Processor/Memory/Array Size Tradeoffs in the Design of SIMD Arrays for a Spatially Mapped Workload*

Martin C. Herbordt
Owais Kidwai
Anisha Anand
Renoy Sam
Charles C. Weems

Department of Electrical and Computer Engineering
University of Houston
Houston, TX 77204-4793

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract: Though massively parallel SIMD arrays continue to be promising for many computer vision applications, they have undergone few systematic empirical studies. The problems include the size of the architecture space, the lack of portability of the test programs, and the inherent complexity of simulating up to hundreds of thousands of processing elements. The latter two issues have been addressed previously, here we describe how spreadsheets and tk/tcl are used to endow our simulator with the flexibility to model a large variety of designs. The utility of this approach is shown in the second half of the paper where results are presented as to the performance of a large number of array size, datapath, register file, and application code combinations. The conclusions derived include the utility of multiplier and floating point support, the cost of virtual PE emulation, likely datapath/memory combinations, and overall designs with the most promising performance/chip area ratios.

1 Introduction

It is commonly recognized that the computational characteristics of many low-level, and even some intermediate-level, vision tasks map well to massively parallel SIMD arrays of processing elements. Primary reasons for this are that these vision tasks typically require processing of pixel data with respect to either nearest-neighbor or other two-dimensional pattern exhibiting locality or regularity. Tasks within this domain often require complex codes. As a consequence, gauging computational requirements is facilitated by evaluating performance with respect to real program executions. What we examine here are the effects on both cost and performance of varying the array size (number of PEs), the datapath, and the memory hierarchy with respect to a set of computer vision applications.

*This work was supported in part by an IBM Fellowship; by the NSF through CAREER award #9702483; by DARPA under contract DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; and under contract DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Laboratory.

The design issues are as follows. The first is to rank datapath and memory designs with respect to performance, eliminating those with poor cost/performance ratios. The second is to see how the performance of these designs varies with array size. This is complicated by the fact that although the effect of the datapath on performance varies predictably with array size, that of the memory hierarchy does not. The next issue is to relate the datapath to the memory designs to find which combinations result in i) well-balanced systems and ii) systems which could be enhanced with PE cache. Finally, the overall designs (including array size, datapath, and memory) with the best performance/chip area ratios are determined.

The constraint that we wish to examine processor designs by running real application programs, together with the wide variety of designs to be evaluated, clearly requires that simulation be used. Many problems remain, however, including how to i) implement a simulator with the flexibility to emulate large numbers of models without requiring substantial recoding, ii) achieve throughput given the slowdown inherent in software simulation, and iii) guarantee fairness for a wide range of designs that may not share the same optimal algorithm for any particular task.

These issues are dealt with by the ENPASSANT environment; the latter two are discussed elsewhere [6, 4], the former has been addressed recently and is described here. The key idea is to use spreadsheets coupled with tk/tcl and C code to specify templates (classes of architectures) relating design parameters to the performance of virtual machine constructs. The user then can specify models (specific designs) within those templates from a graphic user interface.

2 Architecture and Application Domains

2.1 Architecture Space

Massively Parallel SIMD Arrays (MPAs) are asymmetric, having a controller and an array consisting of from a few thousand to a few hundred thousand processing elements (PEs). The PEs execute synchronously code broadcast by the controller. One consequence is that PEs do not have individual micro-sequencers or other control circuitry; rather

their 'CPUs' consist entirely of the datapath. Within this constraint, however, there are few limitations. See Figure 1. Here we examine array sizes of from 4K to 64K PEs.

The complexity of current PE datapaths ranges from the MGAP which does not have even a complete one-bit ALU [9] to the MasPar MP2 which contains a 32-bit datapath and extensive floating point support [1]. The particular features that are examined here are the ALU/datapath width, the multiplier, and the floating point support. The effect of other features, such as the number of ports in the register file, have been found to be second order.

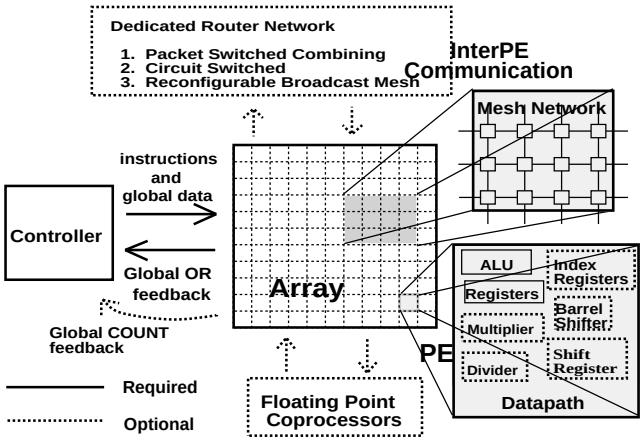

Figure 1: Shown is some the MPA architecture space. Note that PEs do not have local control and that their number ranges from the thousands to 100's of thousands.

Another consequence of SIMD control is on MPA PE memory configurations: Each array instruction operates on the same registers and memory locations for each PE.[1] In this case, it is possible to model the memory hierarchy, including cache, with a simple serial model. See Figure 2.

Existing MPAs and MPA designs contain on-chip memory in the range of 32 to 1024 bits.[2] Off-chip memory typically consists of DRAM, although SRAM has also been used. Although no MPA has yet been built with PE cache, studies have shown the clear benefits [8]. The particular features we examine here are the register file size from 128 to 1600 bits and the cache size. We do not examine very small register file sizes because our programmer's model requires at least three 32 bit registers. We assume enough main memory to store all the data the programs require, and test caches up to the sizes necessary to achieve a 99% hit rate.

2.2 Application Space

The fundamental criterion for selecting test suite programs is that they should be representative of the expected workload of the target machine. Since SIMD arrays, however, are not fully general purpose processors, general purpose benchmarks such as SPEC [12] are not appropriate. In fact, it is

[1] An exception is for processors which support a local indexing mode where the register can be a pointer to a memory location—as use of this mode is not critical in our current application test suite we postpone this analysis to a later study.

[2] We refer to on-chip memory as *register file* because of its explicit control.

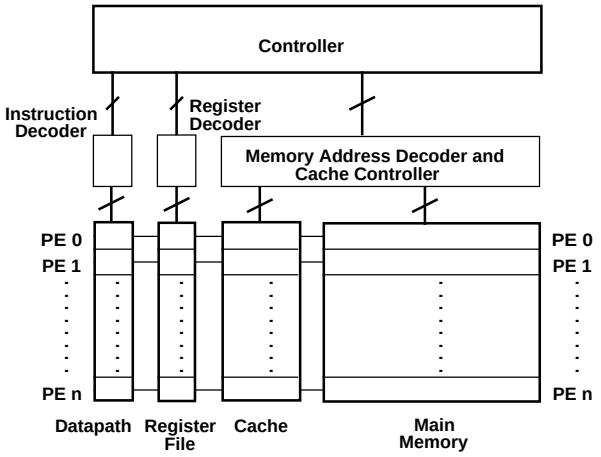

Figure 2: Shown is the MPA PE memory design space. Note that all PEs share the same controllers.

becoming more and more clear that, for many application domains, evaluation of parallel processors requires using application specific test suites. That this has long been the concensus of the computer vision community is shown in [10, 11, 13].

Our test suite consists of the parts of the second IU Benchmark particularly suited to MPAs, plus other important vision tasks. These inlude a convolution-based correspondence matcher, a motion-from-correspondence analyzer, an image preprocessor that uses complex edge modeling, and two segmentation-based line finders.

Some general comments are as follows: they are all non-trivial in that they consist of from several hundred to several thousand lines of code; they are, with the exception of the IU Benchmark, in use in a research environment; and they span a wide variety of types of computations. As an example of the last point, some are dominated by gray-scale computation (which is mostly 8 bit integer), others by floating point. They are described in detail in [3].

2.3 Mapping Applications to Architectures: Virtual PEs

The standard MPA programmer's model maps elements of parallel variables to individual PEs. It is rarely the case, however, that a processor has enough PEs to create this precise mapping; rather, some number of elements must be mapped to each PE.

In order to align the programmer's model with physical machines, the PEs assumed by the programmer's model (referred to as virtual PEs or VPEs) are *emulated* by the physical PEs. We refer to the number of VPEs each PE must emulate as the VPE/PE ratio, or VPR. This emulation can either be done entirely in software, through hardware support in the array control unit (ACU), or through a combination of the two. One of the fundamental goals of this study is to assess the cost of VPE emulation.

We now briefly describe how VPE emulation is carried out. VPE variables are mapped VPR elements per PE. We refer to a single physical slice of a VPE variable across the array of physical PEs as a tile. For simple non-interacting instructions such as those for logic, arithmetic, and activity

control, VPR physical instructions must be executed, one for each tile. Emulating interacting VPE instructions such as feedback and communication is substantially more complex.

The best performance is achieved when as many instructions as possible are executed within each tile; this is because a change of tile is effectively a context switch. Tiles *must* be swapped, however, whenever feedback or communication instructions are encountered. This is because feedback instructions cause control hazards in the controller and communication instructions inherently involve tile interaction. Compilers, such as that for ICL [4], can schedule instructions to minimize tile swapping.

The overhead due to VPE emulation therefore has several components: i) a linear component due to VPR tiling, ii) an increase in working set resulting in a decrease in locality of reference, iii) a sub-linear increase in working set size due to instruction scheduling, and iv) further overhead due to emulations of communication instructions [5].

3 Apparatus: ENPASSANT

There are two fundamental ideas behind our work. The first is to use only a single programmer's model, which we call the MPA virtual machine (VM), to cover the entire class of MPAs. To make this work without introducing the tremendous inefficiencies that can be caused by a mismatch between programmer's model and target machine requires two pieces of software: one provides emulations of features present on some, but not all, machines; the other contains critical application specific functions.

The second key idea is what we call *trace compilation.*[3] All code is written in a single data parallel language that defines the MPA VM. That code is not compiled directly to a target machine instruction set; rather, it is run on an MPA VM emulator. In the process a 'coarse-grained' trace is generated. This trace is then refined through a series of transformations wherein greater resolution is obtained with respect to the details of the target architecture. This process has two benefits: 1) virtual machine emulation and trace compilation together are one to two orders of magnitude faster than simulating an MPA at the instruction level, and 2) once the trace has been generated, it can be reused to evaluate a large number of design alternatives.

The simulator is perhaps best illustrated through its usage: these are the basic steps in using the system (see Figure 3):

1. Write (or select) and compile an application program in ICL, a data parallel language [2].
2. Execute the program (on any machine which supports C++) with trace generation turned on.
3. Specify or select a target architecture.
4. Run the trace through the target-architecture-dependent transformations and trace analysis tools which culminate with reporting performance data.

The major simulator components follow. The first three comprise the input constructor.

[3]Note that trace compilation is not related to the work by Ellis and Fisher involving trace-scheduling compilers.

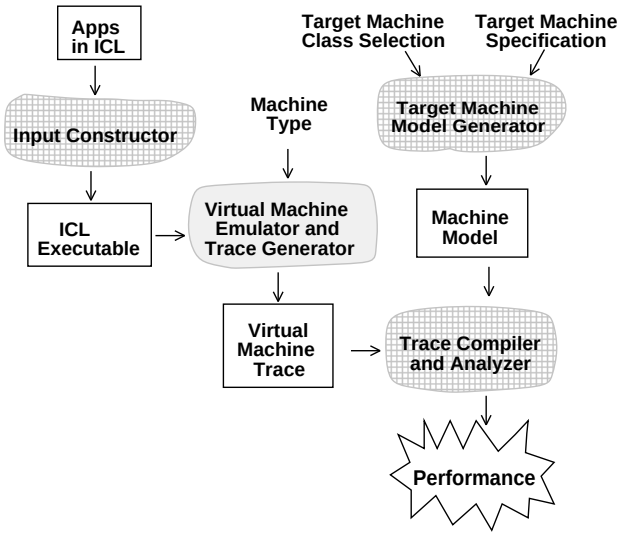

Figure 3: Shown are major components of ENPASSANT.

- **ICL language.** ICL is a data parallel language similar to C∗ and MPL. It has been designed specifically to present the application programmer with the MPA Virtual Machine. ICL is constructed entirely with C++ class libraries making it easily extensible should the VM be changed.
- **Operator emulation library.** The OEL is a library of ICL functions that emulate in software those hardware features present in some, but not all, MPAs. The OEL guarantees that ICL programs are portable within the class of MPAs.
- **Application function library.** The AFL is a library containing a small number of critical ICL application functions coded using different algorithms. The AFL is essential when the choice of algorithm is highly architecture dependent.
- **VM trace generator.** Code embedded within ICL generates the VM trace whenever an ICL program is executed with the trace generation option turned on.
- **Target machine specifier/model generator.** These programs provide a GUI for specifying MPA templates (i.e. families of machines) and variations within those templates. The output is the target machine model used by the trace compiler.
- **Trace compiler.** Inputs a VM trace and a target machine model and reconstructs what the application program trace would have looked like had it been generated by that particular target machine.

The ideas behind the input constructor and the trace compiler have been described in detail elsewhere [6, 4]. The target machine specifier has not, however.

As shown in Figure 4, machine specification is done in two levels. To create a specification for a family of machines, the user creates a template. This process consists of selecting elements from a menu (with the option of adding new elements), indicating whether those elements are optional or required, and providing the legal parameter ranges. Elements

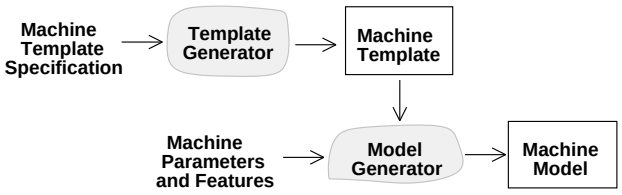

Figure 4: Shown is the Target Machine Model Generator. Templates define classes of target machines which have similar relationships among components. The parameters specify which components are present and their characteristics.

present in all MPAs are things like ALUs, neighbor communication networks and registers. Optional elements in a template can include, e.g., floating point units and broadcast networks. Typical parameters are ALU width, neighbor communication latency per bit, and number of ports in the register file. The template must also specify how the elements are related to each other, to the parameters, and to the virtual machine.

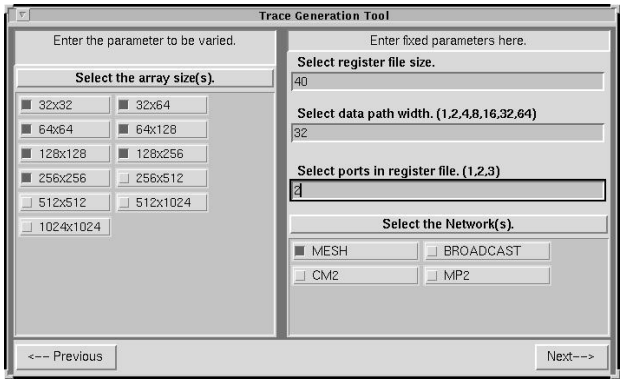

Figure 5: Shown is a page from an evaluation dialogue. The user is requesting that a particular target machine specification be used for 7 different array sizes.

The particular target machines are specified by choosing from the menu suggested by the template. The template-based relations and machine specifications are then combined to create target machine models. Since particular machines can be specified in great detail and because of the speed of evaluations, users have the option of inputting ranges of parameters. The following is a typical request.

- a set of application programs
- a set of array sizes (e.g. "256x256, 128x128, 64x64")
- a set of register file sizes (e.g. "10, 20, 40")
- a default prespecified datapath (e.g. "MasPar MP1-like")
- cache parameters (e.g. "no cache")
- a default prespecified communication network (e.g. "CM2-like")

A page from a similar dialogue is shown in Figure 5.

Template generation is time-consuming and it requires expertise to create one that is useful. Once created, however, it can be used to evaluate a huge space of machines. We also note that templates need to be generated only very rarely: the entire space of MPAs that we have considered so far in our research was specifiable using a single template.

4 Experimental Methodology

Complexity Ordering	ALU Datapath	Multi-plier	FP Support (inverse)
Level 1	1		
Level 2a	1		1/32
Level 2b	2		
Level 3a	2		1/16
Level 3b	4		
Level 4a	4	4	
Level 4b	4		1/8
Level 4c	8		
Level 5a	8	4	
Level 5b	8		1/8
Level 6a	8	8	
Level 6b	8		1/4
Level 6c	16		
Level 7a	16	4	
Level 7b	16		1/8
Level 8a	16	8	
Level 8b	16		1/4
Level 8c	32		
Level 9a	16	16	
Level 9b	16		1/2
Level 9c	32	4	
Level 9d	32		1/8
Level 10a	32	8	
Level 10b	32		1/4
Level 11a	32	16	
Level 11b	32		1/2
Level 12a	32	32	
Level 12b	32		1

Table 1: levels of datapath complexity.

The ideal situation is to search the space of possible designs optimizing for cost/performance, or, given a cost (or performance) goal, finding the design with the best possible performance (or cost). The critical problem for this approach is the difficulty in determining the costs of designs since there are so many variations in technology, process, etc. What we have done instead is to rank designs by a rough measure of complexity and determine the performance of a large sample of those designs. We believe this is valuable for two reasons: i) a design whose complexity is not warranted by its performance can be eliminated, and ii) non-linearities, or discontinuities, can be found which can help determine promising areas for detailed study. We also recognize that these rankings are implementation dependent and are likely to vary somewhat with changes in technology and design.

The space of designs is daunting due to the number of axes along which parameters can be varied. In order to make this study tractable, we have reduced the number of axes to three: datapath complexity, memory complexity (with and without cache), and array size. In order to do this, we have built upon the following assumptions:

- With some exceptions, increases in complexity of most individual design components improve performance monotonically, though there are some surprises.
- The performance improvement is not always linear: the effect of working set size in memory hierarchy measurement is one example.
- Some axes are independent, others are not. For example, memory hierarchy and array size are related, while the memory hierarchy and the datapath are independent.
- Some axes are related by an obvious relationship (e.g. array size and datapath performance), others have a non-linear relationship (e.g. memory hierarchy and array size).

We have several tasks: create an ordering and characterization of datapath designs, create an ordering and characterization of memory hierarchy designs, combine them with the array size, and combine all three.

Looking at datapath alone is a non-trivial task. We have reduced the parameters of the datapath to the three that dominate performance: ALU/datapath width, multiplier/divider support, and floating point support.

Multiplier/divider support is given in terms of the size of the multiplier circuit, if any. This is an approximation since it does not include the divider or other ways of implementing multiplication. However we believe it is reasonable since i) it is relatively common for complex PEs to contain multipliers but not dividers, and, more importantly, since ii) there are numerous good multiplication-based division algorithms. This is especially true for less complex datapaths since small multipliers make a great deal of sense, while small dividers do not.

Floating point support is given in terms of number of PEs sharing a floating point unit. Clearly, this is not the way every SIMD array implements floating point, but is certainly valid as a way of characterizing relative floating point support. We make the further assumption that floating point support includes fixed point multipliers/dividers and that these units are available for the integer operations.

We are left with the following parameters:

- ALU/datapath $\rightarrow$ 1, 2, 4, 8, 16, 32
- Multiplier $\rightarrow$ 4, 8, 16, 32
- FP Support $\rightarrow$ 32 bit FP per 32, 16, 8, 4, 2, 1 PEs

We combine them into classes of similar complexity as shown in Table 1. As our studies of datapath complexity continue, we expect this list to be refined.

The two primary components of memory hierarchy are the register file size, which determines the number of loads and stores, and the cache size, which determines the latency of the loads and stores. The register file size is varied simply along a range from 20 to 200 bytes per PE. The cache size is more problematic.

Although it is easy to vary the cache size in the same way we varied the register file size, it makes much less sense. One reason is that the cache is an off-chip system component and therefore not as closely involved in chip area trade-offs as the register file. Another reason is that it is unlikely that a system will be built with PE cache if that cache has not been designed to have a hit rate of over 90%. We therefore use two parallel rankings: register file size with no cache, and register file size with cache. Cache parameters required for this performance have been presented elsewhere [8].

5 Experimental Results

We have examined seven issues. Two involve the datapath, two memory, one memory with respect to datapath, and two overall cost-performance.

1. *As more resources are applied to the datapath, what features should be given priority?*
Performance was measured for each application-datapath-array size combination. Execution time was plotted versus datapath complexity with one graph per application and one line for each array size. The results are shown in Figure 6. The models referred to on the X axis are shown in Table 1. The key results are as follows:

- In the two floating point applications, having floating point support makes a huge difference.
- For the non-floating point applications, very little speedup is achieved by increasing the datapath beyond 8. In the floating point applications, 16 bit ALUs appear to be sufficient.
- Multipliers are very useful: this is especially apparent in Prewitt where a four bit datapath with a four bit multiplier gives better performance than an 8 bit datapath which must execute multiplies in software.

2. *How does increasing the array size affect the datapath component of the performance?*
Results from experiment 1 establish the basic relationship between datapath and performance. Here we see find the cost of VPE emulation to the datapath. The VPR is inversely proportional to the array size. In these experiments, 256x256 has a VPR of 1 and 64x64 has a VPR of 16. Recall that VPR physical instructions are required for each arithmetic or logical VPE instruction, but that increased overhead is required to emulate NEWS communication.

Execution time was plotted versus array size with one graph per application and one line for each datapath design (see Figure 7). We find that the overhead beyond simple emulation is significant: for the IU Benchmark, going from a VPR of 1 to 2 causes a slowdown of a factor of nearly 4. Succeeding jumps triple the slowdown. Spiral3, which also has significant communication, has correspondingly large overhead.

3. *For a given array size, how big should the register file size be?*
Recall that for VPE emulation, the working set size is increased, but that this increase is non-linear due to instruction scheduling. Execution time was plotted against register file size with one graph per array size and one line per application. See Figure 8. The results are surprisingly consistent: for 256x256, 128x256, and 128x128 arrays, a register file size of about 60 bytes seems to be indicated. For 64x128 arrays, a register file size of about 100 bytes appears appropriate. For 64x64 arrays, the working set may be too large to fit in a reasonable sized register file and caching may be required.

4. *How does each application's working set vary with array size?*

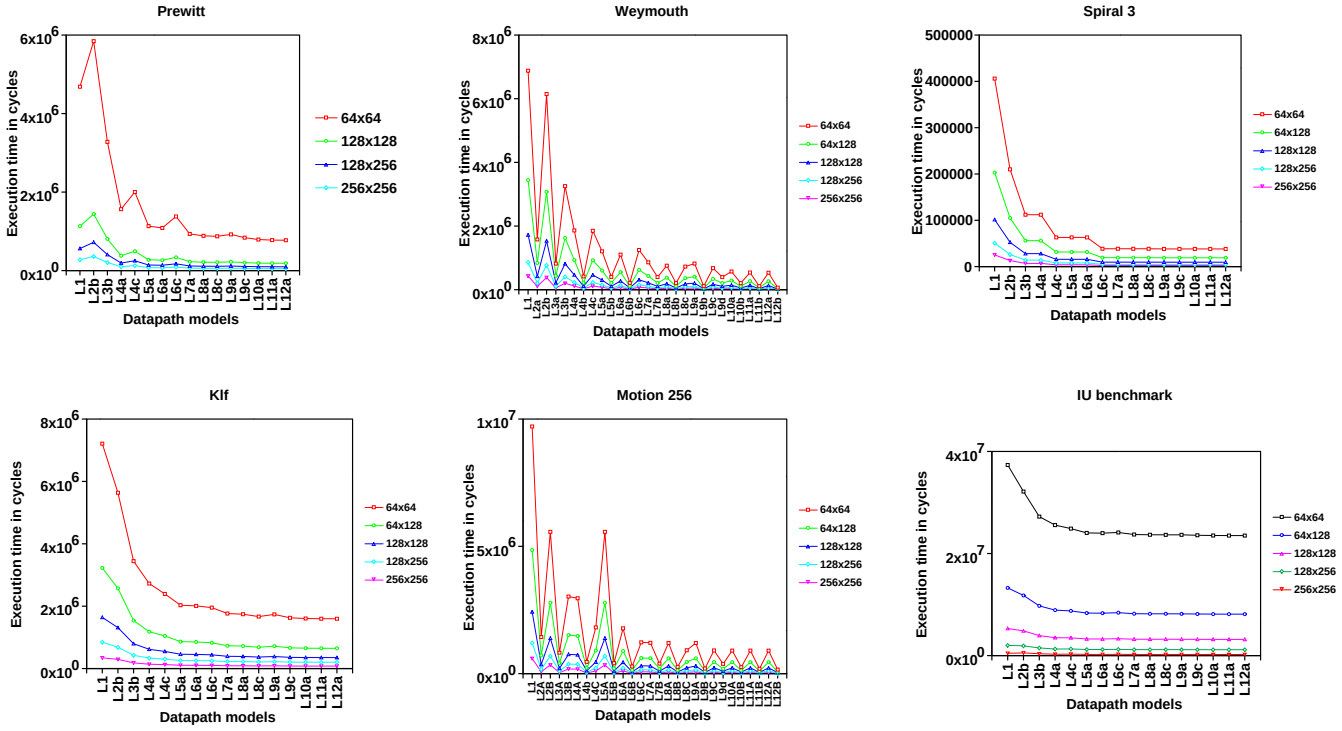

Figure 6: Experiment 1—execution time is plotted versus datapath complexity (see Table 1) with one graph per application and one line for each array size.

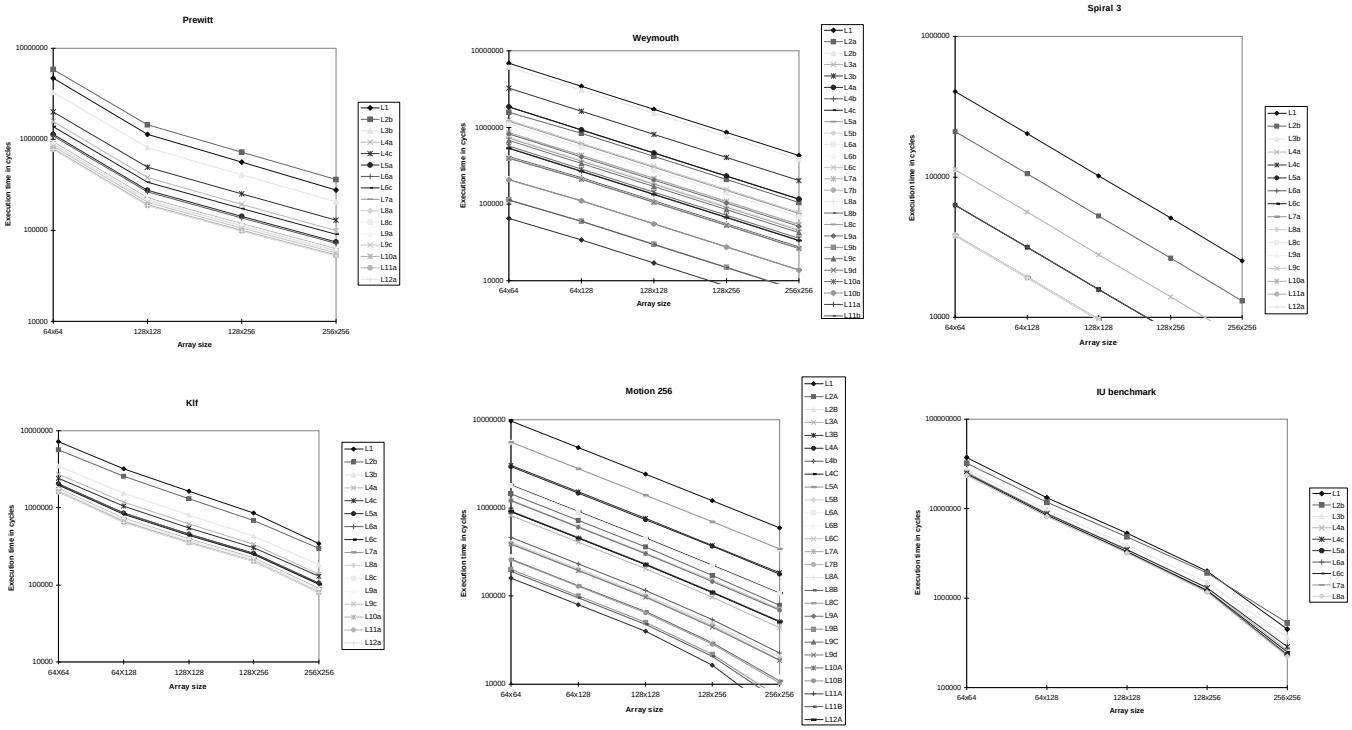

Figure 7: Experiment 2—execution time is plotted versus array size with one graph per application and one line for each datapath design

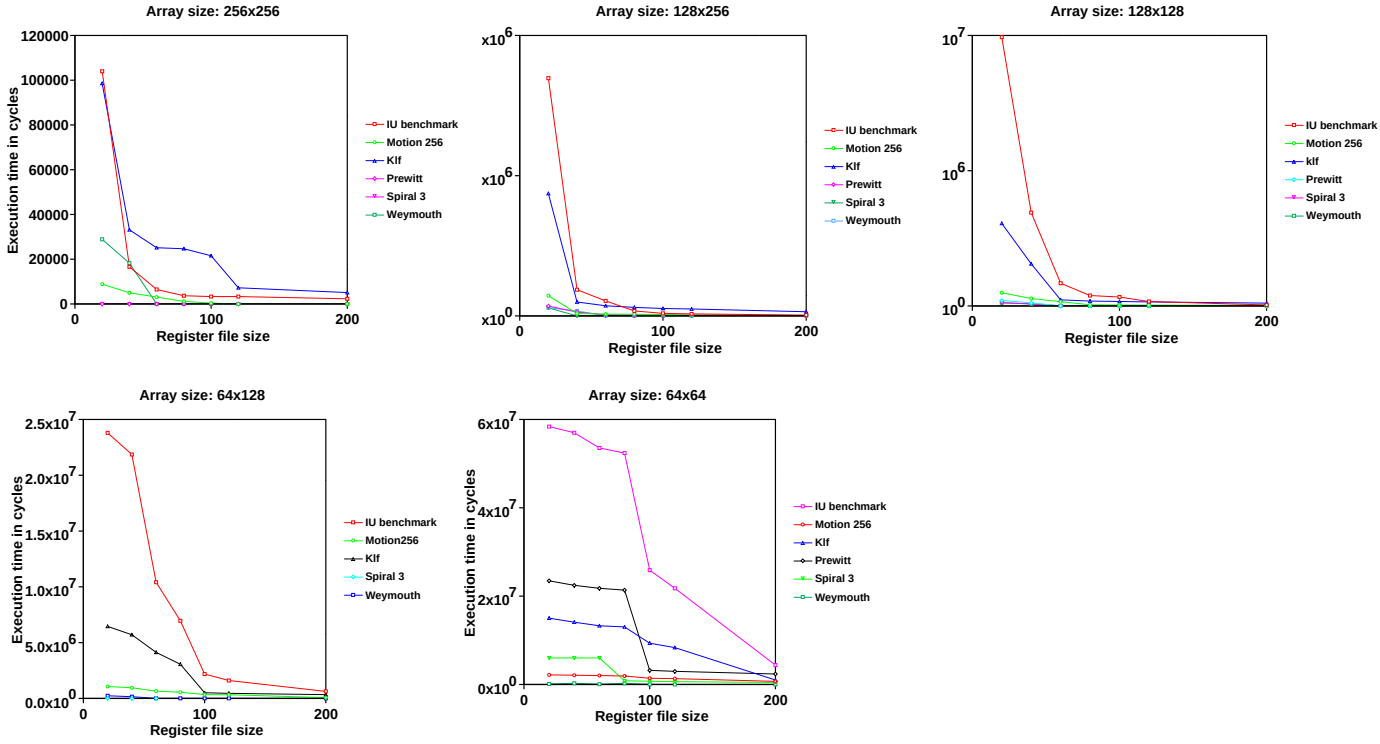

Figure 8: Experiment 3—execution time was plotted against register file size in bytes with one graph per array size and one line per application.

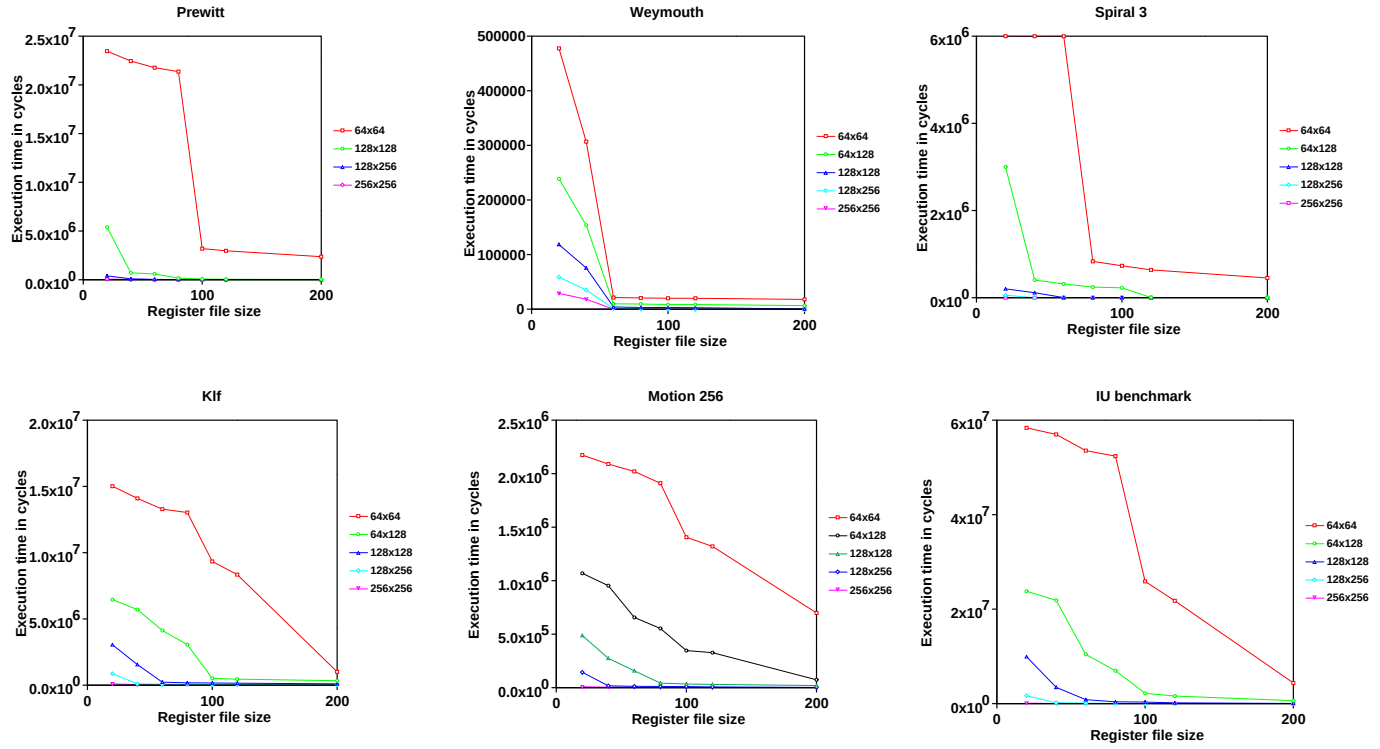

Figure 9: Experiment 4—execution time is plotted against register file size in bytes with one graph per application and one line per array size.

These are the same results as in 3. but plotted by application rather than array size. See Figure 9. Again, if VPE emulation were as simple as repeating each instruction VPR times, then we would expect the working apparent working set of each application to rise proportionally with the VPR. However, that is not the case: rather, VPE emulation is carried out by having each physical PE emulate a VPE *for as long as possible*. The result is seen as, e.g., the working set size for the IU Benchmark appears to go roughly from 40 to 60 to 100 to 150 instead of doubling each time.

5. *How should resources be partitioned between memory and datapath to achieve system balance?*

The assumption made here is that to the first order, good system design presupposes the removal of bottlenecks, or equivalently, good system balance. To find systems that meet these criteria, we examined performance versus both datapath complexity and register file size for each application and array size. The result was a set of 3D graphs with execution time on the Z axis and datapath complexity and register file size on the other two axes. From these graphs, points were extracted where memory references contributed 10% of the total cycles, the assumption being that in that case memory accesses would not be considered a bottleneck. For designs where memory references contribute significantly, enhancements such as PE cache might be considered.

These data are illustrated in a plot of register file size versus datapath model shown in Figure 10. For each level of datapath complexity in each application, only the best design was selected. Some observations are as follows:

- The expected result—that as VPR increases the importance of the memory hierarchy increases as well—is indeed borne out.
- Also as expected, as the datapath complexity increases, the amount of register file required to maintain system balance usually increases as well.
- In situations where increased datapath complexity does not significantly improve performance, the graphs are flat.
- The volatility in the Weymouth (and to a lesser extent Motion) graph appears to be a sensitivity issue: the surface is nearly flat and small performance differences can result in large movements in the isobars.
- Graphs where the larger arrays 'fall off the bottom' and smaller arrays 'burst out the top' (e.g., Spiral has both) indicate situations where cache is never or always needed, respectively.

6. *For each array size, which designs give the best performance per chip area?*

For experiments 6 and 7 we have estimated PE area by synthesizing components such as register files, ALUs, and multipliers of various sizes. Since there were problems in synthesizing directly to the gate level, the area is given in terms FPGA cells. Work continues here and obviously these results should only be used for grossest comparison.

Figure 12 shows execution time versus estimated chip area for various array sizes for the IU Benchmark. Only the most promising datapath- register file combinations are plotted. Figure 11 shows a plot where all possible datapath-register file combinations are included.

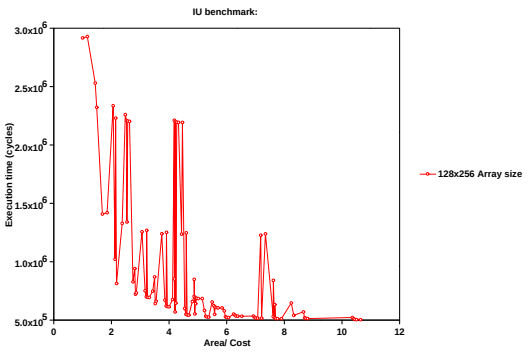

Figure 11: Experiment 6—shown is a plot of chip area versus IU Benchmark execution time for all 128 x 256 PE designs described here.

array size	datapath model	reg. file size	Est. chip area (000s)	Execution time (000s)
64x64	4a	20	1417	67457
64x64	2b	100	3629	41181
64x64	4a	100	4039	34950
64x128	2b	60	4637	16703
128x128	2b	40	6652	6256
64x128	4a	100	8077	5701
128x128	2b	60	9273	3646
128x128	4a	80	13531	1892
128x256	4a	40	16580	814
128x256	5a	40	20054	722
128x256	5a	100	35783	553
256x256	5a	40	40108	165

Table 2: This table depicts the most promising array designs in terms of performance-chip area tradeoff for the IU Benchmark application. The data are shown graphically in Figure 13.

7. *Normalizing for array size, which designs give the best performance per chip area?*

Here the most promising designs in terms of cost-performance have been selected from Figure 12, normalized for array size, and plotted in Figure 13. The points are described further in Table 2. Since only the IU Benchmark has been plotted so far, none of the datapath selections contained floating point support.

6 Conclusion

We have run numerous experiments and derived results relating to datapath, support by the datapath of VPE emulation, working set sizes and how they relate to VPE emulation, points in the memory hierarchy/datapath design space at which system balance is roughly achieved, and, for one application, determined the most promising designs in terms of performance-chip area ratio. Further work continues in automatic synthesis of these designs for the more accurate determination of actual hardware cost.

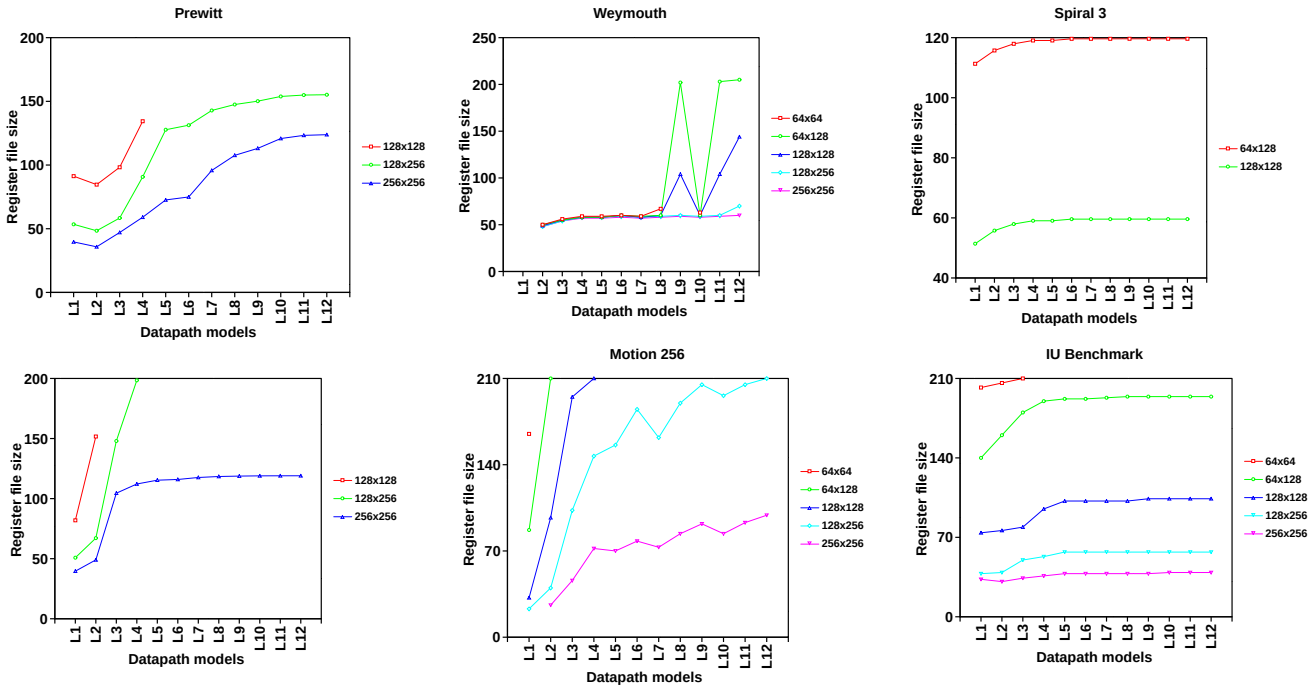

Figure 10: Experiment 5—each line represents the limit in register file size (in bytes) above which adding cache is not likely to significantly improve performance. One line is plotted per array size with one graph per application.

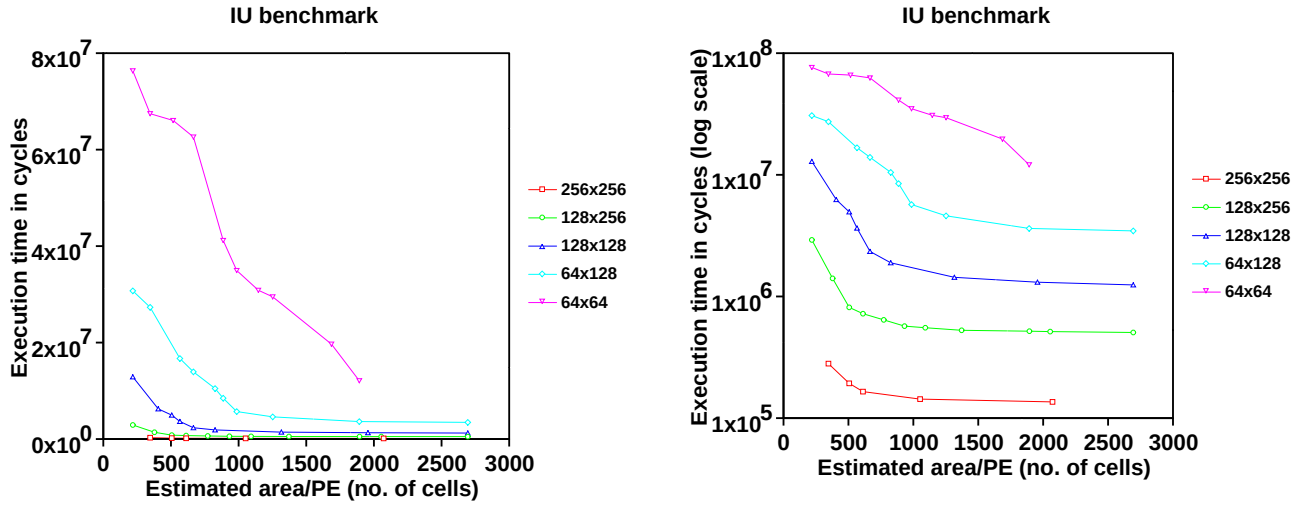

Figure 12: Experiment 6—shown are plots of chip area versus IU Benchmark execution time for selected PE designs of various array sizes.

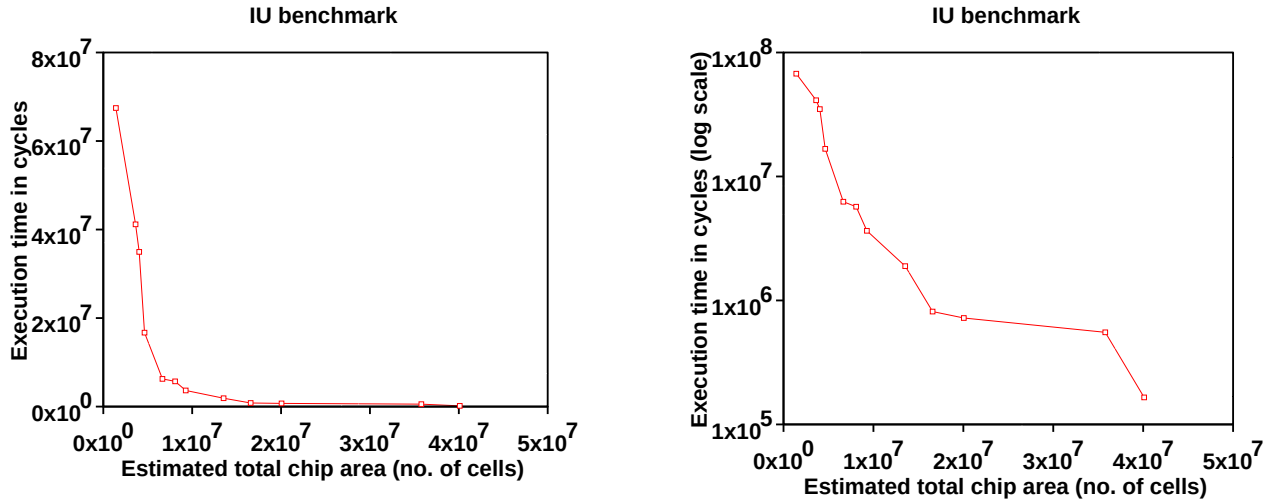

Figure 13: Experiment 7—shown are plots of chip area (normalized for array size) versus IU Benchmark execution time for selected PE designs. The data points are described in Table 2.

References

[1] Blank, T. The MasPar MP-1 architecture. In *Proc. 35th IEEE Comp. Conf.* (1990), pp. 20–24.

[2] Burrill, J. H. *The Class Library for the IUA: Tutorial*. Amerinex Artificial Intelligence, Inc., Amherst, MA 01003, 1992.

[3] Herbordt, M. C. *The Evaluation of Massively Parallel Array Architectures*. PhD thesis, Dept. of Comp. Sci., U. of Mass. (also TR95-07), 1994.

[4] Herbordt, M. C., Burrill, J. H., and Weems, C. C. Making a dataparallel language portable for massively parallel array computers. In *Proc. of Computer Architectures for Machine Perception* (1997).

[5] Herbordt, M. C., Corbett, J. C., Spalding, J., and Weems, C. C. Practical algorithms for online routing on fixed and reconfigurable meshes. *J. Par. Dist. Comp. 20*, 3 (1994), 341–356.

[6] Herbordt, M. C., Kidwai, O., and Weems, C. C. Preprotyping simd coprocessors using virtual machine emulation and trace compilation. *Performance Evaluation Review 25*, 1 (1997), 88–99.

[7] Herbordt, M. C., and Weems, C. C. An environment for evaluating architectures for spatially mapped computation: System architecture and initial results. In *Proc. of Computer Architectures for Machine Perception* (1993).

[8] Herbordt, M. C., and Weems, C. C. Experimental analysis of some SIMD array memory hierarchies. In *Proc. of the 1995 Int. Conf. on Parallel Processing* (1995), vol. 1: Architecture, pp. 210–214.

[9] Owens, R. M., Irwin, M. J., Nagendra, C., and Bajwa, R. S. Computer vision on the MGAP. In *Proc. Comp. Arch. for Machine Perception* (1993), pp. 337–341.

[10] Preston, K. The Abington Cross benchmark survey. *IEEE Computer 22*, 7 (1989), 9–18.

[11] Rosenfeld, A. A report on the DARPA Image Understanding Architectures Workshop. In *Proc. Image Understanding Workshop* (1987), pp. 298–301.

[12] Systems Performance Evaluation Cooperative. *SPEC Newsletter: Benchmark Results*. Waterside Associates, Freemont, CA, 1990.

[13] Weems, C. C., Riseman, E. M., Hanson, A. R., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *JPDC 11* (1991).