

Retargeting Image-Processing Algorithms to Varying Processor Grain Sizes

Samuel T. Sander
Clemson University
samsander@ieee.org

Linda M. Wills
Georgia Institute of Technology
linda.wills@ece.gatech.edu

Abstract

Embedded computing architectures can be designed to meet a variety of application specific requirements. However, optimized hardware can require compiler support to realize the potential of the hardware. This is especially true for embedded image processing systems where significant architectural variation is possible, and targeted software can change drastically based on architectural variation. This paper presents methods to compile a single high-level source given a fundamental variation in data-parallel target architectures – processor granularity ranging from a single processor to a massively parallel processor array. The approach uses single PPE virtualization, which supports pixel-level data-parallel expressions that operate on a virtual one pixel per processing element (PPE) network and applies pixel-locating transformations to retarget the code into a given target PPE. Unlike mainstream parallel computing techniques, this technique can be applied to lightweight SIMD targets that do not provide global communication hardware or shared memory.

1. Introduction

For applications such as image processing, data-parallel architectures provide high efficiency, and this has led to a wide range of SIMD usage: from SIMD arrays integrated into memory chips [16] to SIMD arrays exploiting subword parallelism in multimedia instruction set enhancements [17, 21, 22], as well as classic SIMD general purpose processors [6] and commercial SIMD arrays for embedded image processing [27]. Although data-parallel architectures provide efficient high-performance computing, software development for these architectures is challenging, and optimization often requires laborious target-specific hand-coding [9]. Current compilation techniques sacrifice portability or performance.

To allow portability to varied data-parallel architectures, the programming specification must be

architecture independent, especially with respect to granularity. In image-processing applications, the grain size of the processing elements determines the number of pixels that are mapped to each processing element, which is called the pixels per processing element (PPE) ratio. The PPE ratio determines how a pixel is accessed relative to another pixel. Varying the PPE ratio changes how a particular pixel location is determined relative to other pixels and how its value is retrieved (e.g., from local memory or via communication from other processors). Because sequential programming languages obscure parallelism, it is difficult to take a sequential specification and automatically extract uniform data-level parallelism that can be mapped onto a SIMD multiprocessor network. Consequently, a data-parallel algorithm must be used in the application specification when targeting these architectures.

The approach developed in this research is to use single PPE virtualization. In this technique, pixel-level data-parallel expressions operate on a virtual 1 PPE processor network. Pixel-locating transformations are then used to retarget the code into the target PPE given the processor network size and the inter-processor communication capabilities of the target system.

A research compiler has been developed using these techniques, and performance measurements were made for a range of PPE ratios in terms of execution time and generated code size [26]. First, the research compiler code is compared with handwritten assembly code (using the SIMPil architecture [10, 24]) for different configurations of a SIMD network representing a range of PPE values. Second, to evaluate the performance at the other end of the granularity spectrum (sequential processing), a comparison is made between the research compiler and a standard C compiler for a general-purpose processor (using SimpleScalar [1]). Together, these two studies demonstrate retargeting the same PPE-independent specification to processors of varying grain sizes with only a small increase in execution time when compared with PPE-dependent source programs.

2. Related research

When targeting multiprocessor SIMD architectures, current compilation techniques require architecture-dependent explicit parallelism, sacrifice performance, and/or are incompatible with newer architectures (e.g. focal plane arrays [8]). As an example of explicit parallelism, High Performance Fortran includes data layout specifications to help the compiler divide loop iterations between processors to generate efficient code for a specific number of processors, which limits software portability [14]. This inability to produce code with portable performance was shown in a performance comparison with the ZPL compiler [18, 20]. The ZPL language is an architecture-independent parallel language developed at the University of Washington [5, 18]. ZPL compilation techniques assume that the underlying reference architecture is MIMD [5]; however, ZPL's region concept and the treatment of boundary conditions provide elegant high-level expression that could facilitate efficient target code for SIMD architectures [3, 4]. Consequently, ZPL was selected as the source language for the research in this paper. There are other MIMD languages (MPI [19]) and data-parallel languages and associated compilers (Modula-2* [23], C* [25], APL [12], etc.), but the code generation techniques are either not compatible with SIMD architectures or are not able to target the range of processor granularities considered in this research and produce efficient code.

The most significant shortcoming in the few existing architecture-independent SIMD compilers in existence is the handling of grain size, or the amount of data that is processed by each processor. Depending on the dataset and the number of processors available, a retargeting compiler should be able to produce target code that provides efficient use of available resources. A common method for handling grain size for SIMD compilation is tiling [12, 13, 15]. In some applications the dataset can be broken up into subsets, called tiles, each having the same number of data elements as the number of processors in the system (PPE equal to one); see Figure 1 (top) [13]. Each of the tiles is then sequentially processed. This method effectively maintains a single pixel per processor ratio independent of problem size, which greatly simplifies software development. However, this approach is often inefficient or incompatible with a given application. For example, computing the edge values of the tiles requires overlap in the tiles for distributed memory systems. Also, inter-processor communication is increased because of the spatial locality that image-processing algorithms typically exhibit. Additionally,

tiling is simply not an option in some architectures, such as focal plane arrays [8].

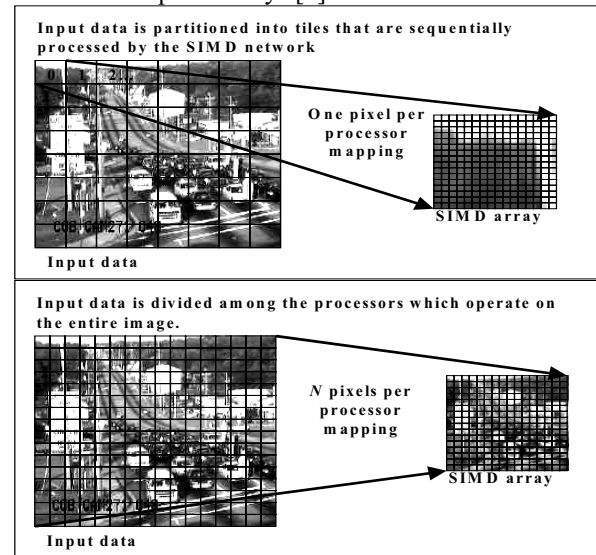


Figure 1. Mapping data to processors: single PPE (top) and multiple PPE (bottom).

A second method for handling a larger dataset than the processor array is to divide the entire image among the processing elements (PEs) in the system (see Figure 1 (bottom)). A sequential architecture can be seen as one extreme of this approach in which the entire image is mapped to a single processor. This pixel-to-processor mapping minimizes inter-processor communication and allows for algorithm optimizations within the pixels assigned to a processor [23]. However, this method changes the PPE ratio, which causes drastic changes in the instructions necessary to complete an operation.

3. Virtualization and retargeting

This section describes targeting uniform data parallel expressions to data parallel processor networks of arbitrary size. Pixel locating transformations are used to scale a program written assuming pixel-level parallelism (PPE equal to one) to another PPE ratio. This method uses relative pixel specifications to identify the processor and offset of the selected pixel. Based on the PPE and the inter-PE communication mechanism, the necessary code can be generated to seamlessly execute pixel-level data-parallel operations.

As the number of pixels per processing element (PPE) varies, the instructions necessary to complete a desired operation change. For each high-level data-parallel operation, every processing element must loop through the pixels assigned to it and incrementally execute the high-level operation. As the PPE varies,

the means of accessing pixels changes from determining the correct pixel index in the current PE to determining and reading the correct pixel index in a neighboring PE or even in a PE several network hops away.

A high-level relative pixel location specification is used to express pixel-level data-parallel algorithms. The pixel location specification is of the form $P@[dr,dc]$, where dr and dc can be any integer: dr represents the row offset relative to an arbitrary pixel, and dc represents the respective column offset. Positive column direction is to the right, and positive row direction is down. For example, $P@[-1,-1]$ refers to the pixel immediately to the upper left of every pixel, and the statement $P = P@[-1,-1]$ would assign every pixel the value of its upper left corner neighbor (effectively shifting the entire image down and to the right one pixel). If the PPE ratio were 1, then the calculations would be trivial. The pixel index is always 0, and the desired pixel is dr PE rows away and dc PE columns away. At 4 PPE, the desired pixel would be either within the current PE, in the PE to the left, in the PE above, or in the PE to the upper left as shown in Figure 2.

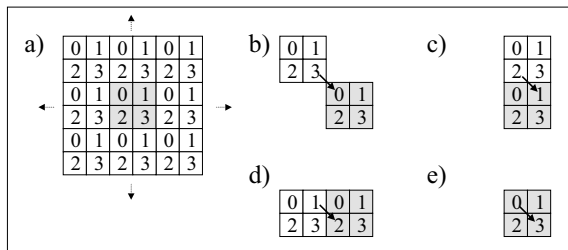


Figure 2. Pixel locations for a PPE ratio of 4
a) Pixel mapping, b, c, d and e) $P@[-1,-1]$ relative to pixel 0, 1, 2, and 3 (respectively).

Given a pixel index and a relative pixel specification, the location of the target pixel can be determined in terms of the horizontal and vertical distance of the PE and the pixel index within that PE. The steps involved in computing the target pixel location are shown in Figure 3. The numbers used in Figure 3 are for a specific example: locating $P@[-1,-1]$ at 4 PPE for the pixel in the upper right corner of every PE. The necessary offset calculations can be performed in the compiler or at runtime on the target. Aside from the offset calculations, only inter-PE transfer instructions are needed. A compile time algorithm for computing the pixel offset and generating the communication instructions takes the relative offsets and the current pixel index and produces the instructions necessary to put the specified pixel into a register. For lower PPE values, compile-

time location is used and only the load and transfer (inter-PE communication) instructions are executed at runtime resulting in between four and six instructions for each pixel.

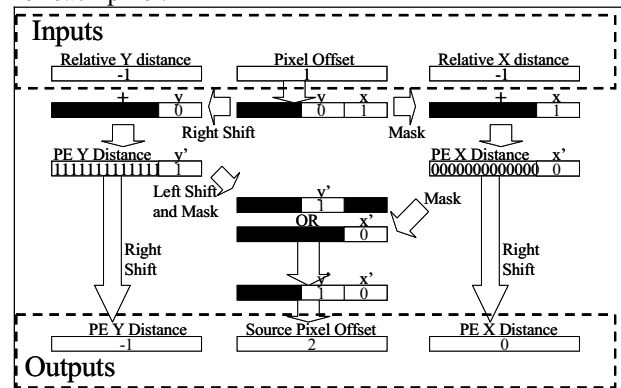


Figure 3. Pixel location example:
 $P@[-1,-1]$ at 4 PPE for pixel 1.

```

;po          :Pixel offset (R0)
;po_x (_y)   :x (y) po component
;rel_x_dx (_y_) :x (y) distance (R1(R7))
;mask_x (_y)  :1's in x (y) component
;width_x (_y) :# of bits in mask_x (_y)
;vrtl_x (_y)  :virtual x (y) offset
;src_x (_y)   :source x (y) offset
;src_po       :source pixel offset
;pe_y_dx (_y_) :PE X and Y Distance

andi R2, R0, 3 ; po_x = po AND mask_x
lshi R3, R0, -2 ; po_y = po >> width_x
add R4, R2, R1 ; vrtl_x=po_x + rel_x_dx
add R2, R3, R7 ; vrtl_y=po_y+rel_y_dx
andi R3, R4, 3 ; src_x=vrtl_x AND mask_x
andi R5, R2, 3 ; src_y=vrtl_y AND mask_y
lshi R6, R5, 2 ; src_y=src_y << width_y
or R5, R6, R3 ; src_po=src_x OR src_y
ashl R3, R2, -2 ; pe_y_dx=vrtl_y>>width_y
ashl R2, R4, -2 ; pe_x_dx=vrtl_x>>width_x

```

Figure 4. Runtime pixel location of $P@[r,c]$ for 16 PPE on SIMPIL using SIMPIL Assembly.

The compile-time method produces highly efficient code, but at 4,096 PPE this would produce about 100 kilobytes of code per operation. A runtime version of the location logic should be used for larger PPE ratios depending on the importance of code size. The runtime code generation method must generate the necessary instructions to perform the offset calculations in addition to generating conditional loops containing the inter-PE transfer instructions. In the runtime version each PE loops through the pixels and must locate and retrieve the necessary pixels relative to each pixel. A sample of runtime pixel location is given in Figure 4 (SIMPIL Assembly), which follows the calculations

shown in Figure 3. Since the PPE and other details are known by the compiler, the target code can be optimized based on the operation and the target architecture.

3.1. Optimization

To produce efficient code, the available information for each pixel location operation must be used in conjunction with the desired compilation goals. In cases where the runtime version is used, any knowledge of the direction specifiers can be used to improve performance. Specifically, knowing the direction and magnitude of the remote value allows simplification of the process. For $P@[0,0]$, no location instructions are needed. For $P@[x,0]$ and $P@[0,x]$, only one of the directions must be calculated. If the magnitude of a direction is known at compile-time then the maximum number of processor hops is also known, and a transfer loop in that dimension may be avoided. If the sign of the direction is known at compile-time, then the direction of any transfer is also known, and the runtime code can have a simplified transfer loop and will not have a sign check. Additionally, in a single processor system only the pixel offset must be calculated, but the PE distance result can be used for detection of border conditions.

3.2. Border conditions

Pixel location logic also allows for the handling of border conditions with little overhead. In the ZPL language, the border condition is specified as “east of region = 0;” [18]. In other languages the border conditions must be explicitly tested in the code, which is not as elegant for the high-level source, and it can be difficult to express in data-parallel terms. Alternatively, using the same high-level specification as ZPL, the border conditions are implemented by altering the pixel location code. The border condition logic is inserted when the transfer instructions are generated. In the absence of a border specification, or if the expanded data array method is used, then the extra instructions are omitted, and the border region becomes a “don’t care” region. The border logic can be implemented by increasing the dimensions of the data and initializing border values, or it can be implemented using additional conditional logic during pixel location. Either method is compatible with single PPE virtualization and pixel location. The first method is the fastest (unless a large border is needed), but it may not be compatible with focal-plane architectures.

3.3. Conditionals

PE-level conditionals are treated specially in SIMD architectures, since all of the processors execute the same instruction for every clock cycle. There are primarily two methods for achieving PE-level conditionals: masking and disabling. Disabling requires hardware support, and works by conditionally instructing a PE to ignore instructions (i.e., sleep) until a wakeup instruction is received. Masking requires minimal hardware support, and it works by constructing conditional masks with either all ones or all zeros. In masking, a conditional operation like the one below is translated to a SIMD compatible construct. For example “if (condition) then $A:=B;$ ” is interpreted as “mask:=condition * 0xFFFF; $A:=(B \text{ And } \text{mask}) \text{ Or } (A \text{ And } (\text{Not } \text{mask}))$;”.

Both techniques also apply to loop statements with the exception that the terminating condition must be met by all PEs. This can be checked using a single instruction if hardware support is provided; otherwise, it requires a costly software reduction operation.

When multiple pixels are stored in each PE, pixel-level conditionals become even more challenging. The method described in this paper is compatible with pixel-level conditionals using either of the PE conditional methods described.

This method maintains a Boolean variable for each runtime (pixel, conditional) combination that is true if a pixel has not met the execution condition for a given conditional. For each pixel, an assignment operation is executed only if all of the current conditions are met. Non-assignment instructions may be required to compute addresses for data needed by another pixel and are allowed to execute unconditionally. The basic steps are as follows, broken down by their application time.

- At the beginning of program execution:
 - ♦ Within a PE, each pixel, i , is assigned a bit vector ($C_i[]$) to store conditional Booleans
 - ♦ Vector C_i is initialized to False
 - ♦ The controller maintains a global value, last conditional identifier (C_ID_L)
 - ♦ C_ID_L is initialized to zero
- At the beginning of conditional statement j :
 - ♦ Controller increments C_ID_L
 - ♦ A temporary variable (C_ID_j) is allocated for this conditional and assigned C_ID_L
 - ♦ For each pixel i , $C_i[C_ID_j] := \text{Not}(\text{condition})$
- During an assignment operation:
 - ♦ For each pixel i , execute only if the vector C_i contains no true values
- At the end of conditional statement j :

- ♦ $C_i[C_ID_j] := 0$
- ♦ C_ID_L is decremented

The multiple Boolean operations required can be efficiently implemented using bitwise operations. This method handles an arbitrary number of nested conditionals; however, for performance the number of nested conditionals can be limited to the number of bits in a processor word.

4. Validation and evaluation

A research compiler has been developed targeting modified ZPL source [18] to SIMPil [10, 24] and SimpleScalar [1] using the techniques described in this paper. This section describes the research compiler and the evaluation of its performance.

4.1. Compiler description

A block diagram of the research compiler is shown in Figure 5. The modified ZPL source language is parsed into an abstract syntax tree and a symbol table. An intermediate representation is generated based on the number of processors, inter-processor communication, register width, and the size of the image. Register allocation is then performed on the intermediate code using Chaitin's graph-coloring algorithm [2]. Retargeting is performed in two steps. In the first step, an intermediate representation is generated that contains the basic data layout and control flow required for the target architecture (e.g. the number of pixels assigned to each processor). The final code generation step has specific knowledge of the target architecture in terms of register usage and other instruction-set details. Currently, the compiler produces assembly code output for the two target architecture types.

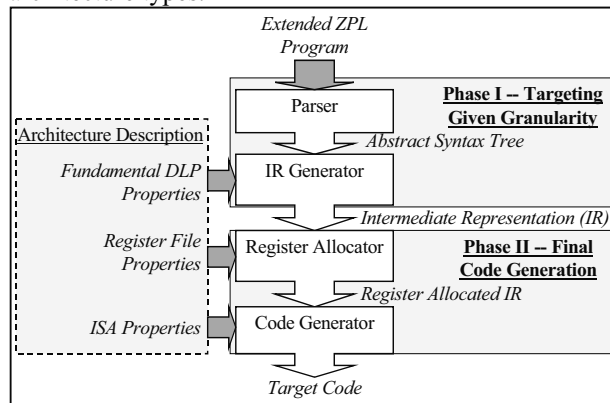


Figure 5. Block diagram of research compiler.

4.2. Target architectures

The techniques described in this paper can be used on any processor network with at least one processor and a means of inter-processor communication for systems with more than one processor. To show that the same high-level source can be retargeted to widely different architectures with varying PPE values, performance comparisons were made from simulation results using both a SIMD array target architecture and a general-purpose processor target architecture in terms of execution time and generated code size. Architectures with fully developed simulation and analysis tools were chosen for the target architectures of the compiler. This provides easy access to validation and performance analysis. Additionally, the simulators are not dependent on implementation technology and can be modified to assess the impact of architectural changes.

The first target architecture for the research compiler is SIMPil. The SIMD Pixel Processor (SIMPil) architecture is a portable, single-chip, focal-plane SIMD processor developed by the Portable Image Computation Architecture Group (PICA) at the Georgia Institute of Technology [8, 24]. SIMPil provides multimedia supercomputing in a portable package. The SIMPil architecture is capable of executing over 1000 GOPS (billions of operations per second). In SIMPil each PE is equipped with a 4x4 photo-detector array, which samples live images and stores the data in memory. The PEs communicate using a NEWS network, and each PE has its own local memory and registers.

The infrastructure of tools and applications that are available for SIMPil facilitated the evaluation of the research compiler. First, a fully functional software simulator has been developed by the PICA group [11]. This simulator allows architectural features like PPE ratio and memory size to be changed, which facilitate testing the functionality and performance of the retargeting techniques. When targeting SIMPil, the number of pixels per processing element (PPE) was varied, and values of 1, 4, 16, 64, 256, and 1024 PPE were used.

Furthermore, many image-processing algorithms have been implemented (hand coded) in the SIMPil assembly language, and these implementations serve as a basis of comparison for determining the effectiveness of the techniques. SIMPil Assembly is targeted for use with the SIMPil simulator.

The second target architecture for the research compiler is SimpleScalar. This target serves as a representative general-purpose architecture. The SimpleScalar tool set from SimpleScalar LLC is a

software infrastructure that includes simulation, analysis, and other tools [1]. SimpleScalar has a configurable architecture, and in this research the PISA instruction set configuration of the SimpleScalar simulators is used on a big-endian host system. The research compiler produces PISA assembly output, and “AS,” the GNU assembler is used to assemble the SimpleScalar target code [7].

For the SimpleScalar target, C applications compiled using the GCC compiler with optimization (O4) [7] are used for comparison with the output of the research compiler. For SimpleScalar, both the “C” source and the ZPL source used the same input and output format: a bitmapped image file.

4.3. Metrics

The effectiveness of the techniques is measured by comparing instructions executed and code size of the output produced by the research compiler with code produced by hand-targeted coding. For SIMPIL and SimpleScalar the total number of instructions executed is used for the main performance metric. However, in SIMPIL an instruction execution represents an instruction execution at every processing element. For SIMPIL the static number of instructions in the assembly code listing is used as the code size. For SimpleScalar, the resulting binary executable is used as the code size.

A suite of image-processing applications, including two-dimensional convolution, region magnification, and vector quantization, was compiled and simulated on each of the target configurations. Region magnification and vector quantization are small programs that perform these respective image processor operations.

The evaluation focuses on granularity changes by comparing performance for targets with a different number of pixels per processor for each target configuration. Consequently, the results are independent of the image size. The compiler and the simulators support arbitrary image sizes, and during evaluation a small 64x64 test image is used to minimize simulation time.

Together, these performance studies test the ability to retarget the same PPE-independent specification to processors of varying granularity as compared with the performance of programs produced using existing architecture-specific programs (C programs are considered architecture-specific because their sequential expression makes them incompatible with data-parallel architectures).

5. Results

The graph in Figure 6 shows the relative executed instruction count versus the PPE ratio of the target architecture. The relative executed instruction count is determined by dividing the number of executed instructions when using the research compiler by the number of executed instructions when using the alternative method (assembly code for SIMPIL and compiled C for SimpleScalar). The overall average efficiency of the research compiler is 78%. Only region magnification had efficiency below 70%. Although some optimization was implemented in the research compiler, optimizations were not necessary to accomplish the retargeting goals of the compiler. Implementing further optimizations is a subject for future work.

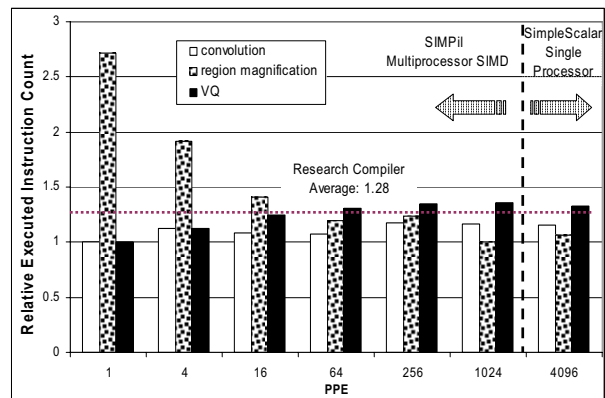


Figure 6. Relative executed instruction count.

For convolution, the compiled code is on average 90% as fast as the hand-coded and C-coded versions. The best case is 100% efficiency at one PPE, and the worst case is 85% efficiency at 256 PPE. This indicates that the compiler is successful at targeting variations in PPE with minimal loss of performance.

For region magnification, the research compiler is on average 66% as efficient as hand-coded assembly, ranging from 37% at one PPE to 100% at 1024 PPE. The efficiency for region magnification suffers at lower PPE values because the region magnification application requires a large amount of data-parallel conditional logic that has not been fully optimized in the research compiler.

The research compiler for vector quantization has an average of 80% efficiency, which is more efficient than for region magnification. The maximum efficiency of 100% is achieved at 1 PPE, and the minimum efficiency of 74% occurs at 1024 PPE. Contrary to region magnification, vector quantization has a higher efficiency at lower PPE and a lower

efficiency at higher PPE. This is because of a more complex random-access algorithm that the compiler is not equipped to optimize. At lower PPE values a simple data-parallel algorithm is used resulting in performance similar to hand coding.

The graph in Figure 7 shows the corresponding relative code size for the same code used to produce the data in Figure 9. Relative code size is the ratio of the code size for the research compiler and the alternate code source (assembly code for SIMPil and compiled C for SimpleScalar). On average the code size of the compiled code is 15% smaller than that of the hand-coded and compiled C applications. Together with the execution length results, these results indicate that the research compiler output code is comparable to architecture-specific hand-coded applications. For these applications, the research compiler is very successful at providing a small code size.

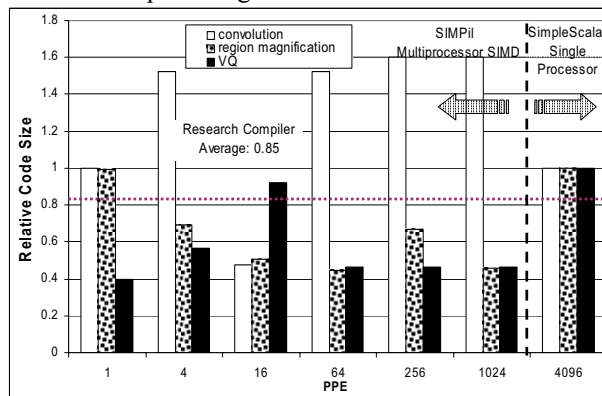


Figure 7. Relative code size comparison.

To measure usability in terms of source lines for SIMPil development, the number of statements for the hand-coded versions is equal to the number of statements in each hand-coded assembly program, which varies between 53 and 682 instructions for the six SIMPil configurations studied. To measure usability in terms of source lines for C code, the number of statements for the C code is equal to 73 statements not including declarations. This is compared with 18 statements (or 11 if declarations are excluded) in the ZPL source. More importantly, the single ZPL source replaces all seven of these architecture-specific programs.

These results indicate that the compiler is successful at targeting variations in PPE, and that the compiled code is comparable to architecture-specific coded applications. With additional optimizations, higher efficiencies are realistically attainable.

6. Future work

Current plans for improving the research compiler include implementing more of the ZPL grammar and adding additional goal-oriented optimization. Additional optimization is needed in the research compiler to improve performance and reduce code size. As stated earlier, the research compiler includes only limited optimization. In addition to adding standard compiler optimizations, support for goal-oriented optimization would be beneficial. This would allow specification in terms of fixed constraints and relative priority of throughput, code size, and energy usage.

7. Conclusion

In this paper, compilation techniques were presented that take a single high-level source and target it for data-parallel execution on a wide range of processor granularities from single processor systems to multiprocessor SIMD arrays with varying number of processing elements. The techniques developed in this research help remove software development obstacles for data-parallel multi-processor architectures, enabling future research and implementation of these architectures. These techniques greatly simplify the development and improve the portability of software for current, future, and exploratory configurations of specialized data-parallel architectures. Additionally, the high-level language used in this research, ZPL, has already been ported to MIMD targets [5], which, together with this research, provides the potential for wide-ranging portability for high-performance architectures using a single source language.

8. References

- [1] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," SimpleScalar LLC, Ann Arbor, MI, <http://www.simplescalar.com>.
- [2] G. J. Chaitin, "Register Allocation & Spilling Via Graph Coloring," *ACM Sigplan Notice*, Vol.17-6, June 1982.
- [3] B. L. Chamberlain, "The design and implementation of a region-based parallel language," *PhD dissertation*, Department of Computer Science and Engineering, University of Washington, Nov. 2001.
- [4] B. L. Chamberlain, S. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines," *Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.

<http://www.cs.washington.edu/research/projects/zpl/papers/abstracts/lcpc96.html>.

[5] B. L. Chamberlain, S. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: a machine independent programming language for parallel computers," *IEEE Transactions on Software Engineering*, Vol. 26, Issue 3, pp. 197–211, Mar. 2000.

[6] D. Dale, L. Grate, E. Rice, and R. Hughey, "The UCSC Kestrel General Purpose Parallel Processor", *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pp. 1243-1249, Las Vegas, Nevada, June 1999.

[7] Free Software Foundation Inc., GNU Compiler Collection (GCC), Boston, MA, <http://GCC.gnu.org/>.

[8] A. Gentile and S. Wills, "Portable Video Supercomputing," *IEEE Trans. on Comp.*, 53(8), pp. 960-973, Aug. 2004.

[9] A. Gentile, S. Sander, L. Wills and S. Wills, "The impact of grain size on the efficiency of embedded SIMD image processing architectures," *Journal of Parallel and Distributed Computing*, Vol. 64, No. 11, pp. 1318 – 1327, November 2004.

[10] A. Gentile, A. López-Lagunas, H. H. Cat, K. S. Chung, L. Codrescu, M. Deb, J. C. Eble III, and D. S. Wills, "The SIMD Pixel Processor (SIMPil): SIMPil16 Specifications," *Technical Report PICA-TR-1996-32*, PICA Research Group, Georgia Institute of Technology, Atlanta, Georgia, 1996.

[11] A. Gentile, H. H. Cat, and D. S. Wills, "The SIMD Pixel Processor (SIMPil): SIMPilSim Instruction-Level Simulator for Windows," *Technical Report PICA-TR-1997-18*, PICA Research Group, Georgia Institute of Technology, Atlanta, Georgia, 1997.

[12] R. Greenlaw and L. Snyder, "Achieving speedups for APL on an SIMD distributed memory machine," *International Journal of Parallel Programming*, Vol. 19(2), pp. 111-127, Apr. 1990.

[13] M. C. Herbordt, J. H. Burrill, and C. C. Weems, "Making a dataparallel language portable for massively parallel array computers," *Proceedings of the Fourth IEEE International Workshop on Computer Architecture for Machine Perception*, pp. 160 –169, 1997.

[14] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Jan. 1997, <http://www.crpc.rice.edu/HPFF/>.

[15] K. E. Iverson, *A Programming Language*, Wiley, New York, 1962.

[16] P.M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, E. Sha, "Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies", pp. 88-97, *6th Symp. Frontiers of Massively Parallel Computation*, Annapolis, MD, Mar 1996.

[17] R. Lee, "Subword Parallelism with MAX-2", *IEEE Micro*, Vol. 16 No. 4, pp 51-59, August 1996.

[18] C. Lin, *ZPL Language Reference Manual (DRAFT)*, Technical Report UW-CSE-TR 94-10-06, University of Washington, May, 1996.

[19] Message Passing Interface Forum "MPI-2: Extensions to the Message-Passing Interface," 1997, www.mpi-forum.org.

[20] T. Ngo, L. Snyder, and B. Chamberlain, "Portable performance of data parallel languages," *Proceedings of the 1997 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing*, Nov. 1997.

[21] H. Nguyen and L. John, "Exploiting SIMD Parallelism in DSP and Multimedia Algorithms using the AltiVec Technology," *Proc. Intl. Supercomputer Conference*, pp. 11-20, June 1999.

[22] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, pp. 25-38, Jan. 1997.

[23] M. Philippsen, "Automatic data distribution for nearest neighbor networks," *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 178-185, Oct. 19-21, 1992.

[24] Portable Image Computation Laboratory (PICA), SIMPil, The SIMD Pixel Processor, Georgia Institute of Technology, <http://www.ece.gatech.edu/research/pica/simpil/>.

[25] J. R. Rose and G. L. Steele Jr. "C*: An Extended C Language for Data Parallel Programming," *Proceedings Second International Conference on Supercomputing*, pp. 2-16, May 1987.

[26] S. Sander, "Retargetable Compilation for Variable-Grain, Data parallel Execution in Image Processing," PhD dissertation, School of Electrical and Computer Engineering, Georgia Institute of Technology, 2002

[27] WorldScape, Inc., Single-Instruction Multi-Threaded Array Processor (SIMTAP), available online at www.wscapeinc.com, May 2004.