EE482C Final Project Report

Data-level, Instruction-level, and Thread-level Parallelism on the Imagine Stream Processor

James Bonanno
Rex Petersen
Suzanne Rivoire

June 6, 2002

Overview

Using the Imagine stream processor as a baseline, we explored the effects of exploiting parallelism along three axes: data-level parallelism (DLP), instruction-level parallelism (ILP), and thread-level parallelism (TLP). We developed a cost model to evaluate different configurations in terms of area, and we wrote a JPEG-like encoder in StreamC and KernelC as a sample application on which to evaluate performance. We then selected five different cluster configurations on which to test ILP. Since the tools prevented us from directly simulating DLP or TLP, we developed a method to extrapolate the ILP results to these cases.

Our ILP results show that cluster configurations at least as large as the original Imagine configuration exhibit the best balance between area and kernel performance. They also show that for kernels well matched to the cluster configuration, the microcode size does not generally explode with increased ILP.

Performance / cost analysis for our JPEG application indicates that starting from the configuration of Imagine, it is first beneficial to exploit DLP by adding to the number of clusters. After the total number of clusters reaches 32, it is more efficient to divide them among several thread execution units – each having 8 clusters. For the JPEG application, changing the internal configuration of a cluster (targeting ILP) is not beneficial.

We concluded that the current Imagine processor should be the configuration of choice for single-threaded execution. In order to make recommendations about multithreaded execution, we tried to identify common types of kernels and the hardware best suited to each. Our results are speculative but suggest that a multithreaded stream processor should include a stripped-down, low-ILP execution unit in addition to a more powerful execution unit like the current Imagine processor.

Finally, we recognize that our results are preliminary and are weakened by our inability to directly test DLP and TLP and by some assumptions made in our models. We suggest future work to eliminate these sources of uncertainty.

1. Introduction

Our goals as stated in our proposal were to find the costs and bottlenecks of extreme DLP, ILP, and TLP and to recommend an optimal balance among the three in terms of cycle count and area. During the course of the project, we set the additional goal of classifying common types of kernels as a step to recommending non-uniform execution units for thread-level parallelism.

We developed the following tools to further these goals:

- A cost model normalized to the area of the original Imagine processor
- A JPEG-like encoding application
- Five different machine configuration files to test ILP
- A method for extrapolating ILP results to DLP and TLP

We present conclusions about:

- The performance and performance-area relationship of different levels of ILP
- The relationship between increased ILP and microcode size
- The best way to exploit DLP and TLP
- Tentative classifications of kernels and suggestions for thread execution units

Section 2 describes the sample applications we used for testing, and Section 3 describes our cost model. Section 4 explains our ILP tests and results, and Section 5 shows how we extrapolated these results to DLP and TLP. Section 6 presents our findings on classifying kernels, and Sections 7 and 8 conclude and recommend future research.

2. Sample Applications

We used several sample applications in order to evaluate performance of various machine configurations. Our analysis focused primarily on versions of a JPEG-like encoder, which we wrote. Additionally, we did limited analysis on the MPEG application provided with the Imagine tools as well as our implementation of a wavelet transform.

2.1 JPEG-like encoder

We chose to implement a JPEG-like encoder in StreamC and KernelC as a typical media application. It is part of the MediaBench [1] benchmark suite, and is a widely used image format

standard. Our implementation is a modification of the baseline sequential codec, illustrated in the following figure [2].

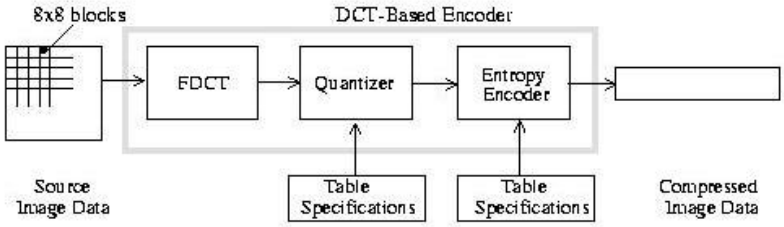


Figure 1. DCT-Based Encoder Processing Steps

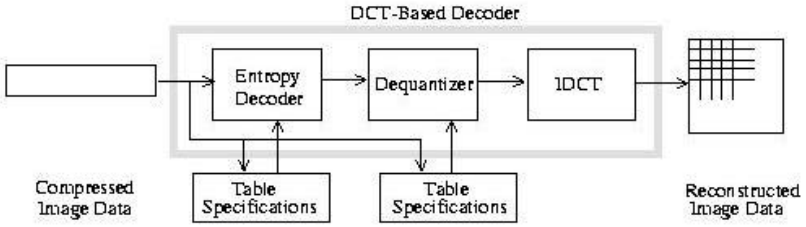


Figure 1: DCT-Based Decoder Processing Steps

For the sake of simplicity, we did not implement an entropy encoder, such as Huffman coding. Rather, we performed simple one-dimensional run-length encoding of the quantized coefficients. For source image data, we used a sample 8x8 portion extracted from a real image [2]. We also created larger images by repeating this 8x8 sample.

The following table presents our KernelC kernels and their variations.

Kernel	Description	Variations/notes
load_cos	Loads a stream of 128 cosine values into a persistent array used by some versions of dct.	Not used in Taylor series implementation of dct.
gen_idx_str	Given microcode variables indicating which 8x8 segment of the image is being processed, generates an index stream of length 64.	Implementation changes slightly as the number of clusters are varied. This is not taken into account with the performance model described later.
dct	Computes one DCT coefficient given the coordinates of that coefficient, and a stream of 64 pixel values of the current 8x8 segment.	Either uses a lookup table to compute cosine values, or uses a Taylor series approximation.

quantize	Divides elements of the first input stream by elements of the second stream, and rounds the result to the nearest integer.	This kernel actually performs a divide operation. If the second input stream's data were reformatted as its reciprocal, this operation could be replaced with a multiply.
rle1	Identifies the locations of "runs" in the image and uses conditional output streams to produce their values and locations.	In the sample image segment, there are 9 runs.
rle2	Transforms the stream of run length values and locations into a stream of run length values and lengths.	

Table 1: JPEG kernels

The equation for a discrete cosine transform (DCT) is:

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cdot\cos(\frac{(2x+1)up}{16})\cdot\cos(\frac{(2y+1)vp}{16})$$

$$C(a) = \frac{1}{\sqrt{2}}, a = 0$$

$$C(a) = 1, a \neq 0$$

When cosine is computed with a Taylor series approximation, the following formula is used:

$$\cos(x) \approx \sum_{n=0}^{N}(-1)^{n}\cdot\frac{x^{2n}}{(2n)!}$$

In our case, N=5, and we have pre-computed the constant multiplier for each term.

2.2 MPEG encoder and wavelet

We also examined kernel schedules from our fast wavelet transform application done as a homework assignment for this class, and for the MPEG sample application provided in the Imagine toolset. We did not simulate these programs in isim.

3. Cost Model

In order to compare different parallel configurations, we developed an area-based cost model. We derived a formula for cost that is based on the number of functional units in each cluster; the total number of clusters; and the total number of threads. This cost model is meant to give a relative area estimation for the region containing the ALU clusters and microcontroller of an

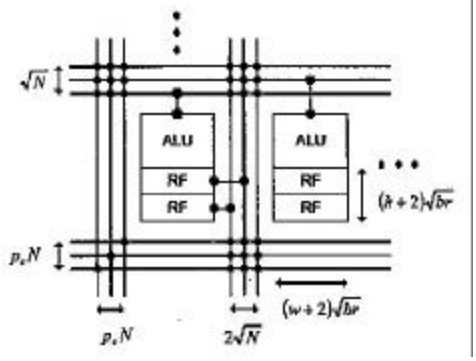
Imagine-like processor. This model does not include the area of SRF, memory system, and external interfaces. We chose to express cost roughly in terms of mm^2 and then scale the result so that the cost of Imagine is 100. Each part of the cost model is described in Table 2.

Variable	Value	Description
N		Number of <subscript> (i.e., N_C = number of clusters)
A		Area of <subscript> (i.e., A_C = area of a cluster)
Subscripts For N & A		T = Threads; C= Clusters; SP=Scratchpad A = Adder; M = Multiply; D = Div/Sqrt ALU= A, M,&D; CU=CommUnit; U=uController; UM=ucMemory
N_F		Number of functional units / cluster = $N_A + N_M + N_D + N_{SP} + N_{CU}$
A_{UC}	0.5	Average area of microcontroller decode logic / ALU in cluster
p_e	¼	Number of external ports per functional unit
b	32	Data width of the architecture
w	1.8	Wire pitch (typically 0.64 –2µm/wire in a 0.18µm process)
O_F	0.75	Overlap factor. % of functional unit area that switch can overlap

Table 2: Variables used in the cost model

3.1 Estimating cluster switch sizes

Each cluster has a switch that allows each functional unit to send data out of the cluster or store data into the local register files for other functional units. The area of this switch grows quadratically with the number of functional units inside the cluster. On the right is a diagram taken from *Register Organization for Media Processing* [3], which details a way to model a 2D version of such a switch. The basic formula for the switch size is:



$$\text{SwitchSize} = (p_e N + 2N)(p_e N + N) * w^2 * b^2$$

To estimate the size of this switch, we used a wire pitch of 2 µm to allow enough room for power, ground, and noise shielding wires. Using these values and converting to mm^2, the formula becomes:

$$\text{SwitchSize} = 2.81 * N^2 * 2^2 * 32^2 / 10^6 = 0.012\, N^2 \approx N^2 / 100$$

Taking this result and solving for wire pitch, we get 1.86 µm, which is still reasonable. We used the same formula for both the internal cluster switch (N_F^2 /100) and the intercluster communication switch (N_C^2 /100).

3.2 Cluster Area

To estimate the area of the functional units we again used Imagine as a reference, where the cluster contains 3 adders, 2 multipliers, 1 divider, 1 scratchpad, and 1 communication unit, for a total size of 7 mm^2. In our model, we scaled the number of scratchpads and communication units by adding one for every 5 ALUs. We estimated an area of 1 mm^2 for each multiplier or divider and 0.5 mm^2 for adders, scratchpads, and communication units, which makes:

$$A_F = 0.5\ N_A + N_M + N_D + 0.5\ N_{SP} + 0.5\ N_{CU}$$
$$N_{SP} = N_{CU} = \max(\ 1, \text{int}(N_{ALU}/5)\ )$$
$$A_C = \text{ClusterArea} = \max(A_F, O_F * \text{SwitchSize}) + (1- O_F) * \text{SwitchSize}$$
$$A_C = \max(A_F, O_F * N_F^2\ /100) + (1- O_F) * N_F^2\ /100$$

3.3 Microcontroller

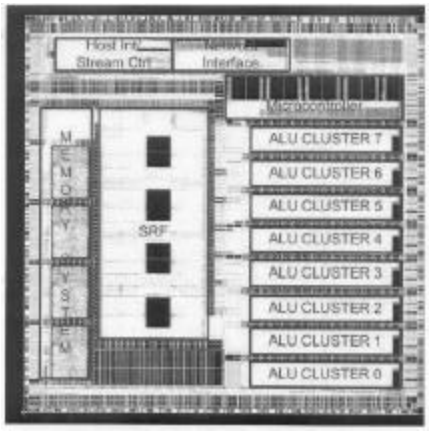
The microcontroller size is constant as DLP increases, since every cluster receives the same instruction. For TLP, each thread requires an additional microcontroller. The more interesting aspect of our cost model was how we chose to scale the microcontroller size as ILP increases. As described in Section 4.3, we observed that the total microcode size in terms of bits did not grow dramatically as we added functional units to each cluster. Therefore, the memory storage part of the microcontroller can remain constant, but the control logic and instruction decoders will have to grow as we scale ILP:

$$A_U = A_{UM}\ + A_{UC} * N_F$$

3.4 Complete formula for cost model

$$\text{Cost} = N_T\ (N_C A_C + N_C^2\ /100 + A_U)$$
$$\text{Cost} = N_T\ (N_C\ [\max(A_F, O_F * N_F^2\ /100) + (1- O_F) * N_F^2\ /100] + N_C^2\ /100 + 12\ + 0.5\ N_F)$$

The main limitation of this cost model is that is does not consider the entire chip area, which was beyond the scope of this project. It is an effective model to use when comparing various implementations that are similar in size, but not as useful when comparing configurations whose areas vary by more than 2x. From the layout of Imagine shown at the right, one can see that the area considered in our cost model in the cluster and microcontroller regions is about 40% of the chip. Therefore a cost of 200 may only increase the total chip area by 40%. A better model of the entire chip area would add a scaling model for the SRF and other components.



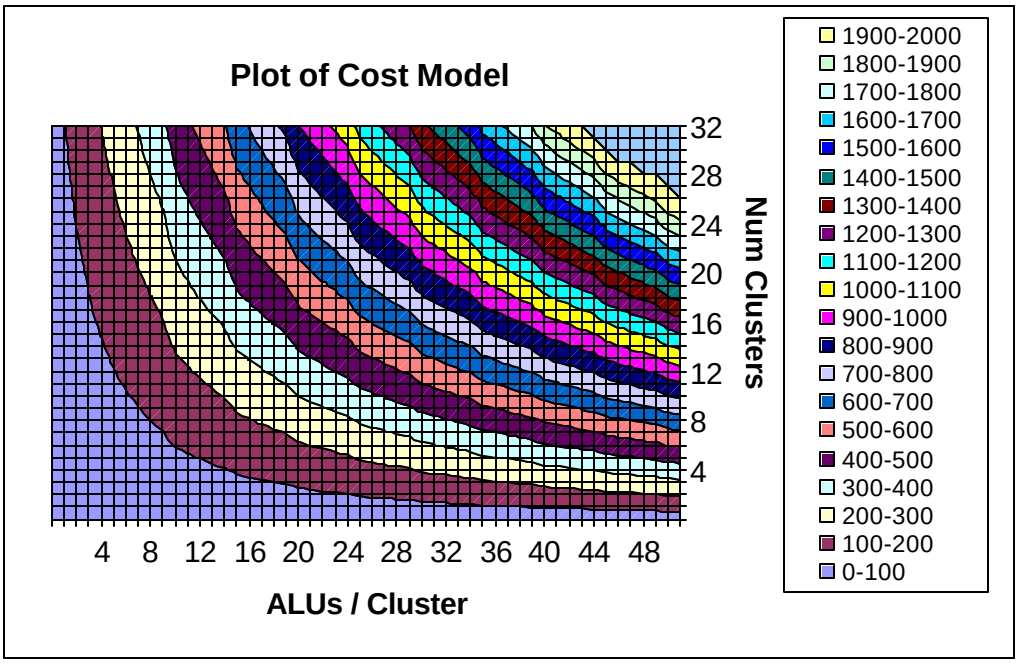


Figure 2: Plot of cost model function

4. Measuring Instruction-level Parallelism

To simulate varying degrees of instruction-level parallelism, we created five representative machine configurations (.md files) and examined results for three different applications. We studied the JPEG application the most closely, but we also looked at kernel schedules for the provided MPEG application and for our fast wavelet transform in order to check our intuitions.

4.1 Cluster configurations

The cluster configurations we tried are summarized below. Wimp exhibits extremely low ILP; Stud is extremely bloated. All of the machine configurations we tried have eight clusters. Unfortunately, we were not able to actually scale the number of scratchpads and communication units in our .md files as detailed in our cost model and as reflected in the area numbers. Our results should not be weakened by this limitation, however, since the kernels we analyzed most heavily were almost free of scratchpad and communication operations.

Name	#ADD	#MUL	#DIV	Norm. Area	Instr. Width (bits)
Wimp8	1	1	1	70	375
Tin8	3	1	2	100	478
Gold8	3	2	1	100	497
Straw8	6	4	2	181	705
Stud8	12	8	4	352	1129

Table 3: Five cluster configurations

4.2 Efficiency of the JPEG application

The performance per unit area (**efficiency**) is shown in Table 4 for both the kernel cycles and for the overall JPEG application. The numbers for the overall application reflect a high proportion of StreamC overhead, which is exaggerated because we used a small input, meaning less time spent in the kernels, and because we were unable to optimize our StreamC code. We suspect that the kernel performance numbers are therefore more representative.

Name	Kernel efficiency	Overall efficiency
Wimp8	1.14	0.82
Tin8	0.86	0.54
Gold8	1.00	1
Straw8	0.72	0.97
Stud8	0.38	0.81

Table 4: ILP data for the JPEG application, normalized to Imagine

Our data show that, at eight clusters, Gold8 and Straw8 yield approximately equivalent performance per unit area. Stud8, however, suffers because of "wimpy" kernels that are bottlenecked by short-stream effects and light computation.

4.3 ILP and microcode size

For kernels that are well matched to their execution hardware, increasing ILP does not necessarily increase microcode size; in fact, unrolled code on a larger execution unit is often smaller and faster than code that is merely pipelined on a smaller unit. We only observed code size bloat in two cases. The first is when hardware is grossly underutilized, meaning that a large proportion of the instruction width is unused. The second is when code is unrolled extensively for tiny performance gains on a large execution unit. The chart below shows this principle for the dct kernel, which is representative of the computationally heavy kernels we observed. The chart shows that code size on Stud8 does bloat when it is unrolled to achieve a small performance gain, but that no tradeoff between ILP (and thus increased performance) and code size exists in general.

Name	Directives	Main loop cycles	Code size (B)
Wimp8	pipeline(1)	55	6937
Tin8	pipeline(1)	55	8724
Gold8	pipeline(1) unroll(2)	30	5964
Straw8	pipeline(1) unroll(2)	15	4583
Stud8	pipeline(1) unroll(2)	7.5	7339
Stud8	pipeline(1) unroll(4)	7.25	9314

Table 5: ILP and code size. The directives given for each kernel are the ones that yielded optimum absolute performance.

5. Extrapolating Data-level and Thread-level Parallelism

Because the tools prevented us from changing the number of clusters, we developed a method to extract DLP and TLP performance data using the kernel schedules for the various cluster configurations. This approach neglects all execution time due to non-kernel events, such as loading microcode, and perhaps more significantly, memory operations.

From the microcode (.uc) file for a scheduled kernel, we can directly determine the number of instructions in each basic block. Furthermore, we know which of these blocks correspond to the kernel's loop body and which are outside the loop. For kernels with one loop, we model its execution time as:

$$NumCycles = \left[\left(LoopCycles \cdot NumLoopIter\right) + NonLoopCycles\right] \cdot NumKernInvoke$$

LoopCycles and *NonLoopCycles* are extracted from the .uc file. *NumKernInvoke* is a property of the StreamC code calling the kernels. *NumLoopIter* depends on *numTimes* (determined from either the amount of data sent to the kernel, or a constant, or dependent on the specific data sent to the kernel), *numClusts* (the number of clusters), and *unrollAmt* (the extent to which the loop is unrolled), as expressed in the following formula.

$$NumLoopIter = MAX\left(1, \frac{numTimes}{numClusts \cdot unrollAmt}\right)$$

Knowing the length of a single kernel invocation and understanding the StreamC dependencies between kernel invocations, it is possible to develop schedules for the kernels distributed among **thread execution units (TEUs)**. Performance data can then be extrapolated from the ILP data.

For the case of the JPEG application, the execution time of DCT is about 98% of the total execution time. Therefore pipelining the different stages of the process would not be beneficial. So for increased TLP, different TEUs would process different 8x8 segments of the image.

The following figures shows how efficiency of the JPEG applications changes as support for DLP varies. The results are normalized to the efficiency of one Gold8 TEU running the particular application.

The first result is for the JPEG-Taylor implementation. It is interesting to note that for this application, Gold8 (Imagine) is the most efficient. Also, going from 8 to 16 clusters gives nearly identical efficiency. This leads us to the tentative conclusion that given more area, using it to exploit DLP would be most cost-effective first step. These results also show that for the configurations targeting increased ILP (Straw and Stud), efficiency drops rapidly after DLP is increased past a certain point. Tin shows very low efficiency for this application because we replaced divide instructions by multiplies whenever possible. As expected, with kernels that have no division instructions, replacing a multiply unit with a divide unit is not at all beneficial.

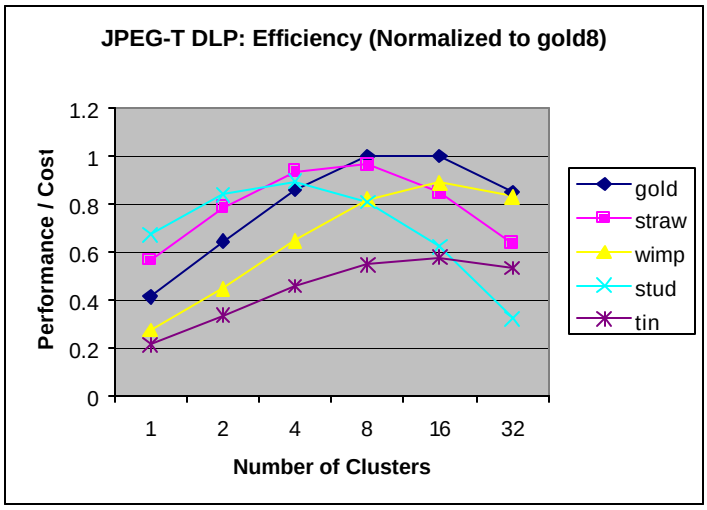


Figure 3: JPEG (Taylor) DLP Efficiency vs. Num. Clusters

The following figure shows the performance of JPEG-D, that is the version of JPEG that is divide-limited, or Dumb, since divisions by a constant were not transformed into multiplications.

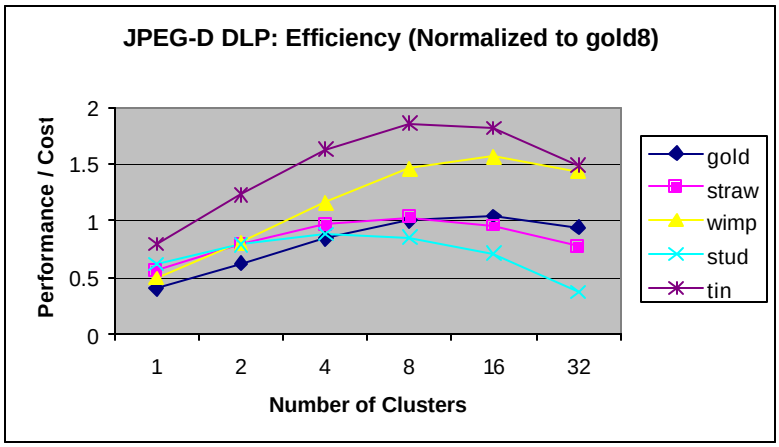


Figure 4: JPEG (Lookup, Divide) DLP Efficiency vs. Num. Clusters

It can be seen that in this case, the Tin configuration is the most efficient, since it has traded a multiply unit for an additional divide unit. Comparing this figure to the previous one demonstrates the fact that the specific implementation of a kernel can drastically affect performance. Certain types, or implementations, of kernels are significantly more efficient on certain cluster configurations.

In the following graph showing the efficiency for JPEG-S, the smart lookup-table version that has divides converted into multiplies, we see that the Tin configuration does not fare as well in the dumb implementation. This does show the advantage of the Tin configuration on a kernel that is divide-limited.

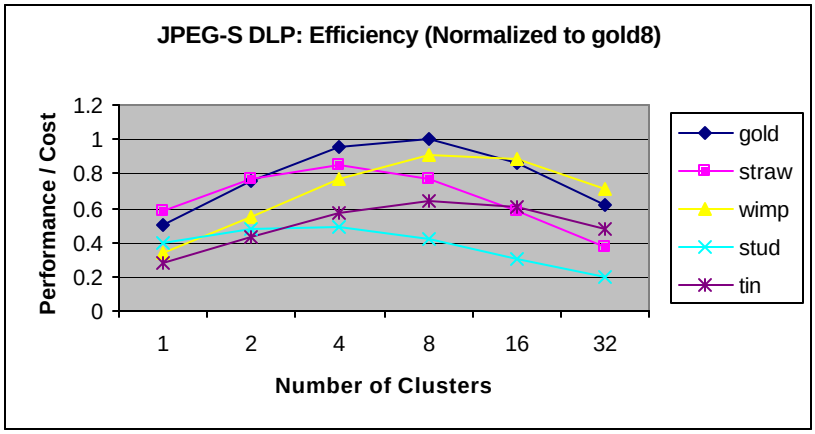


Figure 5: JPEG (Lookup, Smart Mult) DLP Efficiency vs. Num. Clusters

The next two figures show how the efficiency of JPEG (Taylor) changes as a constant number of identical clusters are redistributed among varying numbers of TEUs.

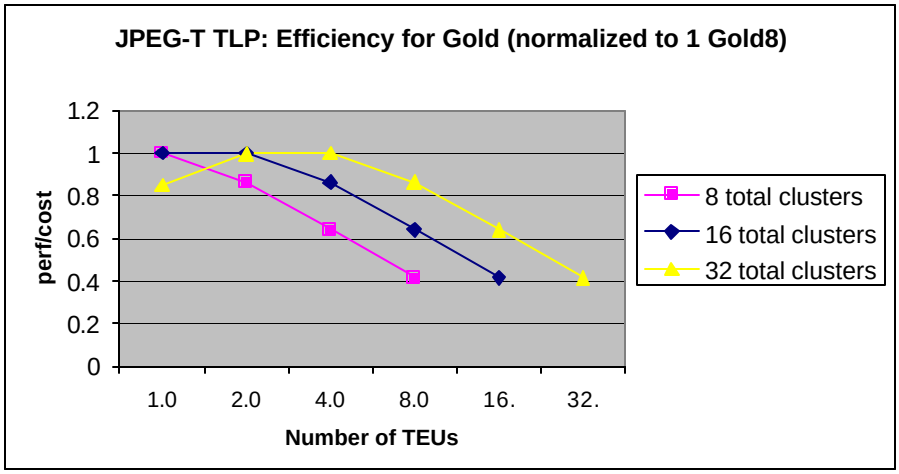


Figure 6: JPEG (Taylor) TLP Efficiency vs. Num. TEUs of Gold

Because there are no dependencies among the different TEUs, we assume ideal speedup in terms of raw performance. This is why one Gold8 has the same efficiency as two Gold8s, which is the same for four Gold8s. 32 or more total clusters should be arranged into multiple TEUs, while it is not beneficial to do so with configurations having fewer numbers of total clusters.

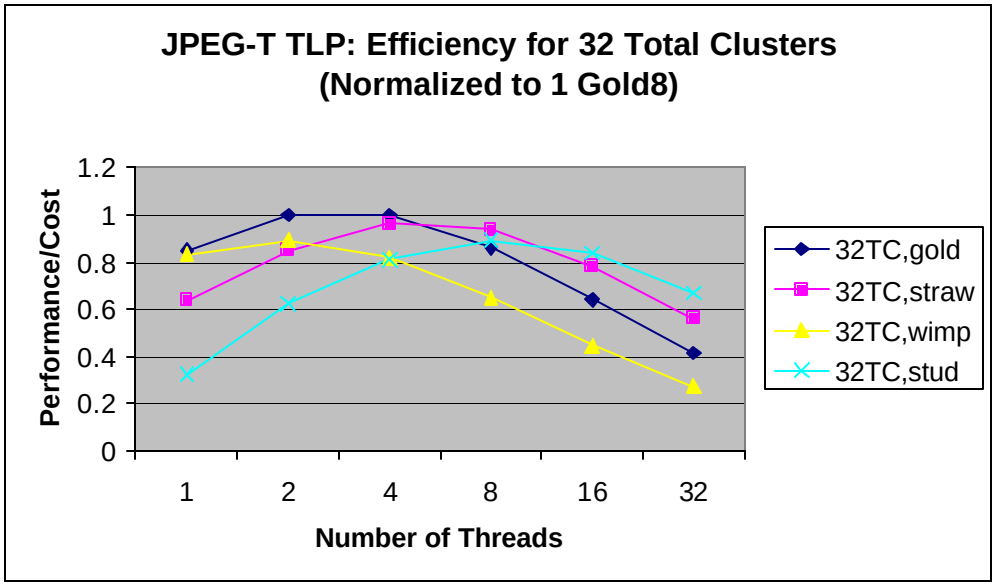


Figure 7: Efficiency for 32 Total Clusters vs. Num. Threads
8 or 16 total clusters yields the same graph, shifted to the left.

The TLP results in actuality are a restatement of the DLP results for this application, namely that using Gold8 or Gold16 configurations gives the highest efficiency. Having multiple TEUs of those configurations is more efficient than constructing single TEUs with additional clusters.

6. Classifying Kernels

We made a preliminary effort to identify common types of kernels and the execution hardware best tailored to each. Our wavelet and JPEG code contained two major types of kernels, described below. However, the kernels in the provided Imagine sample applications were more intense than ours and exhibited different characteristics; therefore, we recommend that further attempts to classify kernels examine the work of many different programmers.

6.1 Wimpy kernels

Wimpy kernels are kernels that run approximately equally well on all five hardware configurations. These kernels tend to be computationally light and hard to optimize because of intercluster communication and short stream effects that prevent unrolling. In our JPEG program,

gen_idx_str and the run length encoding kernels were wimpy, as were all of our wavelet kernels except FIR. Wimp8 yields the best efficiency for these kernels.

6.2 Functional unit-limited kernels

The rest of our kernels were limited by a single functional unit; doubling or quadrupling the number of that functional unit doubled or quadrupled the performance of that kernel. These kernels were most efficient on Straw8 and Stud8, since double or quadruple speedup over Gold8 could be achieved without as large an increase in area.

7. Conclusions

Below are our conclusions with respect to the goals stated in our proposal.

7.1 Consequences of pushing parallelism

ILP: The effects of adding additional functional units in each cluster were very kernel-specific, helping kernels that were FU-limited, but leaving wasted resources for wimpy kernels. The cost of the additional microcontroller and the internal cluster switch make configurations with greater than 10-12 ALUs per cluster show diminishing returns.

DLP: To maintain a constant number of ALUs when we pushed in the DLP direction, we had to use wimpy clusters with few ALUs. The main overhead then became the cost of the scratchpad and communication unit in every cluster amortized over fewer ALUs.

TLP: The additional microcontroller for every thread is the major cost when pushing TLP. The big win could come from a small TEU that performs as well as a larger one for some kernels at a fraction of the cost. Wimp8 is therefore optimal for the many wimpy kernels.

7.2 Preliminary recommendations on aspect ratio

For a single-threaded implementation, the aspect ratio of Imagine achieves a high efficiency on the applications we tested. As a rule of thumb, the best aspect ratios will be those that allow as many ALUs on the chip as possible at minimum cost. Because of the N_F^2 and N_C^2 terms in the cost model there is a first-order minimum cost point with this aspect ratio:

$$N_{ALU} = N_C = \sqrt{TotalNumberALUsOnChip}$$

For a chip with 48 ALUs like Imagine, this would be a value of about 7. Because the microcontroller area grows with Nf, it is better to have $N_{ALU} < N_C$ so setting $N_{ALU} = 6$ and $N_C = 8$ is a very minimal cost implementation for 48 ALUs. For a very simple performance model that assumes every ALU on a chip can be fully utilized, a plot of total ALUs / cost is shown below (again, the values have all been scaled so that the value for Imagine is 100).

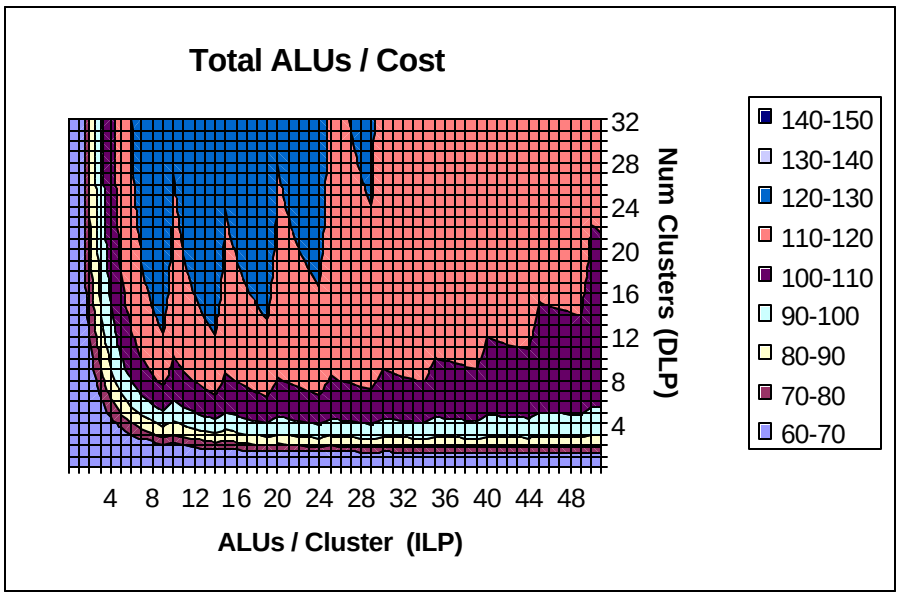


Figure 7: Plot of total ALUs/cost as a simple efficiency metric

Note that this plot assumes full utilization of every ALU, which is a major simplification of reality that will especially break down at high numbers of ALUs/cluster. Even with this limitation, there are some interesting points to note about this plot:

- The dark blue region is the best place to be from a performance/cost standpoint.
- With 8 clusters, each subsequent ALU added to a cluster increases the performance by about 10% until the ALUs/cluster reaches 8, after which the incremental gain drops. Imagine has 6 ALUs/cluster so it may benefit from adding 2 more.
- 16 clusters with 8 ALUs/cluster would be 30% better than Imagine.
- The jagged edges are caused by the scratchpad and communication units that are added for every 5 ALUs per cluster.

As fabrication processes improve and more area is available, we would recommend:

1. First push in the DLP direction to 16 or 32 clusters.
2. Push ILP slightly by adding another DIV unit. Fully pipelining the single DIV unit would also be beneficial.
3. Push TLP, since results suggest that multithreading when you hit 32 clusters is beneficial.

4. Explore non-uniform TEUs. Wider ILP TEUs for computationally intensive kernels, smaller custom clusters for wimpy kernels, etc.

8. Recommendations and Future Work

This section contains our recommendations for the developers of the Imagine toolset and for researchers who wish to extend our work.

8.1 Improving the Imagine toolset

We have two suggestions for improving the Imagine toolset. The first is to automatically generate .md files with a script that prompts the user to enter the number and kinds of functional units per cluster and the number of clusters. This task should be straightforward and would alleviate the tedium and difficulty of manually making the many changes to .md files.

Our second suggestion is to have the Imagine compiler change divide operations in kernels to multiply operations whenever possible, since the divide unit is not fully pipelined and therefore is less efficient. Perhaps the compiler could detect exceptions to this rule as well, like kernels that fully utilize the multiplication units already.

8.2 Exploring thread-level parallelism and kernel classification

Additional work can be done to further classify types of kernels from more applications and coding styles. Once a set of targeted kernels is understood, explore non-uniform TEUs that will perform well on these kernels with minimal hardware. We also recommend that future researchers continue use of the "wimp" and "stud" nomenclature, which we are very proud of.

8.3 Scalability

Further research could take scalability into account more fully than we did. Future researchers may want to develop a cost model that scales the SRF and other areas of the chip that we left constant, particularly as they study TLP. Additionally, our inability to actually add scratchpads and communication units would seriously hinder our ability to test a wider range of applications.

9. References

1. “MediaBench Home”, http://www.cs.ucla.edu/~leec/mediabench/

2. Wallace, Gregory K. "The JPEG Still Picture Compression Standard", Communications of the ACM, April 1991 (vol. 34 no. 4), pp. 30-44.

3. Rixner, Scott et al. “Register Organization for Media Processing”, p.4