Statistical Evaluation of a Self-Tuning Vectorized Library for the Walsh–Hadamard Transform *

Michael Andrews and Jeremy Johnson

Department of Computer Science, Drexel University, Philadelphia, PA USA

Abstract. Short vector instructions (SIMD) can significantly increase performance, yet are difficult to use effectively. Recently, several efforts (ATLAS, FFTW, SPIRAL) have successfully used automated performance tuning and search to find good SIMD implementations of high-performance kernels such as matrix multiplication, the FFT and related transforms. In this paper, we review techniques, similar to those used by SPIRAL, for vectorizing sparse matrix factorizations, and incorporate them into a package for computing the Walsh–Hadamard Transform (WHT). These techniques along with search are used to discover algorithms with close to theoretical speedup. Not all algorithms provide the same amount of speedup and it is not the case that the vectorized version of the best sequential algorithm leads to the best algorithm with SIMD instructions. The goal of this paper is to better understand the search space and which algorithms lead to the best speedup and overall performance. Several SIMD specific algorithm features are used to predict speedup and performance. The resulting performance model is highly correlated with runtime performance and measured speedup, and can be used to partition the search space and expedite the search for good vector performance.

1 Vectorizing the Walsh Hadamard Transform

The Walsh–Hadamard transform of a signal x of size $N = 2^n$, is the matrix–vector product $\mathbf{WHT}_N \cdot x$, where

$$\mathbf{WHT}_N = \bigotimes_{i=1}^{n} \mathbf{WHT}_2 = \overbrace{\mathbf{WHT}_2 \otimes \cdots \otimes \mathbf{WHT}_2}^{n}. \tag{1}$$

$$\mathbf{WHT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2}$$

and the $\otimes$ operator denotes the tensor or Kronecker product. Algorithms for computing the **WHT** can be derived by factoring the standard matrix representation into a product of sparse structured matrices [1]. Let $N = N_1 \cdots N_t$, where $N_i = 2^{n_i}$, then

$$\mathbf{WHT}_N = \prod_{i=1}^{t} \left(\mathrm{I}_{N_1 \cdots N_{i-1}} \otimes \mathbf{WHT}_{N_i} \otimes \mathrm{I}_{N_{i+1} \cdots N_t} \right) \tag{3}$$

* This work supported by DARPA through the DOI grant NBCH1050009 and the ARO grant W911NF0710416, by NSF grant ACI–0325687 and by a grant from Intel.

Let $x_{b,s}^{M}$ denote the vector $[x_b, x_{b+s}, \ldots, x_{b+(M-1)s}]$. The evaluation of $\mathbf{WHT}_N \cdot x$ using (3) can be expressed as a triply nested loop

$$\begin{aligned}
&R = N; \quad S = 1; \\
&\text{for } i = 1, \ldots, t \\
&\quad R = R/N_i; \\
&\quad \text{for } j = 0, \ldots, R-1 \\
&\quad\quad \text{for } k = 0, \ldots, S-1 \\
&\quad\quad\quad x_{jN_iS+k,S}^{N_i} = \mathbf{WHT}_{N_i} \cdot x_{jN_iS+k,S}^{N_i}; \\
&\quad S = S * N_i;
\end{aligned} \tag{4}$$

where $\mathbf{WHT}_{N_i}$ is an algorithm for computing the transform at stride. The computation of each $\mathbf{WHT}_{N_i}$ is computed recursively in a similar fashion until a base case of the recursion is encountered. Base cases are smaller transforms where the code is unrolled in order to avoid the overhead of loops or recursion. This factorization assumes that the algorithm works in–place and is able to accept stride parameters. Alternative algorithms are obtained through different sequences of the application of Equation 3. When $n_1 = \cdots = n_t = 1$ the algorithm is called *iterative* since there are no recursive calls. When $t = 2$ and $n_1 = 1$ and $n_2 = n-1$ the resulting algorithm is called *right recursive*, and when $n_1 = n-1$ and $n_2 = 1$ the algorithm is called *left recursive*. The right recursive and iterative algorithms are the two standard approaches to computing the **WHT** and they correspond the standard recursive and radix 2 iterative FFT algorithms. Vectorized versions of these algorithms are easily obtained since $\mathbf{WHT}_N \otimes \mathrm{I}_V$ naturally corresponds to applying $\mathbf{WHT}_N$ on vectors of length V [3]. Vectorization for specific vector lengths is achieved using *rules*, which correspond to a textual substitution operation. The algebraic manipulations required to derive these rewrite rules are discussed in depth in [2]. Both rules assume that the vector length $V = 2^i$. A key element in all following factorizations is the stride permutation matrix L_N^{NP} [3], which permute the elements of the input vector. For instance if $x = [x_0, x_1, x_2, x_3]$, then $\mathrm{L}_2^4 \cdot x = [x_0, x_2, x_1, x_3]$. The first rule describes how to vectorize a general term in the recursive factorization, e.g. the second for loop in 3.

$$R1_{P,V}(\mathrm{I}_M \otimes \mathbf{WHT}_N \otimes \mathrm{I}_{PV}) \rightarrow \mathrm{I}_M \otimes (L_N^{NP} \otimes \mathrm{I}_V)(\mathrm{I}_P \otimes \mathbf{WHT}_N \otimes \mathrm{I}_V)(L_P^{NP} \otimes \mathrm{I}_V) \tag{5}$$

Where $R1_{P,V}$ is a parameterized textual substitution operator, resulting in an easily vectorized expression. The operator is not applied unless the right identity has size greater than or equal to PV. This allows applying $R1$ with fixed V and varying P resulting in different interleaving strategies. The application of $R1$ can be viewed as code transformation that tiles the innermost for loop by a factor of V,

$$\begin{aligned}
&\text{for } k = 0, \ldots, M-1 \\
&\quad \text{for } i = 0, \ldots, P-1 \\
&\quad\quad \text{for } j = 0, \ldots, V-1 \\
&\quad\quad\quad x_{iV+j,PV}^{N} = \mathbf{WHT}_N \cdot x_{iV+j,PV}^{N}
\end{aligned} \tag{6}$$

The rest of the code transformation is realized by unrolling the loop by P, interleaving the loop iterations, vectorizing the arithmetic operations with vector operations of length V. For example, suppose that $N = 2$ and $V = 2$, then we may vectorize the innermost for loops in 5 as follows,

$$\begin{array}{l} \text{Unroll} \\ \text{for } i = 0, \ldots, P-1 \\ \quad x_{2i} = x_{2i} + x_{2i+2P} \\ \quad x_{2i+2P} = x_{2i} - x_{2i+2P} \\ \quad x_{2i+1} = x_{2i+1} + x_{2i+2P+1} \\ \quad x_{2i+2P+1} = x_{2i+1} - x_{2i+2P+1} \end{array} \rightarrow \begin{array}{l} \text{Interleave} \\ \text{for } i = 0, \ldots, P-1 \\ \quad x_{2i} = x_{2i} + x_{2i+2P} \\ \quad x_{2i+1} = x_{2i+1} + x_{2i+2P+1} \\ \quad x_{2i+2P} = x_{2i} - x_{2i+2P} \\ \quad x_{2i+2P+1} = x_{2i+1} - x_{2i+2P+1} \end{array} \rightarrow$$

$$\begin{array}{l} \text{Vectorize} \\ \text{for } i = 0, \ldots, P-1 \\ \quad x_{2i,1} = x_{2i,1} + x_{2i+2P,1} \\ \quad x_{2i+2P,1} = x_{2i,1} - x_{2i+2P,1} \end{array}$$

In this example the stride parameter is 1, though in general this is not always the case. Many SIMD architectures only support loads from contiguous data address, e.g non strided loads. We resolve this issue by using contiguous loads in conjunction with intra register shuffling instructions to simulate a contiguous load. Such non contiguous loads introduce the overhead of register shuffling as well as negating the effect of cache line prefetching, and incur more cache miss penalties. We also vectorize base case codelets using the recursive rule

$$R2_V(\mathbf{WHT}_N) \rightarrow \begin{cases} (\mathbf{WHT}_2 \otimes \mathrm{I}_{N/2})(\mathrm{I}_2 \otimes R2(\mathbf{WHT}_{N/2})) & N > 2V \\ \prod_{i=1}^{t} (\mathbf{WHT}_2 \otimes \mathrm{I}_V)\, \mathrm{L}_2^{2V} & \text{otherwise} \end{cases} \tag{7}$$

The stride permutations in the base case are achieved using intra register shuffle operations.

2 The WHT Package

The original WHT package [1] was extended to support vectorization and other architectural paradigms. Algorithm alternatives in the package, corresponding to different formulas, are described by a tree like plan which is denoted by the grammar

```
W(n) := small ( params ) [n]
     |  split ( params ) [W(n1), ... ,W(nt)]
```

Nodes in the tree compute the WHT when applied to an input vector. The `split` nodes correspond to the recursive factorization in (3), and compute the WHT by iterating over a series of smaller WHTs. The `small` leaf nodes correspond to the base case of the recursion and compute the WHT in place using straight line code. Alternative factorizations, such as (5) and (7) are specified by substituting the identifier (`small` or `split`) with an alternative identifier.

For instance, `smallv(2)`[4] represents a base case unrolled codelet of size 2^4 with SIMD instructions of length 2. New factorization rules can easily be added to the package via a lookup table.

3 Performance Distribution and Model

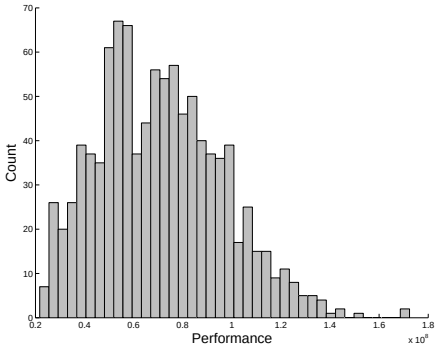
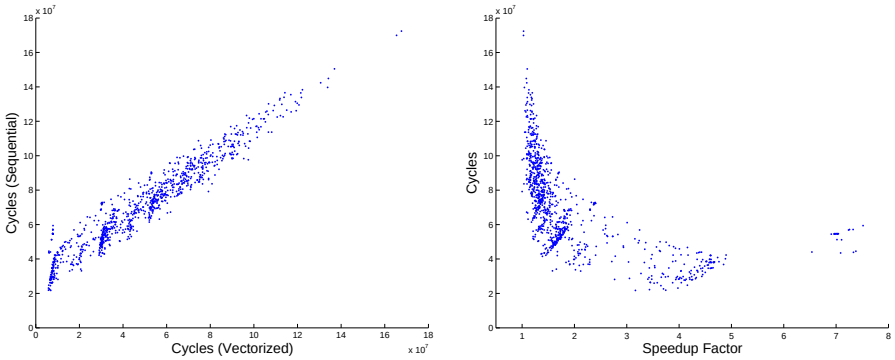


Fig. 1. Runtime distribution for random single precision WHTs of size 2^{18} on Intel Xeon.



(a) Runtimes after vectorizing sequential algorithms.

(b) Speedup after vectorizing sequential algorithms.

Fig. 2. Effect of vectorizing random sequential single precision WHTs of size 2^{18} on Intel Xeon.

In [4] it is shown that approximately $O(7^n)$ different algorithms can be obtained from (3). Figures 1 and 3 shows the distribution of possible

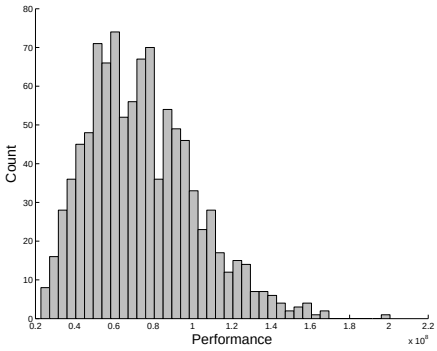
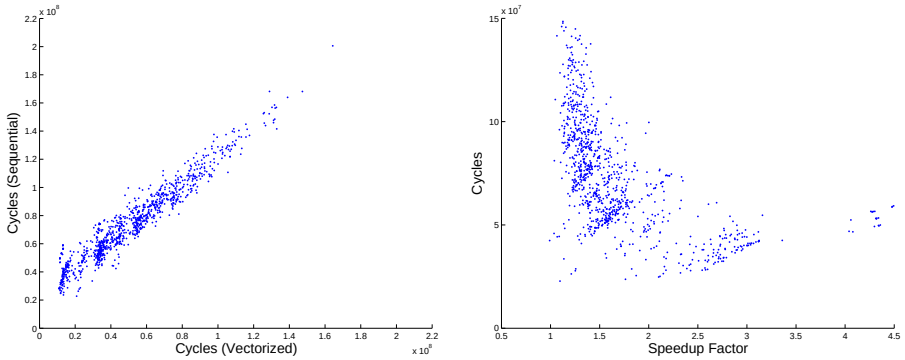


Fig. 3. Runtime distribution for random double precision WHTs of size 2^{18} on Intel Xeon.

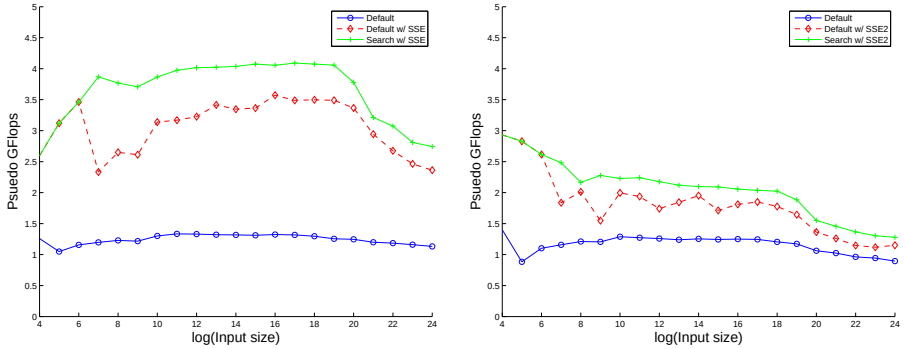


(a) Runtimes after vectorizing sequential algorithms.

(b) Speedup after vectorizing sequential algorithms.

Fig. 4. Effect of vectorizing random sequential double precision WHTs of size 2^{18} on Intel Xeon.

sequential runtimes from a sample of 1,000 random WHT algorithms of size 2^{18} using single and double precision respectively. Runtimes were collected on an Intel Xeon processor (E5335) running at 2.0 GHz with a 32 Kb L1 data cache and 4 Mb L2 unified cache. Performance measurements were taken with PAPI 3.5.0. Code was compiled with the Gnu C Compiler version 4.1.3 (`gcc`), using standard optimization flags `-O2`. and `-msse` for single precision transforms and `-msse2` for double precision transforms. Figures 2(a) and 4(a) show the relationship between sequential and vectorized runtimes for single and double precision respectively. Vectorization in this context was achieved using $R1_{8,4} \circ R1_{4,4} \circ R2_4$ and $R1_{8,2} \circ R1_{4,2} \circ R2_2$. Figures 2(b) and 4(b) show the speedup obtained by vectorizing using these rules. Observe that with both vector lengths the absolute performance is not linearly correlated with speedup.



(a) Best single precision WHTs (b) Best double precision WHTs

Fig. 5. Pseudo GFlops of best sequential and vectorized WHTs determined by dynamic programming.

This suggests that searching over all possible sequential WHT algorithms, and then applying our vectorization rules will not achieve optimal performance. To verify this hypothesis and benchmark the performance

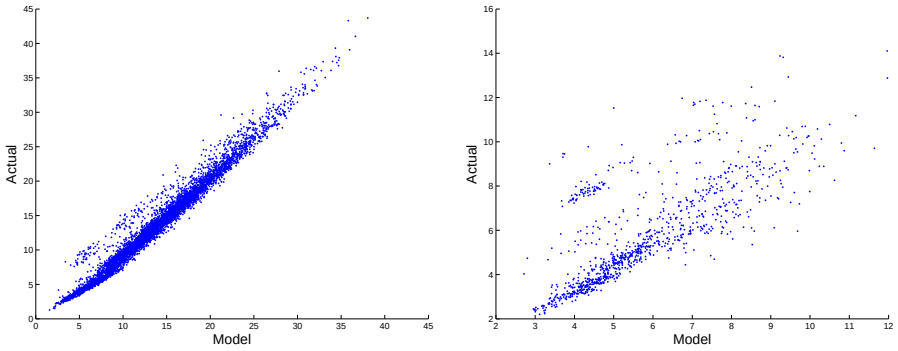
Remark	Sizes	`max-child`	`max-cache`	α	ρ	m
Fits in L1	[1, 11]	4	4	0.5 %	0.1 %	50
Fits in L2	[12, 15]	4	3	0.5 %	0.1 %	10
Fits in L2	[16, 18]	3	3	0.5 %	0.1 %	10
Fits in RAM	[19, 24]	2	2	5.0 %	1.0 %	5

Fig. 6. Parameters to Dynamic Programming.

of generated code, we performed a modified version of dynamic programming to search for optimal sequential and vectorized WHT We achieve close to the theoretical performance improvement with vectors of length 4 and better than theoretical performance improvement with vectors of length 2. Figures 5(a) and 5(b) show the result of this experiment. Our hypothesis is validated in that there is clear performance discrepancy between searching for the optimal sequential algorithm and then vectorizing as opposed to searching the optimal vectorized algorithm. For both vector sizes, the the first drop in performance corresponds to the switch from using unrolled codelets to a recursive factorization. The second drop in performance corresponds to the L2 cache boundary. Our modified dynamic programming search was configured with the following parameters: the maximum number of children in an integer composition tree (`max-child`), the maximum number of sub–problems to reference when determining the optimal problem (`max-cache`), and parameters which determine the statistical significance for each problem sample. Statistical significance is determined using a Z–Score test and is parameterized

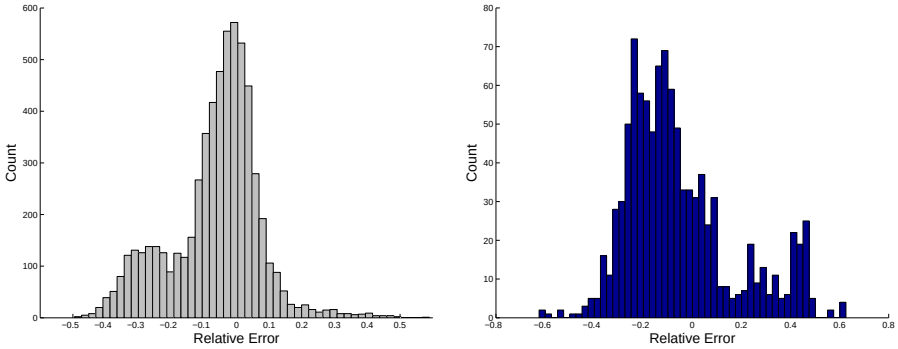
with $1 - \alpha$ % confidence that the sampled mean is within ρ % of the true mean using m initial samples. To efficiently navigate the exponential search space, many of the parameters need to be tweaked in order to have a tractable solution. Listed in Table 6, are search parameters associated with the transform size interval.

4 Performance Model



(a) Single precision model. (b) Double precision model.

Fig. 7. Modeled runtime versus actual runtime using the combined model in 8.



(a) Single precision model error. (b) Double precision model error.

Fig. 8. Distribution of error term ϵ in 8.

The goal of the performance model should be two fold– to speed–up the search process, and to mathematically quantify the performance gains

achieved by vectorization. Previously, [4–6] a linear combination of instructions and cache misses was used to predict performance and prune the search space.

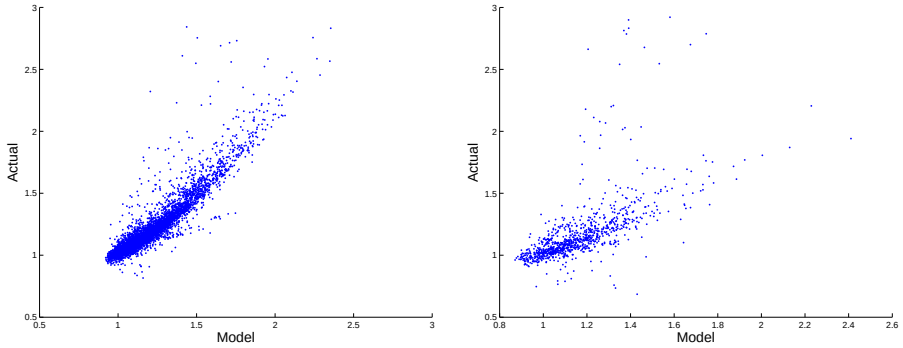
$$\widehat{\text{CYC}} = \alpha\text{INS} + \beta\text{MISSES} + \epsilon \tag{8}$$

The instruction counts and cache misses are computed analytically from recurrence relations, and the factor loadings are computed using multiple linear regression from training data. The evaluation of the model on vectors of length 2 and 4 is shown in Figs. 7(a) and 7(b). The distribution of the error term in the model is shown in 8(a) and 8(b). Our model was trained using a sample of 600 random WHT trees with random applications of $R1$ and $R2$ with fixed V and varying P. The random sample was across WHT trees from size 2^{13} to 2^{19} inclusively. Clearly, vectorization introduces some runtime behavior that is not effectively captured by our model. To remedy this situation we derived an auxiliary model in 9 which captures the speedup due to vectorization used in conjunction with the previous model.

$$\frac{\widehat{\text{CYC}'}}{\text{CYC}} = \frac{\omega P(\text{INS}')}{\text{INS}} + \epsilon \tag{9}$$

$$P(\text{INS}') = \left\{ \begin{array}{l} \text{SIMD ADD, SIMD LOAD,} \\ \text{SIMD SHUF, NON SIMD} \end{array} \right\} \tag{10}$$

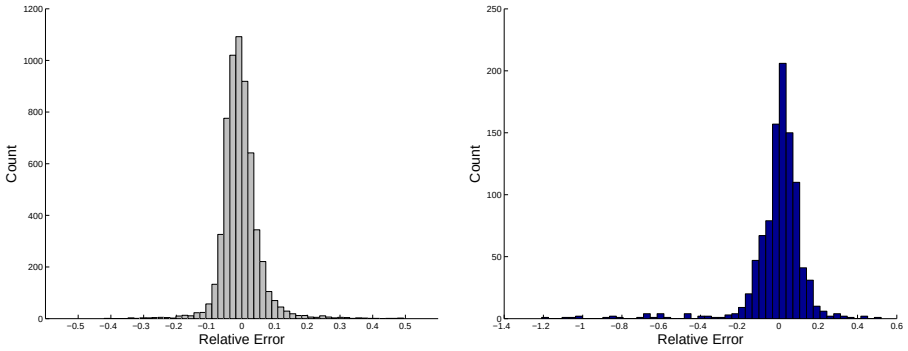
The speedup model achieves a tighter fit by partitioning the instruction counts into classes of instructions and assigning weights to these instruction. Since we could not natively count SSE and SSE2 instruction on this architecture using PAPI, the instruction count model from [4] was used. The evaluation of this new speedup model on vectors of length 2 and 4



(a) Single precision speedup model. (b) Double precision speedup model.

Fig. 9. Modeled speedup versus actual speedup using the model in 9.

is shown in Figs. 9(a) and 9(b). The distribution of the error term in the new speedup model is shown in 10(a) and 10(b).



(a) Single precision model error. (b) Double precision model error.

Fig. 10. Distribution of error term ϵ in 9.

- **TODO**: Try to fit speedup model with ratio of cache misses to get better fit for SSE2. This makes sense as per the discussion above, in which varying P results in different memory access patterns.
- **TODO**: Display and discuss sanity of coefficients in models.
- **TODO**: Combine both model to achieve tighter model for predicted runtime due to vectorization.

References

1. Johnson, J., Püschel, M.: In search of the optimal Walsh–Hadamard transform. In: Proceedings ICASSP. Volume IV. (2000) 3347–3350
2. Franchetti, F., Püschel, M.: A SIMD vectorizing compiler for digital signal processing algorithms. In: International Parallel and Distributed Processing Symposium (IPDPS). (2002) 20–26
3. Johnson, J., Johnson, R., Rodriguez, D., Tolimieri, R.: A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. Circuits, Systems, and Signal Processing **9** (1990) 449–500
4. Pawel Hitczenko and Jeremy Johnson and Hung–Jen Huang: Distribution of a Class of Divide and Conquer Recurrences Arising from the Computation of the Walsh–Hadamard Transform. Theoret. Comput. Sci. **352** (March 2006) 8–30
5. Furis, M., Hitczenko, P., Johnson, J.: Cache miss analysis of WHT algorithms. In Martínez, C., ed.: 2005 International Conference on Analysis of Algorithms. Volume AD of DMTCS Proceedings., Discrete Mathematics and Theoretical Computer Science (2005) 115–124
6. Andrews, M., Johnson, J.: Performance Analysis of a Family of WHT Algorithms. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International (26-30 March 2007) 1–8