical systolic array architecture was described for minimising data transfers between processing elements in the computation of Fastfood. This utilised an efficient implementation of the Fast Walsh Hadamard Transform (FWHT), and our architecture is compatible with other fast transforms, such as the FFT. The reported design can sustain 500 MHz clock rates while supporting problems with an input dimensionality 10^3 times larger than other online kernel methods. Our work paves the way for real-time large-scale learning applications in control, communications and signal processing.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proc. DAC*. ACM, 1216–1225.

[2] Ronan Collobert and Samy Bengio. 2001. SVMTorch: Support vector machines for large-scale regression problems. *JMLR* 1, Feb (2001), 143–160.

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proc. NIPS*. 3123–3131.

[4] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. 2009. Discrete Fourier Transform on Multicores: Algorithms and Automatic Implementation. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"* 26, 6 (2009), 90–102.

[5] Nicholas J Fraser, Duncan JM Moss, JunKyu Lee, Stephen Tridgell, Craig T Jin, and Philip HW Leong. 2015. A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation. In *Proc. FPL*. IEEE, 1–6.

[6] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[7] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[8] Quoc Le, Tamás Sarlós, and Alex Smola. 2013. Fastfood-approximating kernel expansions in loglinear time. In *Proc. ICML*, Vol. 85.

[9] Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM TODAES* 17, 2 (2012), 15.

[10] Yeyong Pang, Shaojun Wang, Yu Peng, Nicholas J Fraser, and Philip HW Leong. 2013. A low latency kernel recursive least squares processor using FPGA technology. In *Proc. FPT*. IEEE, 144–151.

[11] Ali Rahimi and Benjamin Recht. 2007. Random features for large-scale kernel machines. In *NIPS*. 1177–1184.

[12] Stephen Tridgell, Duncan JM Moss, Nicholas J Fraser, and Philip HW Leong. 2015. Braiding: A scheme for resolving hazards in kernel adaptive filters. In *Proc. FPT*. IEEE, 136–143.

[13] Steven Van Vaerenbergh. 2012. Kernel methods toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering. *Grupo de Tratamiento Avanzado de Senal, Departamento de Ingenieria de Comunicaciones, Universidad de Cantabria, Spain* (2012).