Architectural Support for 3D Graphics in the Complex Streamed Instruction Set

Dmitry Cheresiz[1] Ben Juurlink[2] Stamatis Vassiliadis[2] Harry A.G. Wijshoff[1]

[1] *Leiden Institute of Advanced Computer Science*
Leiden University
Leiden, The Netherlands

[2] *Computer Engineering Laboratory*
Delft University of Technology
Delft, The Netherlands

ABSTRACT
In this paper we extend the previously proposed *Complex Streamed Instruction Set (CSI)* architecture to provide for floating-point computations and conditional execution in order to efficiently support 3D graphics applications. The CSI extension is evaluated using an industry standard 3D benchmark, and compared to the Intel's Streaming SIMD Extension (SSE). Compared to a 4-way issue superscalar processor extended with SSE and capable of processing 8 single-precision floating-point operations in parallel, the same processor extended with CSI attains the speedups of 2.8 and 2.13 on the transform and lighting kernels and the speedup of 1.61 on the geometry computations in whole. We also study how performance scales with the number of floating-point units and observe that CSI extension allows to utilize them more efficiently then SSE. Finally, the performance bottlenecks of the SSE-enhanced superscalar CPUs on the 3D graphics workload are identified. Results show that performance of the 4-way issue machines is limited by the issue width and that of the 8-way machines is limited by the number of the cache ports.

KEY WORDS
Multimedia, 3D graphics, processor architecture

1 Introduction

To provide an efficient way of exploiting data-level parallelism present in multimedia applications, many CPU vendors extended their instruction set architectures (ISA) with Single Instruction Multiple Data (SIMD) type of instructions. At first, integer SIMD instructions, such as Intel's *MMX* [13], were provided in order to accelerate audio, video and 2D image processing applications. Later, floating-point (FP) SIMD instructions were provided, aimed primarily at accelerating 3D graphics processing. For example, Intel's *Streaming SIMD Extension* (SSE) [15] provides instructions which operate on four 32-bit FP values packed into a 128-bit register in parallel. While several papers have demonstrated that integer SIMD ISA extensions can improve the performance of many multimedia applications (see, e.g., [1, 14]), these instruction set enhancements have several limitations. These extensions are not able to fully exploit the parallelism present in multimedia applications. Specifically, since SIMD instructions operate on fixed-sized registers, the processor needs to issue more instructions per cycle in order to process more elements in parallel. However, it is generally accepted that increasing the issue width requires a substantial amount of hardware [5] and negatively affects the cycle time [12]. Another way to increase parallelism is to increase the size of the SIMD registers, but this approach implies that existing codes have to be recompiled or rewritten. These limitations were identified, for example, in [9] and a solution, called the Complex Streamed Instruction (CSI) set, was proposed. CSI instructions operate on arbitrary long two-dimensional data streams stored in memory. It has been shown that CSI can exploit more parallelism and attain higher speedups than existing integer SIMD extensions can.

Floating-point SIMD instructions have been quantitatively evaluated by Yang et al. [16] and positive results have been presented. However, the limitations of integer SIMD extensions described above apply to these extensions as well. This largely motivates the work presented here. In this paper we extend CSI with floating-point and conditional instructions in order to efficiently support 3D graphics applications with data-dependent control. The performance of the proposed ISA extension is evaluated using an industry-standard 3D graphics benchmark (SPEC's *Viewperf*). The main contributions of this paper can be summarized as follows:

- Compared to a 4-way issue superscalar processor extended with SSE and capable of performing 8 single-precision FP operations in parallel, the same processor extended with CSI attains a speedup of 1.61 on the geometry computations.
- It is shown that the performance of the CSI-enhanced processor scales better with the number of FP units than the performance of the processor extended with SSE. For example, when the number of FP operations which both processors are capable to perform in parallel is increased from 4 to 8 and to 16, the speedup of CSI over SSE increases from 1.41 to 1.61 and to 1.72.
- The bottlenecks of the SSE-enhanced superscalar processors are identified. Results show that the 4-way SSE-enhanced CPU can not utilize more than a single SSE unit and CPU's issue width constitutes the bottleneck. The 8-way SSE-extended CPU is unable to utilize more than two SSE units and its performance is limited by the number of cache ports.

This paper is structured as follows. In Section 2 we provide background information about 3D image processing. Section 3 describes the floating-point CSI instructions and the conditional operations added to support loops containing if-statements. In Section 4 the simulation methodology and the experimental results are presented and discussed. In this section we also show how the most important kernels of the geometry pipeline were implemented using CSI instructions. Section 5 briefly describes related work and contains our conclusions.

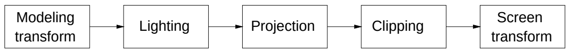

Figure 1. 3D geometry pipeline.

2 Geometry Processing

A 3D application, such as CAD tool, 3D game, or virtual reality application, creates the description of a 3D scene and stores it in a database. This information is then passed to the graphics processing system. In general, 3D graphics processing is a 3-stage pipeline consisting of the following steps [8]: (1) database traversal, (2) geometry calculation, and (3) rasterization. The first stage is responsible for extracting the information necessary for displaying a 3D scene created by an application. 3D objects are usually described by a set of polygons which model the object's surface. The information needed for display includes the coordinates of the polygon vertices, the type of primitive which has to be constructed from these vertices (e.g., line, polygon, triangle), material of the object, lighting model, camera position, and many others. This information is passed to the second stage, geometry computation, which calculates the color and coordinates of each vertex in the *screen* coordinate system. The final stage, rasterization, takes the color and screen coordinates of each vertex as input, calculates the pixel values for each point on the display, and stores them in the frame buffer from where they are taken and displayed on the monitor screen. In contemporary desktop systems, CPU performs the database traversal and the geometry processing stages, and special-purpose hardware (so-called graphics card) is employed for rasterization. The performance of graphics cards increased significantly in the last few years and according to some reports [11], the CPU has become the bottleneck for 3D graphics, because CPUs "only" double in speed every 18 months, while the performance of graphics processors increases by a factor of eight in the same period of time. Thererefore, a significant performance improvement in the geometry processing task performed by the CPU is required. To understand the ways this problem can be attacked, the geometry stage is described in more detail.

The computations carried out in the geometry stage are organized in a pipeline. The data representing a 3D scene flows through a number of kernels, as depicted in Figure 1. The 3D geometry pipeline uses several coordinate systems to describe the objects, namely *modeling coordinates*, *world coordinates*, *view coordinates* and *screen coordinates*. The modeling transform stage employs 3- and 4-dimensional matrix-vector multiplication to transform the modeling coordinates to world coordinates. The lighting stage computes the color of each vertex. The projection stage transforms vertex coordinates from the world to the view coordinate systems using 4D matrix-vector multiplication. During the clipping stage the objects are clipped to the viewable domain to avoid unnecessary rendering of the polygons positioned outside the area. The screen transformation stage converts the 4D view coordinates (x, y, z, w) to the 3D screen coordinates. It consists of two consecutive transformations. The first one divides the x, y, and z component by the w component. The second one multiplies the obtained vector with a 3×3 matrix and adds a displacement vector.

3 Floating-Point and Conditional Operations in CSI

In this section we briefly review the CSI architecture and then extend it to provide for the floating-point operations and conditional execution, which are essential for the 3D graphics.

CSI Overview. CSI is a memory-to-memory vector-like architecture. CSI instructions process arbitrary-length streams of data located in memory, performing arithmetic of logical operations. For example, the `csi_add` instruction loads two input streams from memory, adds their corresponding elements, and writes the resulting stream back to memory. Streams are two-dimensional. Each stream consists of an arbitrary number of rows, and the row elements are stored at a fixed stride which will be referred to as *horizontal stride*. There is also a fixed stride between consecutive rows, which will be referred to as *vertical stride*. Each stream is specified by a *set of stream control registers* (SCR-set). For a detailed description of the CSI architecture, the reader is referred to [9].

Floating-point operations in CSI. The CSI data streams, according to their definition in [9], consist of the elements that belong to one of the following six arithmetic data types: 8-, 16-, or 32-bit binary integers, signed or unsigned. The data type of the elements of a CSI stream is determined by the *Size* and *Sign* fields of the **S4** control register from the SCR-set that contains the stream parameters. We extend the notion of the CSI data streams and allow an extra data type for the stream elements: the 32-bit single precision floating point number format (IEEE 754 format). We also introduce a new 1-bit field *FP* into the **S4** register format, so that streams of the floating-point numbers can be specified. An SCR-set specifies such a stream if the fields of its **S4** register are set as follows: **S4.Size** = 4 (bytes), **S4.FP** = 1. The described extension allows the arithmetic CSI instructions, such as `csi_add`, to process streams of FP numbers.

Conditional Execution Support in CSI. Data-parallel applications operate on long streams of data and usually perform the same operation on each stream element. There are situations, however, in which the operation to be performed on each element is dependent on the element itself, as the following example shows:

```
for(i=0; i<N; i++){
   if(A[i] > 0.0)
      A[i] = A[i]+B[i];
}
```

The necessity to be able to handle such cases was recognized in the vector processing domain. A solution used there is *masking*. First, a *mask vector* is produced, whose i-th element is 1 if the corresponding condition is true and 0 otherwise. Thereafter, a *masked* arithmetic instruction is executed which operates as follows. If the i-th element of the mask vector is 1, then the operation is actually performed. Otherwise, a *no-op* is executed.

In CSI, we provide support for the masked execution in the following way. First, we extend CSI with a new type of data streams, the *CSI bit streams*, which will also be referred to as the *CSI mask streams*. A CSI bit stream is a continuous stream of bits in storage. It is completely specified by two parameters: *Base*, which is the address of the byte containing the first stream element, and *Length*, which is the number of bits (elements) in the stream. We extend the programmer-visible state of CSI with 16 *MSCR-sets*, or sets of *mask stream control registers*. Each of these sets consists of 2 32-bit registers, **Base** and **Length**, and can completely specify a CSI bit stream. Second, we change the instruction formats, adding an extra operand, which is called the *mask operand* and is required to be a CSI bit stream (specified via a MSCR-set). We introduce two modes of operation in CSI: *masked* and *unmasked*. The programmer-visible state of CSI is extended with the *stream status register (SSR)*, which contains the information about the current mode. If a CSI instruction is executed under the masked mode, the instruction processes the data streams under the control of its mask stream operand. If the current mode is *unmasked*, the instruction ignores the mask operand and is executed as usual.

Masked operations have one major disadvantage, however. If many elements of the mask vector are 0, the no-ops occupy resources while producing no useful results. The necessity to handle such situations efficiently was recognized, for example, in [10]. One of the techniques proposed there is the following: when a loop similar to the one presented earlier in this section has to be processed, the elements, for which the `if`-condition is true, should be extracted into a shorter stream. This stream can then be processed without no-ops, and the results can be inserted back into the destination stream. We adopt this idea and introduce the CSI instructions `csi_extract` and `csi_insert`.

- The `csi_extract SCRSk, SCRSi, MSCRSj` instruction performs the following operation. Each element of the mask stream described by MSCR-set `MSCRSj` is checked. If the element is zero, the corresponding element of the arithmetic stream `SCRi` is ignored, otherwise it is *extracted*, i.e., stored, in the output arithmetic stream `SCRSk`.
- The `csi_insert SCRSk,SCRSi,MSCRSj` instruction performs the reverse operation to `csi_extract`. It inserts the elements of the input stream `SCRSi` into that positions of the stream `SCRSk`, for which the corresponding elements of the mask stream `MSCRSj` are 1.

These instructions are intended to be used for loops that contain `if-then` conditionals. Similarly, for efficient processing of the loops which contain `if-then-else` conditionals, we extend CSI with two following instructions: `csi_split` and `csi_merge`. The first instruction splits the input data stream, under the control of the mask stream, into two shorter streams: one contains the elements for which the `if` condition is true, and another contains the elements for which it is false. The second instruction, `csi_merge`, performs the reverse operation, conditionally merging two streams into one.

4 Evaluation

In order to evaluate the performance of the proposed ISA extension, we simulated a superscalar processor without a multimedia ISA extension, a processor extended with SSE instructions and a processor extended with CSI instructions. We studied the SPEC *Viewperf* benchmark using the *DX-06* dataset. *Viewperf* is distributed with five datasets with widely varying characteristics, most important of which is the number of vertices per primitive, because it determines the length of the streams. It varies from 3.4 in *Awadvs* to around 400 in *CDRS*. The *DX-06* dataset lies between these extreme cases and has an average of 95 vertices per primitive, as well as acceptable simulation times.

4.1 Simulation Methodology and Tools

The simulator we used is the `sim-outorder` simulator of the SimpleScalar toolset (release 3.0) [2]. This cycle-accurate simulator simulates an out-of-order multiple-issue processor. A corrected version of the SimpleScalar memory model based on SDRAM specifications given in [4] was used.

The geometry computations in *Viewperf* are performed via API calls to the *MESA* library, which is the public-domain implementation of the *OpenGL* graphics library. To simulate the benchmarks on a standard superscalar processor and on superscalar processors with SSE and CSI extensions, we created three different versions of *Mesa*, *libMesaGL_scalar.a*, *libMesaGL_SSE.a* and *libMesaGL_CSI.a*, and obtained three versions of the *Viewperf* executable by linking it with these libraries. The scalar library was obtained by compiling the *Mesa* sources using *gcc* with the `-O2 -funroll-loops` optimization flags. SSE and CSI versions of the library were created as follows. First, the source files containing the kernels **xform_points_4fv** and **gl_color_shade_vertices_fast** were compiled to assembly using the `-O2` optimization flag. After that, these kernels were manually rewritten using SSE or CSI instructions and the object files were created. Finally, the rest of the *Mesa* source files were compiled to objects and the required libraries were created.

We decided to modify the mentioned kernels because, according to our experiments, on a 4-way superscalar machine they account for 42% and 16% of the total geometry execution time, respectively . In all libraries the call to the **render_vb** function was removed because it implements the rasterization stage of the **graphics** pipeline and this task is usually handled by the graphics card, not by the CPU.

4.2 Coding the Kernels Using SSE and CSI

The **xform_points_4fv** function implements 4x4 floating-point matrix-vector multiplication of a sequence of 4D vectors with a fixed 4x4 matrix. This kernel is a part of the *modeling* and *projection* stages of the pipeline. The kernel can be fully streamed and requires 8 CSI instructions, namely, 4 `csi_mul` and 4 `csi_acc_section` instructions. For the SSE implementation, the vendor code provided in [7] was used.

The **gl_color_shade_vertices_fast** kernel carries out the lighting stage of the pipeline. For brevity, this kernel will be referred to as the **light** kernel. It is a complex kernel consisting of about 80 lines of C code. A simplified pseudo code description is given in Figure 2(a). Before implementing this kernel with CSI instructions, some transformations were performed on the source code level, as depicted in Figure 2(b). The outermost loop was split into three loops. The first loop initializes the R,G,B color components of each vertex to the ambient light constants. The second loop calculates the contributions of the light sources to the vertex colors. Loop interchange was performed on this loop, because the number of lights is likely to be

```
GLfloat R,G,B,A;
/* Alpha is constant for all vertices */
A= ..;
for (j=0; j<n; j++){ /* each vertex*/
   /*set color to ambient light */
   R = ctx->Light.BaseColor[0];
   G= ..; B=..;
   for (each light source){
      calculate light’s R,G,B contribution;
      R+=contribution_of_this_light_R;
      G+=..;B+=..;
   }
   /*convert FP to integer for R,G,B,A*/
   frontcolor[0][j]=(GLfixed)(R);
   frontcolor[1][j]=(GLfixed)(G);
   frontcolor[2][j]=(GLfixed)(B);
   frontcolor[3][j]=(GLfixed)(A);
}
```

(a) Original code.

```
GLfloat tmp_R[],tmp_G[],tmp_B[],A;
A= ..;
for(j=0; j<n; j++){ /* each vertex*/
   /*set to ambient light constants*/
   tmp_R[j] = ctx->Light.BaseColor[0];
   tmp_G[j]= ..; tmp_B[j]=..;
}
for(each light source){
   for(j=0; j<n; j++){ /* each vertex*/
      calculate R,G,B contribution of light;
       tmp_R[j]+=contribution_of_this_light_R;
       tmp_G[j]+=..;tmp_B[j}+=..;
   }
}
/* convert FP to integer for R,G,B,A */
for(j=0; j<n; j++){ /* each vertex*/
   frontcolor[0][j]=(GLfixed)(tmp_R[j]);
   frontcolor[1][j]=(GLfixed)(tmp_G[j]);
   frontcolor[2][j]=(GLfixed)(tmp_B[j]);
   frontcolor[3][j]=(GLfixed)(A);
}
```

(b) Restructured code.

Figure 2. Original and restructured code of the lighting kernel

small (typically 1 or 2) which results in short streams and poor CSI performance. The calculations of the light source contribution contain several *if-then-else* statements and were implemented using conditional CSI instructions. The third loop converts the R,G,B,A color components of each vertex from floating-point to integer format. It was implemented using the `csi_fp_cvt.w` instruction.

For SSE, we did not perform the similar source code restructuring because such transformation would imply storing the intermediate results from SSE registers and then reloading them and, therefore, is likely to decrease the performance. Instead, as a base for the SSE implementation of this kernel, we used the code samples provided by Intel as well as [6].

4.3 Modeled Architecture

The base system is a 666 MHz 4-way superscalar processor with out-of-order issue and execution based on the Register Update Unit (RUU). The processor parameters are listed in Table 1. The *Viewperf* benchmark exhibits a very high instruction cache hit rates. Therefore, and in order to reduce simulation time, a perfect instruction cache is assumed. The processor is configured with the 32 KB 4-way associative L1 data cache which has the cache line size of 64 bytes and with the 1 MB 2-way associative L2 data cache having 128 byte lines. The access times for the caches are 1 and 6 cycles, respectively. The main memory is the standard 166 MHz SDRAM memory which has the row access, the row activate and the precharge times of 2 (memory) cycles. The memory bus is 64 bytes wide and is clocked at 166 MHz as well. The 4-way standard superscalar machine was configured with 4 single-precision FP adders and 4 single-precision FP multipliers. Both the SSE-enhanced and CSI-enhanced processors have one SIMD FP execution unit capable of performing 4 FP numbers in parallel. For SSE instructions, the latencies are the same as the latencies of the corresponding scalar instructions. For the CSI instructions, the latency of the main SIMD computation they perform, such as add, multiply, etc., is taken the same as the latency of the corresponding SSE/scalar instruction. The latency of instruction itself is usually much longer, and is non-deterministic, since the data streams it processes can be of arbitrary length. In this paper we simulated a CSI execution unit interfaced to the L1 data cache (a design of such a unit is presented in [3]). This decision is motivated by the high L1 hit rates exhibited by *Viewperf*. Similar to the standard superscalar and SSE-enhanced processors, the CSI-enhanced processor uses two cache ports which are time-multiplexed between load and store accesses for source, mask and destination streams.

We remark here that CSI instructions are executed in-order. This is necessary to ensure semantic correctness of the executed program. The CSI datapath performs parallel operations on several floating-point numbers, it does not execute several CSI instructions in parallel.

4.4 Experimental Results

In this section we present the speedups attained by the SSE and CSI multimedia ISA extensions and identify the bottlenecks of SSE performance.

Relative performance of the standard, SSE and CSI processors. First, we study the relative performance of all three processor designs and investigate the influence of the instruction window size on their performance by varying the size of the RUU and the LSQ (load-store queue). Recall that all machines considered are able to execute 4 FP operations in parallel. Figure 3 plots the speedups of the 4-way standard (referred as STD in the picture), SSE- and CSI-enhanced processors for varying RUU sizes with respect to the baseline system: the 4-way superscalar processor with an RUU with 32 entries. CSI clearly outperforms SSE and standard processor, especially for smaller RUU sizes. The reason for this is the

Table 1. Processor configuration.

Clock rate	666MHz	
Issue width	4	
Register update unit size	32	
Load-store queue size	16	
Branch Prediction		
Bimodal predictor size	2K	
Branch target buffer size	2K	
Return-address stack size	8	
Functional unit types, number and cycles (latency, recovery)		
Integer ALU	4	(1/1)
Integer MULT	4	
multiply		(3/1)
divide		(20/19)
Cache ports	2	(1/1)
Floating-point ALU	4	(2/1)
Floating-point MULT	4	
FP multiply		(4/1)
FP divide		(12/12)
sqrt		(24/24)

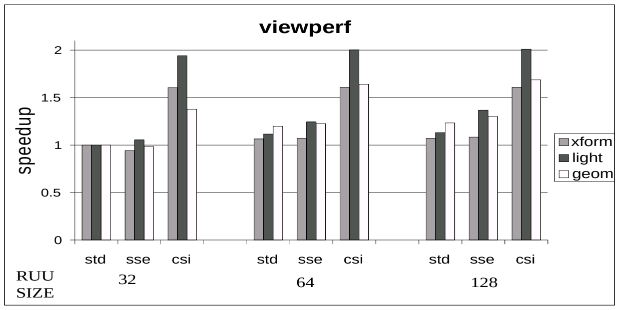

Figure 3. Speedups w.r.t. the baseline processor for varying RUU sizes.

following. Both the standard and the SSE architecture exploit the parallelism present in the 3D geometry pipeline in the form of the instruction-level parallelism (ILP). When the size of the RUU decreases, the processor's ability to utilize the ILP also decreases and so does the performance. For the CSI architecture, the parallelism present in computations is exploited in a single instruction and, therefore, insensitive to the RUU size. This feature of CSI also translates into dramatic reductions of instruction traffic. Results show thatCSI reduces the dynamic instruction count of the **xform** and **light** kernels by the factors of 30 and 5.1 and that of the whole geometry pipeline by the factor of 2.71. SSE provides the reduction only by the factors of 1.45, 1.13, and 1.15, respectively. The ability of CSI to provide high performance with the smaller instruction window is rather beneficial, since this resource is expensive and its cost grows quadratically with respect to its size.

Figure 3 also shows that performance of the SSE-enhanced processor is close to that of the processor without any media extension. However, the SSE implementation is likely to achieve a higher clock rate, because it requires fewer register file ports. Another important advantage of SSE is that it allows scaling the number of FP execution units so that it can execute up to 16 FP operations in parallel without increasing the issue width. We, therefore, no longer include the results for the standard superscalar processor.

Performance scalability and bottlenecks. Next we study the performance scalability of the SSE- and CSI-enhanced machines with respect to the number of floating-point computing resources, the issue width and the number of cache ports in order to identify the bottlenecks. The following machines are considered. As the baseline 4-issue processor we take the 4-issue CPU with 128-entry RUU and other parameters as in Table 1. For the baseline 8-issue machine we double the RUU size and the number of functional units. The processors are configured with 1, 2, or 4 SSE execution units. The same processors are enhanced with CSI execution unit having a 128-, 256-, or 512-bit wide datapath. So, these CPUs are also able to perform 4, 8, or 16 single-precision FP operations in parallel. Each of the processors is then configured with 2 or 4 cache ports.

Figure 4 depicts the speedups of the SSE- and CSI-enhanced processors comparing each to the 4-way SSE-enhanced processor with two cache ports and capable of performing 4FP operations is parallel. The results for CPUs with two cache ports are denoted with 'mem2', and those for CPUs with four ports with 'mem4'. The bars denoted with '4fp', '8fp', and '16fp' show the speedups attained by the SSE- or CSI-enhanced CPUs capable of performing 4, 8, and 16 FP operations in parallel. The presented results lead to the following observations. First, the 4-way SSE CPUs can not exploit more than one SSE unit (4 parallel FP operations). Since increasing the number of cache ports does not significantly improve the performance, we conclude that the issue width of such CPU constitutes the bottleneck. This conclusion is supported by the observation that increasing the issue width of an SSE-enhanced CPU from 4 to 8 speeds up the geometry pipeline approximately by the factor of 1.8, if sufficient number of cache ports is provided. The performance of the 8-way SSE CPU strongly depends on the number of cache ports, leading to the conclusion that the cache ports constitute the bottleneck for this machine. We also observe that even provided with 4 ports, 8-way SSE CPU can not utilize more then 2 SSE units (8 parallel FP operations).

The results show that, contrary to the SSE-enhanced processors, the CSI-enhanced ones can exploit efficiently the hardware that allows to perform up to 16 parallel FP operations and can provide high performance without increasing the issue width. This is rather natural, since the parallelism in CSI is exploited in a single instruction and not in the form of ILP. This is particularly evident for the **xform** kernel which is fully streamed. Bigger issue width provides, however, the performance improvements for the geometry pipeline in whole due to the presence of the non-streamed code sections.

Furthermore, the performance of a CSI-enhanced CPU is not sensitive to the number of cache ports. This is natural as well, because the L1-interfaced implementation of CSI we study delivers a whole cache line in a single access and, therefore, performs less accesses. The ability of CSI to exploit large number of parallel FP processing hardware without increasing the issue width or the number of cache ports is very important for the processor design, because the increase of any of these parameters bears huge hardware costs (we are not aware of any existing CPU which has the issue-width of 8 or a 4-ported cache) and can negatively affect the cycle time. On the other hand, the increasing scale of integration and transistor

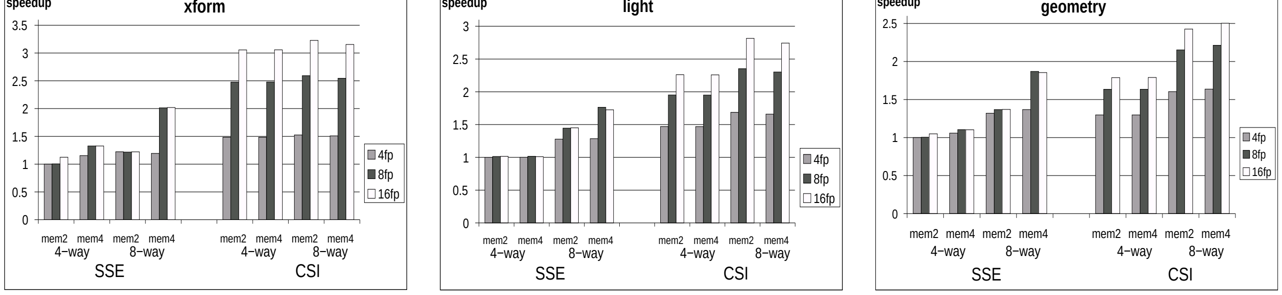

Figure 4. Speedups of CSI and SSE CPUs over the baseline 4-way SSE CPU.

budgets allow a designer to put dozens of functional units of a single chip and the main challenge is how to utilize them. CSI provides the solution, scaling very well when the number of the parallel execution hardware is increased. We observe that a 4-way CSI-enhanced CPU with 2 cache ports and capable of performing 16 parallel FP operations on the **xform** and **light** kernels outperforms very aggressive 8-way SSE-enhanced CPU which has 4 cache ports and performs just 10% slower on the geometry pipeline in whole.

5 Related Work and Conclusions

The CSI architectural paradigm was presented and evaluated in [9]. Another proposal aimed at exploiting higher degrees of data-level parallelism in multimedia applications is the Matrix Oriented Multimedia (MOM) ISA extension. MOM is a register-to-register vector-like architecture and its instructions can be viewed as vector versions of integer SIMD instructions. MOM does not provide floating-point nor conditional instructions. Another related proposal is the Imagine processor [10], which has a load/store architecture for one-dimensional streams of data records. This architecture belongs to a different category than CSI, since it is a stand-alone media processor and not an ISA extension implemented in a CPU.

In this paper we extended the CSI architecture with floating-point instructions and added architectural support for implementing loops with data-dependent control flow. It was shown that the most important parts of the 3D geometry pipeline can be implemented using the proposed extensions. The performance benefits provided by CSI were compared with those achieved by Intel's *SSE* extension. They were evaluated by implementing several routines of the *Mesa* 3D library using CSI and SSE instructions and by evaluating the library performance using the industry-standard 3D performance evaluation benchmark *Viewperf*.

For the 4-way issue machine with an RUU of 32 entries and floating-point resources capable of performing 4 single-precision floating-point operations is parallel, the speedups of CSI over SSE on the kernels **xform** and **light** were 1.70 and 1.84, respectively. These speedups translated to an application speedup of 1.40. The performance scalability of SSE and CSI with respect to the number of floating-point computing resources was also studied, showing that CSI can exploit more parallelism. For the processors capable of executing 8 single-precision FP operations in parallel and having an 128-entry instruction window, the speedups of CSI over SSE were equal to 2.46, 1.92, and 1.62 for the **xform** and **light** kernels, and the **geometry** computation, respectively.

References

[1] R. Bhargava, L. John, B. Evans, and R. Radhakrishnan. Evaluating MMX Technology Using DSP and Multimedia Applications. In *MICRO 31*, pages 37–46, 1998.

[2] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept., 1997.

[3] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff. Implementation of a Streaming Execution Unit. In *EUROMICRO 28*, 2002.

[4] M. Gries. The Impact of Recent DRAM Architectures on Embedded Systems Performance. In *EUROMICRO 26*, 2000.

[5] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[6] Diffuse-Directional Lighting, Application Note AP-596, Intel Corp., 1999.

[7] Streaming SIMD Extensions - Matrix Multiplication, AP-930, Intel Corp.

[8] J.Foley, A. van Dam, S.Feiner, and J.Hughes. *Computer Graphics - Principles and Practice*. Addison-Wesley, 1996.

[9] B. Juurlink, D. Tcheressiz, S. Vassiliadis, and H. Wijshoff. Implementation and Evaluation of the Complex Streamed Instruction Set. In *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[10] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing With Streams. *IEEE Micro*, 21(2):35–47, 2001.

[11] Transform and Lighting, Techinical Brief, Nvidia Corp. Document available via http://www.nvidia.com/docs/IO/1345/ATT/ TransformAndLighting.pdf.

[12] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *ISCA'97*, 1997.

[13] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[14] P. Ranganathan, S. Adve, and N. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *ISCA 26*, pages 124–135, 1999.

[15] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, May 1999.

[16] C. Yang, B. Sano, and A. Lebeck. Exploiting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions . *IEEE Transactions on Computers*, 49(9):934–946, 2000.