Volume 23 (2004), Number 4 pp. 1–18

SIMD Optimization of Linear Expressions for Programmable Graphics Hardware

Chandrajit Bajaj† and Insung Ihm‡ and Jungki Min‡ and Jinsang Oh‡

† Department of Computer Science, University of Texas at Austin, Texas, U.S.A.
‡Department of Computer Science, Sogang University, Seoul, Korea

Abstract

The increased programmability of graphics hardware allows efficient GPU implementations of a wide range of general computations on commodity PCs. An important factor in such implementations is how to fully exploit the SIMD computing capacities offered by modern graphics processors. Linear expressions in the form of $\bar{y} = A\bar{x} + \bar{b}$, where A is a matrix, and $\bar{x}$, $\bar{y}$, and b are vectors, constitute one of the most basic operations in many scientific computations. In this paper, we propose a SIMD code optimization technique that enables efficient shader codes to be generated for evaluating linear expressions. It is shown that performance can be improved considerably by efficiently packing arithmetic operations into four-wide SIMD instructions through reordering of the operations in linear expressions. We demonstrate that the presented technique can be used effectively for programming both vertex and pixel shaders for a variety of mathematical applications, including integrating differential equations and solving a sparse linear system of equations using iterative methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors, Parallel processing, Programmable shader; G.1.3 [Numerical Analysis]: Numerical Linear Algebra, Sparse systems; G.1.6 [Numerical Analysis]: Optimization

1. Introduction

In recent years, commodity graphics hardware has evolved beyond the traditional fixed-function pipeline, to allow flexible and programmable graphical processing units (GPUs). User-programmable vertex and pixel shader technologies have been applied extensively, and have added a large number of interesting new visual effects that were difficult or impossible with traditional fixed-function pipelines. Significant effort has focused mainly on the efficient utilization of pixel shader hardware by effectively mapping advanced rendering algorithms to the available programmable fragment pipelines. Flexible multi-texturing and texture-blending units, as provided by recent graphics cards, allow a variety of per-pixel shading and lighting effects, such as Phong shading, bump mapping, and environmental mapping.

In addition to these traditional per-pixel rendering effects, the list of graphics applications accelerated by programmable shader hardware is growing rapidly. Volume rendering is an actively studied topic in which pixel shading hardware has been exploited extensively for more flexible classification and shading at higher frame rates (refer to [6] for the various volume rendering techniques using user-programmable graphics hardware). In [25, 26], real-time procedural shading systems were proposed for programmable GPUs. Two papers demonstrated that ray casting can be performed efficiently with current graphics hardware. Carr et al. described how ray-triangle intersection can be mapped to a pixel shader[3]. Their experimental results showed that a GPU-enhanced implementation is faster on existing hardware than the best CPU implementation. Purcell et al. presented a streaming ray tracing model suitable for GPU-enhanced implementation on programmable graphics hardware[28]. They evaluated the efficiency of their ray tracing model on two different architectures, one with and one without branching. Recently, global illumination algorithms such as photon mapping, and matrix radiosity and subsurface scattering, were implemented on GPUs by Purcell et al.[27] and Carr et al.[2], respectively.

Programmable graphics hardware has also been used for more general mathematical computations. Hart showed how the Perlin noise function can be implemented as a multi-pass pixel shader[11]. In [18], Larsen et al. described the use of texture mapping and color blending hardware to perform large matrix multiplications. Thompson et al. also implemented matrix multiplication, and non-graphics problems such as 3-satisfiability, on GPUs[33]. Hardware-accelerated methods were proposed for computing line integral convolutions and Lagrangian-Eulerian advections by Heidrich et al.[12] and Weiskopf et al.[35]. In [30], Rumpf et al. attempted to solve the linear heat equation using pixel-level computations. Harris et al. also implemented a dynamic simulation technique, based on the coupled map lattice, on programmable graphics hardware[10]. Moreland et al. computed the fast Fourier transform on the GPU, and used graphics hardware to synthesize images[23].

As the fragment processing units of the recent graphics accelerators support full single-precision floating-point arithmetic, it becomes possible to efficiently run various numerical algorithms on the GPU with high precision. Two fundamental numerical algorithms for sparse matrices, a conjugate gradient solver and a multigrid solver, were mapped to the GPU by Bolz et al.[1]. Goodnight et al. also implemented a general multigrid solver on the GPU, and applied it to solving a variety of boundary value problems[8]. Krüger et al. described a framework for the implementation of linear algebra operators on a GPU[17]. The GPU was also used for solving large nonlinear optimization problems by Hillesland et al.[13]. Harris et al. solved partial differential equations on the GPU for simulating cloud dynamics[9].

Many of the numerical simulation techniques mentioned are strongly reliant on arithmetic operations on vectors and matrices. Therefore, considerable efforts have been made to develop efficient representations and operations for vectors and matrices on GPUs. This paper continues those efforts, and is specifically concerned with the problem of computing linear expressions in the form of affine transforms $\bar{y} = A\bar{x} + \bar{b}$, where A is a matrix, and $\bar{x}$, $\bar{y}$, and $\bar{b}$ are vectors. Such linear transforms constitute basic operations in many scientific computations. Because the matrix A in the expression is usually large and sparse in practice, many CPU-based parallel techniques have been proposed in parallel processing fields for their efficient computations. In particular, several methods for large sparse matrix-vector multiplication have been designed for SIMD machines[24, 37, 15, 29]. Although the proposed techniques work well on specific machines, it is difficult to apply them directly to the simple SIMD models supported by current programmable graphics hardware.

This paper presents a SIMD code optimization technique that enables efficient assembly-level shader codes to be generated for evaluating linear expressions. The technique transforms a given linear expression into an *equivalent* and *optimized* expression that can be evaluated using fewer instructions on the SIMD graphics architecture supported by current GPUs. Arithmetic operations are packed into four-wide SIMD instructions by reordering the operations in the linear expression. Our technique is different from other related GPU programming techniques because it searches for the most efficient linear expression to make the best use of the four-wide SIMD processing power of current GPUs. We demonstrate that the proposed optimization technique is quite effective for programming both vertex and pixel shaders in a variety of applications, including the solution of differential equations, and sparse linear systems using iterative methods.

Operation	Usage	Description
ADD	`ADD D, S0, S1`	`D` ← `S0` + `S1`
MUL	`MUL D, S0, S1`	`D` ← `S0` * `S1`
MAD	`MAD D, S0, S1, S2`	`D` ← `S0` * `S1` + `S2`
DP4	`DP4 D, S0, S1`	`D` ← `S0` · `S1`
MOV	`MOV D, S`	`D` ← `S`

Table 1: *Supported instructions.* `S, S0, S1`*, and* `D` *are four-wide registers.* +*,* ***,** *and* · *represent component-wise addition, component-wise multiplication, and four-component dot product, respectively.*

2. Efficient SIMD Computation of Linear Expressions

2.1. Abstract Model of SIMD Machine

We assume an abstract model of the shader for which our optimization technique is developed. Vertex and pixel shaders of different manufactures are slightly different from each other. Furthermore, they are still evolving rapidly. In this respect, we make only a few basic assumptions about the shader model so that the resulting technique is vendor-independent, and can be easily adapted to future shaders. In this paper, we view the shaders as general purpose vector processors with the following capabilities:

1. The shader supports four-wide SIMD parallelism.
2. Its instruction set includes the instructions shown in Table 1.
3. Any component of the source registers may swizzle and/or replicate into any other component. Furthermore, destination registers may be masked. These register modifiers do not harm shader performance.
4. Every instruction executes in a single clock cycle. Hence, the number of instructions in a shader program is the major factor affecting shader performance.

2.2. Definition of the Problem

As described earlier, this paper describes the generation of efficient shader code that evaluates a linear expression of the

form $E : \bar{y} = A\bar{x} + \bar{b}$, where the matrix $A = (a_{ij})_{i,j=0,1,\cdots,m-1}$ and the vector $\bar{b} = (b_0\ b_1\ \cdots\ b_{m-1})^t$ are constants, and the two vectors $\bar{x} = (x_0\ x_1\ \cdots\ x_{m-1})^t$ and $\bar{y} = (y_0\ y_1\ \cdots\ y_{m-1})^t$ are variables. Because the vertex and pixel processing of currently available GPUs is based on four-wide SIMD floating-point operations, we consider linear expressions with sizes that are multiples of four, that is, $m = 4n$ for a positive integer n. Note that linear expressions of arbitrary size can be *augmented* to such expressions by padding the appropriate number of zeros at the ends of the matrix and vectors.

Let $A_{ij} = (a^{ij}_{pq})_{i,j=0,1,\cdots,n-1,\ p,q=0,1,2,3}$ be the (i,j)-th 4×4 submatrices that partition A. Then the linear expression is described in the following form that is more amenable to four-wide SIMD processing:

$$\bar{y_i} = \sum_{j=0}^{n-1} A_{ij}\bar{x_j} + \bar{b_i} \text{ for all } i = 0, 1, \cdots, n-1, \qquad (1)$$

where $\bar{x_j} = (x_{4j}\ x_{4j+1}\ x_{4j+2}\ x_{4j+3})^t$, $\bar{y_i} = (y_{4i}\ y_{4i+1}\ y_{4i+2}\ y_{4i+3})^t$, and $b_i = (b_{4i}\ b_{4i+1}\ b_{4i+2}\ b_{4i+3})^t$.

If we define T_{ij} to be $A_{ij}\bar{x_j}$ for $i, j = 0, 1, \cdots, n-1$, the following partial product becomes the basic component in our code optimization technique:

$$T_{ij} = \begin{pmatrix} a^{ij}_{00}\cdot x_{4j} + a^{ij}_{01}\cdot x_{4j+1} + a^{ij}_{02}\cdot x_{4j+2} + a^{ij}_{03}\cdot x_{4j+3} \\ a^{ij}_{10}\cdot x_{4j} + a^{ij}_{11}\cdot x_{4j+1} + a^{ij}_{12}\cdot x_{4j+2} + a^{ij}_{13}\cdot x_{4j+3} \\ a^{ij}_{20}\cdot x_{4j} + a^{ij}_{21}\cdot x_{4j+1} + a^{ij}_{22}\cdot x_{4j+2} + a^{ij}_{23}\cdot x_{4j+3} \\ a^{ij}_{30}\cdot x_{4j} + a^{ij}_{31}\cdot x_{4j+1} + a^{ij}_{32}\cdot x_{4j+2} + a^{ij}_{33}\cdot x_{4j+3} \end{pmatrix} \qquad (2)$$

2.3. Our Optimization Technique

Shader performance greatly depends on the number of instructions in the shader program[21]. Our shader optimization technique attempts to minimize the number of instructions generated by fully exploiting the vector nature of both linear expressions and shader instructions. In our scheme, a linear expression is translated into a shader program through the following two stages:

1. **Transformation Stage:** Transform a given linear expression E into a *more efficient equivalent* expression E^*.
2. **Code Generation Stage:** Generate the shader code by evaluating Eq. (1) for E^*.

As will be explained below, the number of shader instructions necessary for evaluating $E : \bar{y} = A\bar{x} + \bar{b}$ depends on the pattern of non-zero elements of A and $\bar{b}$. The goal of the transformation stage is to search for a *better* linear expression that is equivalent to the given expression, and can be evaluated using fewer instructions. Once a more efficient linear expression is found, the shader code is generated using the supported SIMD instructions as efficiently as possible in the second stage. Because the transformation stage relies on a cost function defined in the code generation stage, we explain the latter first.

2.3.1. Code Generation Stage

To compute the four-vector $T_{ij} = A_{ij}\bar{x_j}$ in Eq. (2), 12 additions and 16 multiplications must be carried out in the worst case. However, when the matrix A_{ij} contains zero elements, it can be evaluated in fewer arithmetic operations. Notice that the SIMD instructions offered by graphics processors perform multiple arithmetic operations simultaneously. For instance, the `MAD` and `DP4` instructions in currently available consumer graphics processors perform four additions and four multiplications, and three additions and four multiplications in a single clock cycle, respectively.

When T_{ij} is evaluated with the SIMD instructions, the key is to exploit their parallelism cleverly. If a `MAD` instruction, for instance, is used to multiply any null element of A_{ij}, the GPU's parallel computing power is wasted. It is important to rearrange the arithmetic expressions Eq. (2) appropriately, with trivial terms deleted, so that each applied SIMD instruction performs as many non-trivial additions and multiplications as possible. Note that the computation of multiplying A_{ij} and $\bar{x_j}$ relies on the structure of A_{ij}. A close analysis of the pattern of zero elements of A_{ij} suggests two evaluation methods, which we call column-major multiplication and row-major multiplication, respectively.

Column-major multiplication: This method uses the `MAD` and `MUL` instructions to evaluate T_{ij}. As illustrated in Figure 1(a), A_{ij} is *peeled off vertically* such that each skin covers as many non-zero elements as possible. Let the cost $c_E(i,j)$ (three in this example) be the number of necessary peels. Then, it is trivial to see that T_{ij} can be evaluated in one `MUL` and $c_E(i,j)-1$ subsequent `MAD` instructions. If matrix-vector multiplication is to be implemented in this way, the non-zero elements of A_{ij} that are multiplied in each instruction must be loaded in a register before the shader is executed. In this example, assume that $(a^{ij}_{01}\ a^{ij}_{10}\ a^{ij}_{21}\ a^{ij}_{30})^t$, $(a^{ij}_{02}\ a^{ij}_{12}\ 0\ a^{ij}_{31})^t$, and $(0\ a^{ij}_{13}\ 0\ 0)^t$ are stored in registers `C27`, `C28`, and `C29`, respectively. Let the register `R4` also contain $(x_{4j}\ x_{4j+1}\ x_{4j+2}\ x_{4j+3})^t$. Then the following three shader instructions compute T_{ij} in column-major fashion, and put the result T_{ij} into the register `R3`:

```
MUL   R3, C27, R4.yxyx
MAD   R3, C28, R4.zzzy, R3
MAD   R3, C29, R4.wwwz, R3
```

The source-argument swizzles clearly help in rearranging and replicating the $\bar{x_j}$'s elements.

Row-major multiplication: `DP4` is another instruction that is useful for exploiting SIMD parallelism. In fact, it is very natural to matrix-vector multiplication. If we define the cost $r_E(i,j)$ to be the number of rows of A_{ij} that contain at least one non-zero element, it is also trivial to see that T_{ij} can be evaluated using $r_E(i,j)$ `DP4` instructions (see Figure 1(b)). If the registers `C27`, `C28`, and `C29` hold $(0\ a^{ij}_{01}\ a^{ij}_{02}\ 0)^t$,

$$T_{ij} = \begin{bmatrix} 0 & a_{01}^{ij} & a_{02}^{ij} & 0 \\ a_{10}^{ij} & 0 & a_{12}^{ij} & a_{13}^{ij} \\ 0 & a_{21}^{ij} & 0 & 0 \\ a_{30}^{ij} & a_{31}^{ij} & 0 & 0 \end{bmatrix} \begin{pmatrix} x_{4j} \\ x_{4j+1} \\ x_{4j+2} \\ x_{4j+3} \end{pmatrix} = \begin{pmatrix} \overbrace{a_{01}^{ij} \cdot x_{4j+1}}^{\texttt{MUL}} + \overbrace{a_{02}^{ij} \cdot x_{4j+2}}^{\texttt{MAD}} + \overbrace{a_{13}^{ij} \cdot x_{4j+3}}^{\texttt{MAD}} \\ a_{10}^{ij} \cdot x_{4j} + a_{12}^{ij} \cdot x_{4j+2} \\ a_{21}^{ij} \cdot x_{4j+1} \\ a_{30}^{ij} \cdot x_{4j} + a_{31}^{ij} \cdot x_{4j+1} \end{pmatrix}$$

(a) Column-major multiplication

$$T_{ij} = \begin{bmatrix} 0 & a_{01}^{ij} & a_{02}^{ij} & 0 \\ a_{10}^{ij} & 0 & a_{12}^{ij} & a_{13}^{ij} \\ 0 & a_{21}^{ij} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_{4j} \\ x_{4j+1} \\ x_{4j+2} \\ x_{4j+3} \end{pmatrix} = \begin{pmatrix} \overbrace{a_{01}^{ij} \cdot x_{4j+1} + a_{02}^{ij} \cdot x_{4j+2}}^{\texttt{DP4}} \\ \overbrace{a_{10}^{ij} \cdot x_{4j} + a_{12}^{ij} \cdot x_{4j+2} + a_{13}^{ij} \cdot x_{4j+3}}^{\texttt{DP4}} \\ \overbrace{a_{21}^{ij} \cdot x_{4j+1}}^{\texttt{DP4}} \\ 0 \end{pmatrix}$$

(b) Row-major multiplication

Figure 1: *Efficient SIMD evaluation of T_{ij}. Not all arithmetic operations must be performed because some entries are null. The proposed optimization technique identifies the non-zero entries and reorganizes them to produce a code with fewer SIMD instructions.*

$(a_{10}^{ij}\ 0\ a_{12}^{ij}\ a_{13}^{ij})^t$, and $(0\ a_{21}^{ij}\ 0\ 0)^t$, respectively, the following instructions compute T_{ij} in row-major fashion:

```
DP4    R3.x, C27, R4
DP4    R3.y, C28, R4
DP4    R3.z, C29, R4
```

When T_{ij} is evaluated, the multiplication method with the smaller cost is selected. When $c_E(i,j)$ and $r_E(i,j)$ are even, the column-major method is chosen for the reason that will be explained in the next paragraph. Then the cost, i.e., the number of necessary instructions, of multiplying A_{ij} with $\bar{x}_j$ is defined as $C_E^{mul}(i,j) = \min(c_E(i,j), r_E(i,j))$.

Once the terms T_{ij} are obtained for all j, they are added to $\bar{b}_i$ using the `ADD` instruction. The addition process must be coded carefully. First, neither null T_{ij} nor null $\bar{b}_i$ should be added (notice that T_{ij} vanishes if all elements of A_{ij} are null). Secondly, an `ADD` instruction is saved if it can combine with the first `MUL` instruction of the column-major multiplication. That is, by replacing `MUL` in the column-major multiplication with `MAD`, and accumulating the partial products of T_{ij} to the register corresponding to $\bar{y}_i$, the addition process can be implemented more efficiently. This is why column-major multiplication is preferred to the row-major multiplication when their respective costs are the same. In summary, the code generation stage is described as follows:

1. Terms T_{ij} with $r_E(i,j) < c_E(i,j)$ are evaluated first, in row-major fashion. Their results and $\bar{b}_i$ are accumulated to the destination register for $\bar{y}_i$ using the `ADD` instruction.
2. Then the remaining terms T_{ij} are evaluated in column-major fashion, accumulating to the destination register.

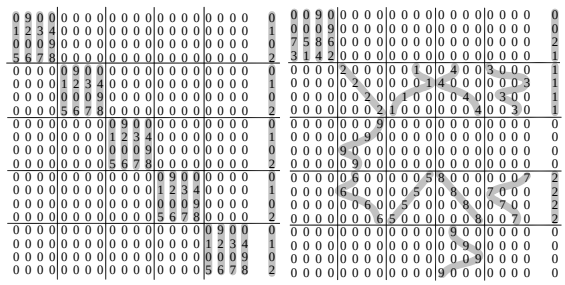
Let $n_E^r(i)$ be the number of terms T_{ij}, $j = 0, 1, \cdots, n-1$, that are computed in row-major fashion. If we define $C_E^{sum}(i)$ to be the cost of the summation process, i.e., the number of additional ADD instructions required, it becomes then $C_E^{sum}(i) = n_E^r(i)$ if $\bar{b}_i$ is not trivial, or $n_E^r(i) - 1$ otherwise.

2.3.2. Transformation Stage

Now by combining the two cost functions, the evaluation cost $C(E)$ for $E : \bar{y} = A\bar{x} + \bar{b}$ can be defined to be $C(E) = \sum_{i,j=0}^{n-1} C_E^{mul}(i,j) + \sum_{i=0}^{n-1} C_E^{sum}(i)$. This metric represents the number of shader instructions required to compute E, and is determined by the patterns of non-zero elements in A and $\bar{b}$. The key to our shader optimization technique is the fact that exponentially many linear expressions exist equivalent to the given expression E. We consider two linear expressions $E : \bar{y} = A\bar{x} + \bar{b}$ and $E' : \vec{y}' = A'\vec{x}' + \bar{b}'$ to be *equivalent* to each other if $\bar{y}$ can be obtained by rearranging the elements of $\vec{y}'$. Basically, the results of two equivalent expressions are the same except for the order in which the computed values are stored in memory.

Figure 2 shows augmented matrices $[A \mid b]$ of two equivalent linear expressions, found in the example of Subsection 3.2, which have costs of 20 and 14, respectively. The thick (curved) line segments in the matrix parts indicate how the corresponding matrix-vector multiplications are performed. It is obvious that the expression in Figure 2(b) can be implemented more efficiently.

As mentioned previously, the goal of our optimization technique is to produce a shader written using the minimum number of instructions. The problem is how to effectively find an equivalent linear expression at the minimum cost.



(a) Before transformation (b) After transformation

Figure 2: *Example of two equivalent linear expressions. When the expression in (a) is evaluated carelessly using* MAD *and* MUL *instructions, the SIMD processing power is wasted. The equivalent expression in (b) can be evaluated more efficiently using fewer SIMD instructions.*

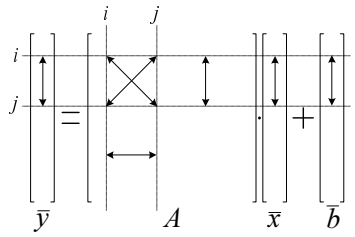


Figure 3: (i, j)*-swapping of a linear expression. Interchanging the i-th and j-th rows and columns does not harm the equivalence of the expression.*

Consider an (i, j)-swapping operation applied to a linear expression, as illustrated in Figure 3. This operation, denoted by $\Rightarrow$, transforms a linear expression to another equivalent expression by swapping the i-th and j-th rows of A and b, and the corresponding columns of A. Because any equivalent expression can be obtained by repeatedly applying a sequence of (i, j)-swapping operations, the shader optimization problem for evaluating linear expressions can be restated as follows:

> *Minimize $C(E)$ subject to $E_0 \stackrel{*}{\Rightarrow} E$, where E_0 is a given linear expression.*

In the transformation stage, we search for an expression with minimum cost by attempting to solve the minimization problem. Notice that the configuration space grows factorially with respect to the size m of the linear expression. A close investigation suggests that it is unlikely that a polynomial time algorithm will be found for this combinatorial minimization problem, although we have not proven its NP-hardness yet. In an attempt to reduce search costs, we use an efficient approximation algorithm that produces at least, near-optimal solutions. This is described in the next subsection.

```
E ← Given linear expression E0;
T ← Initial temperature T0;
while (stopping criterion is not satisfied) {
  while (not yet in equilibrium) {
    E' ← Some randomly picked neighboring equivalent expression
         of E; // line (a)
    ΔC ← C(E')−C(E); // line (b)
    p ← min(1, e^(−ΔC/(k·T))); // line (c)
    if (random(0, 1) < p)
      E ← E'; // line (d)
  }
  Update T; // line (e)
}
Output the best linear expression E;
```

Figure 4: *Simulated annealing algorithm*

2.3.3. Simulated Annealing: an Iterative Improvement Algorithm

The method of iterative improvement, or local search, is a simple and practical search technique using trial and error, and has been shown to be effective for solving many intractable combinatorial optimization problems. The basic idea is to move around the configuration space trying to find the global minimum. Iterative improvement usually keeps track of only the current state, and does not look ahead beyond the immediate neighbors of that state. Simulated annealing defines a class of iterative improvement algorithms, based on the analogy between combinatorial optimization problems and thermodynamics, especially the way that liquids freeze and crystallize, or metals cool and anneal. It was first presented in [16], where the idea was borrowed from the Metropolis algorithm, applied to simulating complex systems in statistical physics[22]. Since its introduction, this method of simulated annealing has attracted significant attention as suitable for large-scale optimization problems, especially ones where a desired global extremum is often hidden among many poorer local extrema [4, 34, 36].

It was designed to overcome the drawback of hill-climbing algorithms, another class of iterative improvement algorithms, which only move in the direction of decreasing cost, and hence easily become stuck in local minima or plateaux. Figure 4 shows our version of the simulated annealing algorithm that is used to search for a linear expression of minimum cost. Unlike hill-climbing, in which the *best* move is selected in each step, the simulated annealing technique picks a *random* move (`line (a)`). If the move actually lowers the cost, i.e., $\Delta C < 0$, it is always accepted. Otherwise, i.e., $\Delta C \geq 0$, the move is accepted only with probability $e^{-\frac{\Delta C}{k \cdot T}}$ (`line (b)-(d)`). This allows occasional 'uphill moves' in an attempt to move out of a local minimum in favor of searching for a better, more global,

one. These uphill moves are controlled probabilistically by the 'temperature' T. At higher values of T, uphill moves are more likely to be allowed. However, they become less likely toward the end of the process, as the value of T decreases. The temperature is lowered slowly according to an annealing schedule that dictates how it decreases from high to low values (`line (e)`). Theoretical analysis shows that if the schedule lowers T slowly enough, the algorithm converges with probability one to a global minimum. Unfortunately, the analysis provides little information on how to define a good schedule.

When the simulated annealing method is applied to combinatorial optimization, the schedule usually controls the temperature with a function of the number of steps that have already been taken. In our implementation, we have carefully selected through repeated trial-and-error experiments such control parameters as the number of iterations taken in each downward step, the decrease rate between subsequent downward steps, and the (Boltzmann's) constant $k(>0)$ that relates the cost C and the temperature T.

3. Applications to Vertex Shader Programming

We have implemented the shader optimization technique presented above, and applied it to various problems. When it is used in practice, the optimization scheme may have to be modified slightly to meet the specific requirements of each problem. In this section, we first explain how we applied our technique to efficiently implement three numerical problems. Then a pixel shader programming technique is presented in the next section.

3.1. Two-Dimensional Wave Equation

Waves are natural phenomena we experience in our everyday lives. Some simple forms of waves can be described mathematically by the wave equation, which is a prototypical second-order hyperbolic partial differential equation[32]. For example, two-dimensional waves like ripples on the surface of water, resulting from a stone being dropped in water, are described by the two-dimensional wave equation

$$\frac{\partial^2 z}{\partial t^2} = \alpha^2\left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right), \tag{3}$$

where the time-dependent wave $z(x,y,t)$ spreads out at the speed of α.

The wave equation is easily discretized using the finite-difference method. Consider a node (x_i,y_j) of a coordinate grid at a sequence of discrete times t_k, where $x_i = i\cdot\Delta x$ for $i = 0,1,\cdots,m$, $y_j = j\cdot\Delta y$ for $j = 0,1,\cdots,n$, and $t_k = k\cdot\Delta t$ for $k = 0,1,\cdots$. Then, applying Eq. (3) at the grid point (x_i,y_j) at the time instant t_k, and approximating the second partial derivatives with a central difference, we obtain a finite difference equation of the form $w_{i,j,k+1} = \lambda^2 w_{i-1,j,k} + \lambda^2 w_{i,j-1,k} + 2(1-2\lambda^2)w_{i,j,k} + \lambda^2 w_{i,j+1,k} + \lambda^2 w_{i+1,j,k} - w_{i,j,k-1}$, where $w_{i,j,k} = z(x_i,y_j,t_k)$ and $\lambda = \frac{\alpha\cdot\Delta t}{\Delta x}$ (for simplicity of explanation, we assume $\Delta x = \Delta y$). Then the entire equation set can be represented in the matrix form $\bar{y} = A\bar{x} + \bar{b}$, where $\bar{y}$, $\bar{x}$, and $\bar{b}$ are comprised of the $(k+1)$-th, k-th, and $(k-1)$-th time-step variables, respectively.

We tested our optimization technique with an 8×8 grid. Figure 5(a) shows a portion of the 64×64 matrix of the corresponding finite-difference equation. It takes 82 SIMD instructions to evaluate the linear expression using our code generation technique. After the linear expression is transformed through the simulated annealing algorithm, the number of instructions reduces to 63. Observe that non-zero elements are packed more compactly in the transformed matrix as shown in Figure 5(b). As a result, it becomes possible to exploit the four-wide SIMD parallelism of the vertex shader more effectively for computing the wave equation.

3.2. Fourth-Order Runge-Kutta Method

Integrating differential equations is an important part of dynamic simulation in computer graphics. The fourth-order Runge-Kutta method offers substantial improvement in accuracy over the lower-order techniques such as the Euler method. Nevertheless, it has not been popular in developing real-time applications because of its computational complexity. As an application of our optimization technique, we show that the fourth-order Runge-Kutta method, known to be expensive for real-time applications, can in fact be implemented efficiently on a SIMD graphics processor.

Consider a system of first-order differential equations

$$y_i' = f_i(t,y_0,y_1,\cdots,y_{n-1}),\ y_i(t_0) = y_i^0\ \ (i = 0,1,\cdots,n-1)$$

where the f_is are linearly defined as follows:

$$f_i(t,y_0,y_1,\cdots,y_{n-1}) = \sum_{j=0}^{n-1}\alpha_j^i\cdot y_j + \beta_i\ \ (i = 0,1,\cdots,n-1)$$

The formula for the *classical* fourth-order Runge-Kutta method is

$$y_i^{k+1} = y_i^k + \frac{1}{6}(a_i + 2b_i + 2c_i + d_i),\ k = 0,1,2,\cdots, \tag{4}$$

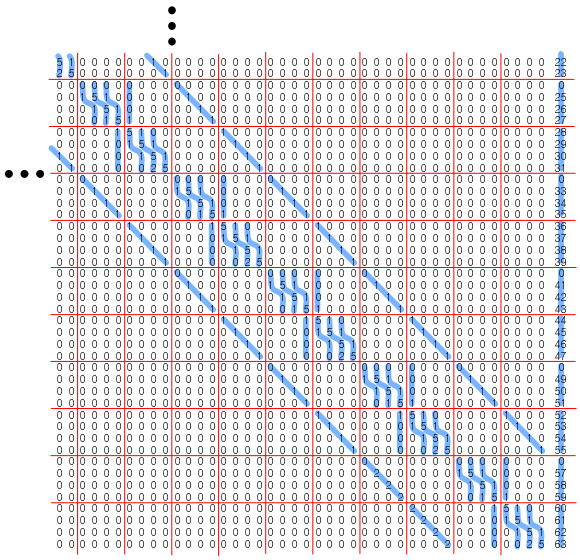
where the four classes of coefficients a_i, b_i, c_i, and d_i are computed through the following steps ($\gamma_j^i = h\alpha_j^i$, $\delta_i = h\beta_i$):

$$a_i \leftarrow \sum_{j=0}^{n-1}\gamma_j^i\cdot y_i^k + \delta_i \ \text{ for } i = 0,1,2,\cdots,n-1; \tag{5}$$

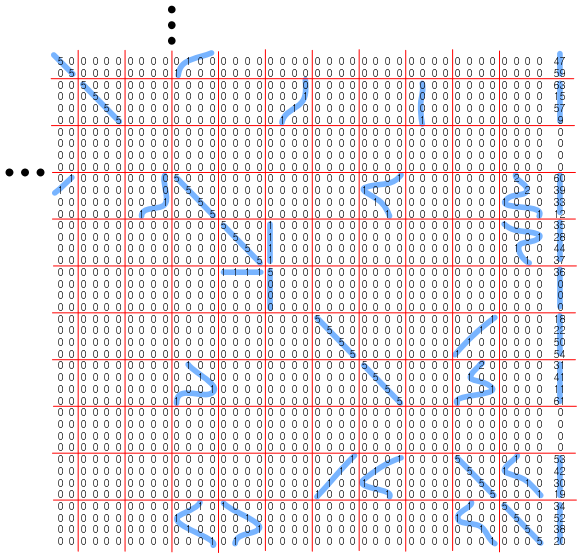
$$w_i \leftarrow y_i^k + \frac{1}{2}a_i \ \text{ for } i = 0,1,2,\cdots,n-1; \tag{6}$$

$$b_i \leftarrow \sum_{j=0}^{n-1}\gamma_j^i\cdot w_j + \delta_i \ \text{ for } i = 0,1,2,\cdots,n-1 \tag{7}$$

$$w_i \leftarrow y_i^k + \frac{1}{2}b_i \ \text{ for } i = 0,1,2,\cdots,n-1; \tag{8}$$



(a) Before transformation



(b) After transformation

Figure 5: *Transformation of the 2D wave equation. We find that the non-zero entries are packed very efficiently after the transformation process. As a result, the number of necessary SIMD instructions drops from 82 to 63. Note that each non-zero integer number in the matrix denotes a non-zero real number.*

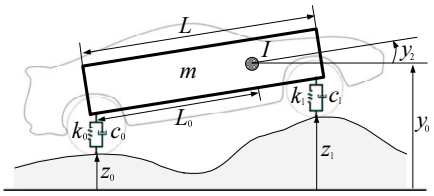


Figure 6: *Application to vehicle suspension design. A vehicle's suspension suitable for modeling bounce and pitch motions, is simulated using the fourth-order Runge-Kutta method on the GPU. This computation can be performed more effectively in the vertex processing stage than in the pixel processing stage, because the computed values are immediately used as parameters for modeling the transform of a vehicle model. No access to the framebuffer is necessary if the vertex state program, as supported by NVIDIA's GeForce GPUs, is used.*

$$c_i \leftarrow \sum_{j=0}^{n-1} \gamma_j^i \cdot w_j + \delta_i \text{ for } i = 0,1,2,\cdots,n-1; \quad (9)$$

$$w_i \leftarrow y_i^k + c_i \text{ for } i = 0,1,2,\cdots,n-1; \quad (10)$$

$$d_i \leftarrow \sum_{j=0}^{n-1} \gamma_j^i \cdot w_j + \delta_i \text{ for } i = 0,1,2,\cdots,n-1; \quad (11)$$

As an example, we consider a representation of a car's suspension suitable for modeling the bounce and pitch motions (see Figure 6). While originally used in investigating the ride-quality effects of road-surface variations in the direction of travel, it turns out to be very useful for increasing realism in real-time animation of moving vehicles. For the vertical translation y_0 of the mass center and the rotation y_2 about an axis, the following equations of motion can be derived[14]:

$$\begin{aligned}
y_0' &= y_1 \\
y_1' &= \frac{1}{m}\{-(k_0+k_1)y_0 - (c_0+c_1)y_1 \\
&\quad + (k_0L_0 - k_1(L-L_0))y_2 + (c_0L_0 - c_1(L-L_0))y_3 + d_0\} \\
y_2' &= y_3 \\
y_3' &= \frac{1}{I}\{(L_0k_0 - (L-L_0)k_1)y_0 + (L_0c_0 - (L-L_0)c_1)y_1 \\
&\quad - (L_0^2k_0 + (L-L_0)^2k_1)y_2 - (L_0^2c_0 + (L-L_0)^2c_1)y_3 + d_1\}
\end{aligned}$$

Here, d_0 and d_1 are values that are functions of the displacements z_0 and z_1 of tires from the reference height, and their derivatives z_0' and z_1'. They must be computed at each time step using the vehicle's speed and the road surface profile.

In this example, we consider five cars moving independently on the road, and obtain a system of 20 linear differential equations. To integrate the equations for each car, the expressions in Eq. (4) to (11) must be evaluated four times. This implies that they must be evaluated 20 times at each

time step. Although the amount of necessary computation appears too much for real-time simulation, it can be coded very compactly on SIMD graphics processors.

The expressions in Eqs. (4)–(11) can be partitioned into three classes: {Eq. (4)}, {Eqs. (5), (7), (9), (11)}, and {Eqs. (6), (8), (10)}. The equations in the second class comprise the most time-consuming part, so we focus on their optimization. Note that there are four sets of equations corresponding to the coefficients a_i, b_i, c_i, and d_i. While these sets must be evaluated in alphabetical order, the 20 equations of each set can be computed in arbitrary order. Furthermore, because the four sets share the same equations except for the run-time values of w_i, optimized codes for one set can be used for the others. Figure 2(a) is, in fact, a linear expression derived from the 20 equations corresponding to a coefficient of this example. The cost, that is, the number of necessary four-wide SIMD instructions when no transformation is applied, is 20. After the linear expression is transformed through the simulated annealing algorithm, we found that it can be coded in 14 instructions. Because there are four sets of identical equations, this results in a saving of 24 ($= 6 \cdot 4$) SIMD instructions. Figure 2(b) shows the transformed linear expression. It is obvious that the parallel computing power of SIMD graphics processor is utilized very well.

3.3. Gauss-Seidel Method

The Gauss-Seidel method is one of the most commonly used iterative methods for solving linear systems $A\bar{x} = \bar{b}$. It starts with an initial approximation $\bar{x}^{(0)}$, and generates a sequence of vectors $\bar{x}^{(l)}$, hoping to converge to the solution $\bar{x}$. It is extensively used in solving large and/or sparse systems found in various problems (some examples include the radiosity equation[5] and several finite-difference equations from computational fluid dynamics[32]).

The iteration in the Gauss-Seidel method is expressed as

$$D\bar{x}^{(l)} = -L\bar{x}^{(l)} - U\bar{x}^{(l-1)} + \bar{b},$$

where D is the diagonal part of A, and L and U are the lower and upper triangles of A with zeros on the diagonal, respectively. Note that each unknown is updated as soon as a newer estimate of that unknown is computed, which speeds up convergence. If we let $A_1 = -D^{-1}L$, $A_2 = -D^{-1}U$, and $b_2 = D^{-1}\bar{b}$, the equation becomes

$$\bar{x}^{(l)} = A_1\bar{x}^{(l)} + A_2\bar{x}^{(l-1)} + \bar{b}_2$$

In matrix form, it is

$$\begin{pmatrix} x_1^{(l)} \\ x_2^{(l)} \\ x_3^{(l)} \\ \vdots \\ x_n^{(l)} \end{pmatrix} = - \begin{bmatrix} 0 & & & & \\ \frac{a_{21}}{a_{22}} & 0 & & & \\ \frac{a_{31}}{a_{33}} & \frac{a_{32}}{a_{33}} & 0 & & \\ \vdots & \vdots & \cdots & \ddots & \\ \frac{a_{n1}}{a_{nn}} & \frac{a_{n2}}{a_{nn}} & \frac{a_{n3}}{a_{nn}} & \cdots & 0 \end{bmatrix} \begin{pmatrix} x_1^{(l)} \\ x_2^{(l)} \\ x_3^{(l)} \\ \vdots \\ x_n^{(l)} \end{pmatrix} -$$

$$\begin{bmatrix} 0 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ & 0 & \frac{a_{23}}{a_{22}} & \cdots & \frac{a_{2n}}{a_{22}} \\ & & \ddots & \vdots & \vdots \\ & & & 0 & \frac{a_{n-1,n}}{a_{n-1,n-1}} \\ & & & & 0 \end{bmatrix} \begin{pmatrix} x_1^{(l-1)} \\ x_2^{(l-1)} \\ x_3^{(l-1)} \\ \vdots \\ x_n^{(l-1)} \end{pmatrix} + \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \frac{b_3}{a_{33}} \\ \vdots \\ \frac{b_n}{a_{nn}} \end{pmatrix}$$

The Gauss-Seidel iteration is different from the linear expressions in the previous examples in that it is composed of two linear expressions of the form $E_1 : \bar{y} = A_1\bar{x}^{(l)}$ and $E_2 : \bar{y} = A_2\bar{x}^{(l-1)} + \bar{b}_2$. To optimize the iteration computation, the cost function is modified as follows

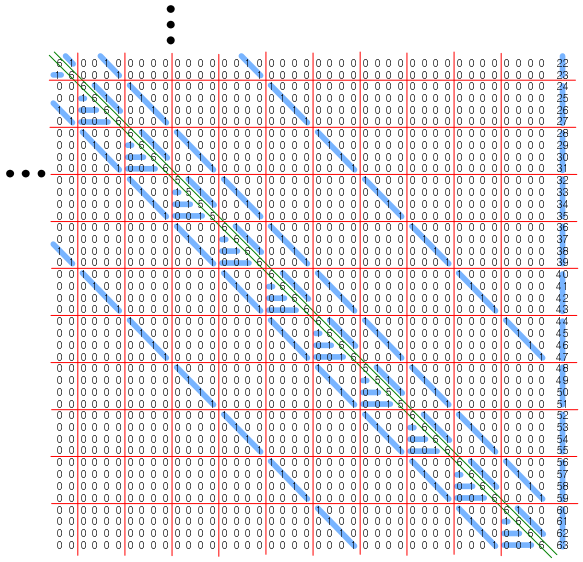
$$C(E_1,E_2) = \sum_{i,j=0}^{n-1} \{C_{E_1}^{mul}(i,j) + C_{E_2}^{mul}(i,j)\} + \sum_{i=0}^{n-1} \{C_{E_1}^{sum}(i) + C_{E_2}^{sum}(i)\}$$

In computing the first matrix-vector multiplication $A_1\bar{x}^{(l)}$, it is important that the elements of $\bar{x}^{(l)}$ be computed in sequential order. This requirement can be satisfied by selecting the row-major multiplication method for all diagonal submatrices A_{ii} of A. For this reason, $C_{E_1}^{mul}(i,j)$ in the cost function is slightly changed to

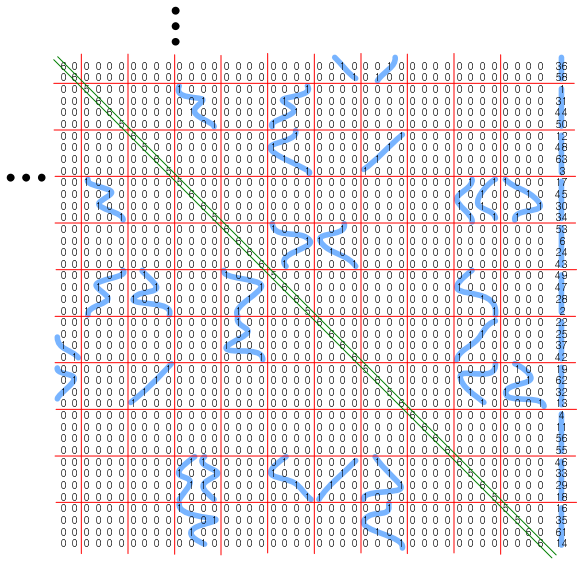
$$C_{E_1}^{mul}(i,j) = \begin{cases} \min(c_{E_1}(i,j), r_{E_1}(i,j)), & \text{if } i \neq j, \\ r_{E_1}(i,i), & \textit{otherwise.} \end{cases}$$

As a test, we applied our optimization technique to solving a Poisson equation that is generated in the process of enforcing the incompressibility condition $\nabla \cdot \boldsymbol{u} = 0$ of the Navier-Stokes equations for fluid animation[7]. Figure 7(a) shows a portion of the Gauss-Seidel iteration for a 64×64 linear system of equations, generated from the fluid-containing cells in a $10 \times 10 \times 10$ grid. For convenience, we put the three matrices A_1 (lower triangular), A_2 (upper triangular), and D (diagonal) in one matrix. Figure 7(b) shows an equivalent iteration obtained after the transformation stage.

In this example, the cost decreased from 112 SIMD instructions to 72 SIMD instructions. This high optimization effect is understood by observing the row-major multiplications that must be performed in an iteration computation before the transform (see the horizontal thick line segments in the diagonal submatrices of Figure 7(a)). Only one nonzero number appears in each segment, implying that only one meaningful multiplication is performed per `DP4` instruction. The inefficiency is evident, because a `DP4` instruction can carry out up to four multiplications and three additions



(a) Before transformation



(b) After transformation

Figure 7: *Transformation for the Gauss-Seidel method. Our SIMD optimization technique generates an equivalent linear expression that can be evaluated more efficiently. The number of necessary SIMD instructions per iteration drops from 112 to 72.*

simultaneously. On the other hand, this inefficiency disappears after the transformation process, as seen in Figure 7(b). Interestingly, each curved segment, corresponding to a MAD instruction, contains four non-zero coefficients. This entails the full exploitation of the SIMD parallelism of shader hardware.

3.4. Statistics for Vertex Shader Implementations

Table 2 summarizes the statistics for implementing vertex shaders for the described examples. Column (a) indicates the number of additions and multiplications that must be computed to evaluate the original linear expression. This column can be viewed as the cost when no SIMD parallelism is exploited at all. The next two columns (b) and (c) compare the numbers of SIMD instructions necessary for evaluating the initial and transformed linear expressions. We find that the transformation effort reduces the cost by 23 to 36 per cent.

Column (d) indicates the ratio of the total arithmetic operations (column (a)) to the vertex shader length (column (c)). This figure represents the average number of arithmetic operations that are performed per SIMD instruction. Recall that the DP4 and MAD instructions can carry out up to seven (three additions/four multiplications) and eight (four additions/four multiplications) arithmetic operations, respectively. Efficiency of the SIMD implementation of linear expression is determined by how cleverly the arithmetic operations are crammed into the SIMD instructions. The efficiency levels range from 6.4 to 8.0, indicating that our optimization technique generates very effective vertex shader codes. Interestingly enough, SIMD parallelism is fully utilized in the Gauss-Seidel example, and its efficiency turns out to be 8.0.

The last column (e) shows the number of instructions that are actually required to write a vertex shader program on NVIDIA's GeForce FX GPU. To use its vertex engine, all the input values, that is, the non-zero values of A, $\bar{x}$, and $\bar{b}$ of linear expression are loaded into the constant registers before the vertex program starts. These values are then supplied to its function unit while the linear expression is evaluated. A problem is that only one constant register may be read by a SIMD instruction on the GeForce FX GPU[19]. This limitation forces frequent data moves between constant registers and temporary (read/write) registers. For the 2D wave equation and the Gauss-Seidel examples, 18 and 15 extra MOV instructions were required, respectively. On the other hand, the extra 64 instructions in the Runge-Kutta example include these move operations, and the arithmetic operations for evaluating Eq. (4), (6), (8), and (10). Due to the extra MOV instructions, the actual efficiency is lower than that shown in column (d). However, we expect it to be enhanced when vertex shaders are implemented on the newer GPUs that are equipped with more temporary registers.

Finally, Figure 8 shows how the cost $C(E)$ decreases during the simulated annealing process, where three different

	(a)	(b)	(c)	(d)	(e)
2D wave equation	462	82	63	7.3	81
Runge-Kutta	360	80	56	6.4	120
Gauss-Seidel	576	112	72	8.0	87

Table 2: *Statistics on vertex shader implementations. (a) Number of total arithmetic operations, (b) & (c) Costs before and after the transform, (d) Efficiency = (a) / (c), (e) Number of instructions in the final vertex shader.*

sets of control parameters were tested. We observe that all three annealing schedules find the same minimum although the times taken to reach it for the first time vary. It appears that the search patterns are less affected by the schedules for the Runge-Kutta example. On the other hand, the difference is noticeable for the other examples. Recall that the search space (20!) of the first example is rather small compared to the search spaces (64!) of the latter examples. During the annealing process, since there are more possible ways to move around the larger spaces, the search process is more sensitive to the annealing schedules. In any case, we conjecture that the simulated annealing algorithm successfully found one of possibly many global minima.

3.5. Implementing the Optimized Vertex Shaders on NVIDIA's GeForce GPUs

So far, we have described how various numerical algorithms can be mapped onto *abstract* vertex shaders that support a simple SIMD parallelism. In implementing this optimization technique, we have used vertex state program offered by NVIDIA's GeForce GPUs. It shares the same SIMD instruction set as vertex program and have a similar execution model. Unlike the vertex program that is executed implicitly when a vertex transformation is provoked by drawing it, the vertex state program is executed explicitly, independently of any vertices.

The major purpose of the vertex state program is to compute various program parameters that are required for executing vertex programs. The computed values stay in program parameter registers, shared by both the vertex state program and the vertex program, and are then used appropriately by subsequently executed vertex programs. Unlike the vertex program, no vertex is put in the graphics pipeline. This execution model is well-suited to implementing our optimization technique. For instance, a vertex state program, containing the optimized differential equation solver explained in Subsection 3.2, is executed for each frame to compute the position y_0 and the orientation y_2 of the vehicle, and store them in program parameter registers. A vertex program, repeatedly called for each vertex of the vehicle model, can then perform modelling transformations appropriately using these parameters.

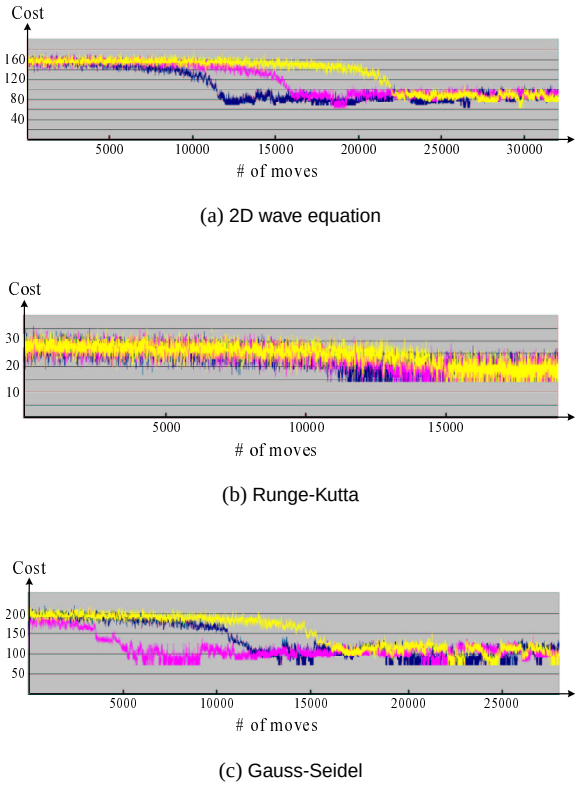


Figure 8: *Performance of the simulated annealing algorithm. Three different sets of control parameters were tested. It is conjectured that the simulated annealing algorithm successfully found a global minimum in these examples.*

When vertex programs or vertex state programs are executed, the data transfer between the CPU and the GPU must be minimized. In fact, it is often unnecessary to transfer the data computed by vertex state programs to main memory. As mentioned above, there is no need to move the computed position and orientation in the example of Subsection 3.2. Furthermore, it is possible to render the waves in Subsection 3.1 without transferring the computed z coordinates back to main memory. When each vertex of the wave is enumerated using the glVertex*(*) command, only x and y coordinates are described. At the same time, the index to the program parameter register that holds its z coordinate is supplied to the vertex program as a vertex attribute. The vertex program can then use the index as a relative offset for reading the z value from the proper program parameter register.

Implementations of vertex shaders with vertex state programs are currently limited because of the shortcomings of available vertex shader hardware. Such hardware generally supports a small number of available program parameter

registers, and simple control structures that do not support branching and looping. For instance, the vertex state program must be called explicitly for every iteration of the Gauss-Seidel method (presented earlier) because looping instructions are not supported by the current vertex state program. When the unoptimized and the optimized vertex shaders were used on an NVIDIA GeForce FX 5900 GPU solving a Poisson equation of size 64×64, they took 4.27 ms and 3.65 ms, respectively, for 174 iterations. Obviously, this is lower performance than modern CPUs, but such problems are expected to be relieved as vertex shader hardware becomes more flexible in future GPUs.

The SIMD optimization technique is not currently as effective for vertex shaders as it is for pixel shaders, yet it is promising for applications such as moving vehicle simulation in Subsection 3.2, which are inherently per-vertex oriented. We believe that efforts to optimize vertex shader codes whether it is on vertex programs or vertex state programs, are important, because to reduce the computation time and shorten the codes, allowing vertex shader codes for larger problems to fit into the vertex shader hardware (the vertex shader still limits the number of instructions in its codes; for example, 256 are permitted for `NV_vertex_program2` of NV30).

4. Applications to Pixel Shader Programming

The presented optimization technique can also be applied effectively for programming pixel shaders. An important architectural difference between vertex shaders and pixel shaders is that the latter offer fast access to texture memory, which can hold a large amount of data. To utilize the texturing function of the pixel shaders, we add another instruction `TEX` to the instruction set (shown in Table 1). Given texture coordinates and a texture image, this additional instruction fetches a correct texel value from texture memory. Unlike the technique presented in the previous section, the SIMD optimization technique can harness the fragment processing power of the pixel shader. In this section, we show a couple of examples which demonstrate how our optimization method is employed to enhance the performance of pixel shaders.

4.1. An Iterative Sparse Linear System Solver

As the first example, we apply the SIMD optimization technique to develop a pixel shader that solves Poisson equations, generated during the integration process of the Navier-Stokes equations. Although our Poisson equation solver has been developed for three-dimensional grids, we will first explain the solver in terms of a two-dimensional problem for the sake of simplicity.

4.1.1. Enhanced Block Jacobi Method for the Two-Dimensional Poisson Solver

Consider an 8×8 grid through which fluid is simulated. A sparse linear system $A\bar{x} = \bar{b}$ is obtained by discretizing the Poisson equation over the pressure term. A linear equation per grid point is produced, where the unknown variable x_{ij} denotes the pressure value at the (i, j)-th point:

$$\frac{1}{h^2}(x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1} - 4x_{ij}) \;=\; b_{ij},$$

for $i, j = 0, 1, 2, \cdots, 7$. Let $\bar{x} \in R^{64}$ be the vector of unknowns, where x_{ij}s are enumerated in row-major fashion, so index i varies faster.

The iteration in the well-known Jacobi method is then expressed as

$$\bar{x}^{(l)} = -D^{-1}(L+U)\bar{x}^{(l-1)} + \bar{b}_2,$$

where L, D, U, and $\bar{b}_2$ are defined in the same way as in Section 3.3. In the Gauss-Seidel method, the updated values of x_{ij} replace older values immediately. This property makes it difficult to implement the method on current pixel shader hardware, because of its inability to read and write the same texture memory location in the same pass[2, 9]. In contrast, the standard Jacobi method computes all new components of $\bar{x}$ before replacement, resulting in slower convergence of the Jacobi iteration. Our pixel shader technique, however, is faster than the standard Jacobi method, as will be explained shortly.

If two T blocks are added in the first and last block rows, the 64×64 matrix $-D^{-1}(L+U)$ of the two-dimensional Poisson equation can be represented as a repetition of a T-S-T block sequence (see Figure 9). This block-tridiagonal

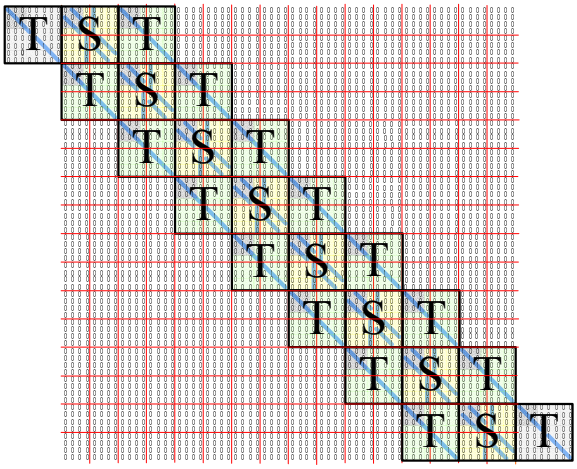


Figure 9: *The block structure of the matrix $-D^{-1}(L+U)$ in the Jacobi iteration. The matrix from the 2D Poisson equation exhibits a block-tridiagonal structure.*

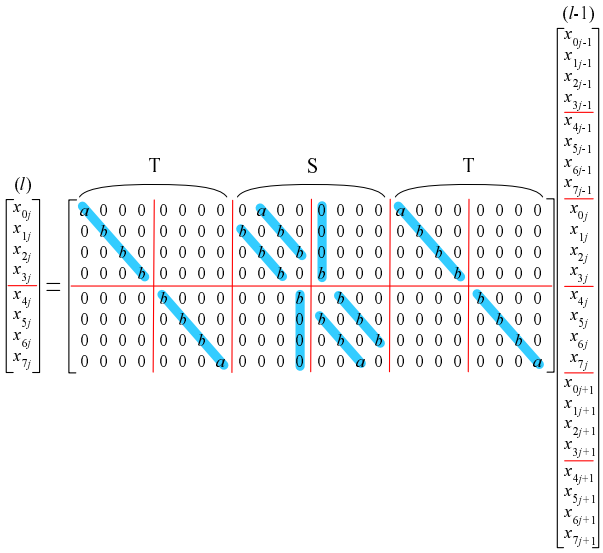
structure exhibits a very simple pattern in the matrix-vector multiplication $\vec{x}^{(l)} = -D^{-1}(L+U)\vec{x}^{(l-1)}$, which is highly desirable for the pixel shader implementation. Figure 10(a) illustrates an *elementary blockwise multiplication* involving the j-th block row, where two sets of artificial unknowns $\{x_{0,-1}, x_{1,-1}, x_{2,-1}, \cdots, x_{7,-1}\}$ and $\{x_{08}, x_{18}, x_{28}, \cdots, x_{78}\}$ are additionally assumed for $j = 0$ and $j = 7$.

As in the vertex shader implementation, each quadruple of $(x_{0j}, x_{1j}, x_{2j}, x_{3j})^t$ and $(x_{4j}, x_{5j}, x_{6j}, x_{7j})^t$, $j = 0, 1, 2, \cdots, 7$ is stored as a four-tuple of floating-point numbers. The figure shows that 10 shader instructions are required to implement the entire elementary blockwise multiplication. However, it is obvious that the SIMD processing power of pixel shader is not fully exploited when the S block is multiplied with the two quadruples.

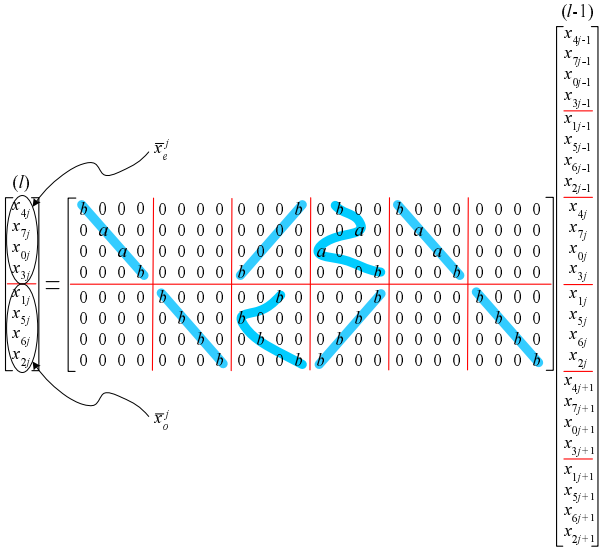
Applying our optimization technique to the 8×8 S block returns a more efficient SIMD computation that requires only four instructions instead of six (see Figure 10(b)). Notice that 30 scalar multiplications are actually carried out using 10 four-wide SIMD instructions before the optimization. Hence, only 75% ($=\frac{30}{10 \cdot 4}$) of the fragment processing power is used. On the other hand, the efficiency increased to 93.75% ($=\frac{30}{8 \cdot 4}$) after the optimization, where eight SIMD instructions were used.

The optimized elementary blockwise multiplication in Figure 10(b) becomes the fundamental computation kernel that is mapped to the pixel shader hardware. Let $\vec{x}_e^j$ and $\vec{x}_o^j$, $j = 0, 1, 2, \cdots, 7$ denote the two permuted four-tuples $(x_{4j}, x_{7j}, x_{0j}, x_{3j})^t$ and $(x_{1j}, x_{5j}, x_{6j}, x_{2j})^t$, respectively. Because the current pixel shader supports four-wide SIMD processing, each optimized multiplication must be performed using two fragments. One problem is that the expressions to compute $\vec{x}_e^j$ and $\vec{x}_o^j$ are not identical. Since the same kernel must be executed over all fragments of a stream in the current pixel shader, we have implemented the matrix-vector multiplication using two pixel shader programs, one for $\vec{x}_e^j$ and another for $\vec{x}_o^j$, so the unknown vector $\vec{x}$ is partitioned into two one-dimensional textures, as illustrated in Figure 11: `TEX_X_EVEN` stores all even-numbered quadruples $\vec{x}_e^j$, and `TEX_X_ODD` stores the remaining odd-numbered quadruples $\vec{x}_o^j$.

These two textures, and another one-dimensional texture `TEX_B` containing the correctly permuted $\vec{b}_2$ vector, are loaded into texture memory. To compute the new unknown vector $\vec{x}$ in the l-th Jacobi iteration, a line segment is rendered twice, bound to the texture images and the correct (alternately even-numbered or odd-numbered) shader program. In the first line drawing, all even-numbered quadruples of $\vec{x}^{(l)}$ are updated by performing the first part of the elementary blockwise multiplication and adding b_{ij} values. The updated unknowns are then used to calculate the remaining unknowns in the second line drawing. The immediate replace-



(a) Before transformation



(b) After transformation

Figure 10: *The elementary blockwise multiplication for the 2D Poisson equation ($a = -\frac{1}{3}$ and $b = -\frac{1}{4}$). As in the vertex shader optimization, the transformation on the S block finds a more efficient SIMD computation.*

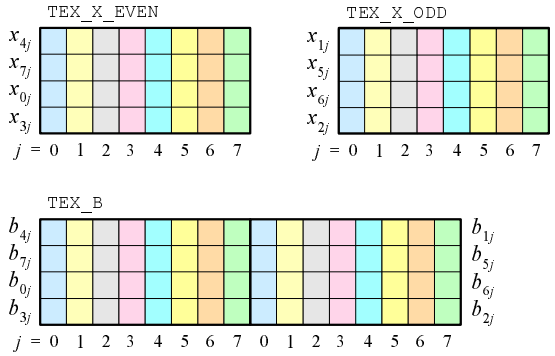


Figure 11: *Packing of data into 1D texture images. Because the evaluation of the elementary blockwise multiplication requires two computation kernels, one each for $\vec{x}_e^j$ and $\vec{x}_o^j$, the unknown vector $\bar{x}$ is partitioned into two textures* TEX_X_EVEN *and* TEX_X_ODD*. In order to perform a Jacobi iteration, a line segment is rendered twice as bound to the correct pair of kernel and texture alternately.*

ment of the unknowns provides a faster convergence of the iterations than that obtained for the standard Jacobi method.

We now discuss how boundary conditions of the Poisson equation are handled in our pixel shader implementation. Dirichlet conditions can easily be imposed by setting boundary values directly. The free Neumann condition $\frac{\partial \bar{x}}{\partial \bar{n}} = 0$ can be realized by copying the unknown values of the first and last rows and columns of the two-dimensional grid towards their respective exteriors. At the two vertical boundaries, this has a consequence that the diagonals in the Poisson equation $A\bar{x} = \bar{b}$, corresponding to the boundary unknowns, decrease by one. The matrix, shown in the example of Figure 10(a), has been constructed by imposing this partial condition, where the denominator for the first and last rows in the elementary blockwise multiplication is 3. However, the boundary condition at the horizontal boundaries has not been reflected in the elementary blockwise multiplication yet, as it would complicate the matrix and require additional pixel shader kernels. Instead, the horizontal boundary condition is imposed during the execution of the pixel shader programs. The unknowns of the first and last rows are stored in the four border texels of the two one-dimensional textures TEX_X_EVEN and TEX_X_ODD. When the unknowns in the exterior rows, whose texture coordinates are outside the texture range, are accessed with the texture wrap mode set to GL_CLAMP_TO_EDGE, the corresponding unknown values in the boundary rows are fetched. In this way, the boundary condition $\frac{\partial \bar{x}}{\partial \bar{n}} = 0$ in both horizontal and vertical directions can be satisfied using only two pixel shader programs.

4.1.2. Extension to Three-Dimensional Space

The method presented here is easily extended for solving three-dimensional Poisson equations. Consider, for instance, a $40 \times 80 \times 80$ grid, where the 256,000 unknowns x_{ijk}, $i = 0,1,2,\cdots,39$, $j,k = 0,1,2,\cdots,79$ are enumerated again in row-major fashion (see Figure 12). Let $\vec{x}^{jk} = (x_{0jk}, x_{1jk}, x_{2jk}, \cdots, x_{39jk})^t$, $j,k = 0,1,2,\cdots,79$ be vectors in R^{40} that partition the unknown vector $\bar{x}$. Then each $\vec{x}^{jk}$ represents a line in the grid that shares the same j and k indices. The (j,k)-th *elementary blockwise multiplication* in the Jacobi iterations is represented similarly as

$$\vec{x}^{jk} = T \cdot \vec{x}^{j-1,k} + T \cdot \vec{x}^{j+1,k} + T \cdot \vec{x}^{j,k-1} + T \cdot \vec{x}^{j,k+1} + S \cdot \vec{x}^{jk}.$$

Here, $T \in R^{40\times 40}$ is a diagonal matrix whose diagonal entry is $-\frac{1}{6}$ (except the first and last diagonals, which are $-\frac{1}{5}$), and $S \in R^{40\times 40}$ is a diagonal block as depicted in Figure 13(a), where the Neumann boundary condition $\frac{\partial \bar{x}}{\partial \bar{n}} = 0$ is already imposed at the two boundary planes perpendicular to the x-axis.

As before, our optimization technique finds a more efficient method of computing the matrix-vector multiplication. Figure 13(b) shows the matrix S after the optimization. In the three-dimensional case, 238 scalar multiplications (78 for S and 40 each for T) are needed to compute an elementary blockwise multiplication. Before the computation is optimized, 78 SIMD instructions (38 for S and 10 each for T) were required, and only 76.3% ($= \frac{238}{78\cdot 4}$) of the SIMD capability was utilized. After the transformation of S, only 60 instructions (20 for S and 10 each for T) are required for the same computation. The efficiency approaches 99.2% ($= \frac{238}{60\cdot 4}$), as the SIMD processor is now almost fully exploited. Notice that there are only two 'wasted' zeros in the thick line segments of the optimized S.

Unlike the two-dimensional Poisson solver, the unknowns in the three-dimensional grid are packed into two-dimensional textures. For the simplicity of notation, $\vec{x}^{jk}$ now denotes the group of unknowns whose elements have been permuted in the optimization process. We let $\vec{x}_{(i)}^{jk}$, $i = 0,1,2,\cdots,9$ be the quadruples $(x_{4i,jk}, x_{4i+1,jk}, x_{4i+2,jk}, x_{4i+3,jk})^t$ that further partition $\vec{x}^{jk}$ (refer to Figure 12 again). Because $\vec{x}^{jk}$ contains 10 such quadruples, 10 fragments must be processed to compute the entire (j,k)-th elementary blockwise multiplication. As in the two-dimensional solver, each fragment is calculated differently, so we need to run 10 separate pixel shader kernels, one for each $\vec{x}_{(i)}^{jk}$. This partition of the task forces the 256,000 unknowns to be stored in 10 two-dimensional texture images of size 80×80, where the i-th texture TEX_X_i ($i = 0,1,2,\cdots,9$) holds all 6,400 quadruples $\vec{x}_{(i)}^{jk}$, $j,k = 0,1,2,\cdots,79$. Figure 12 illustrates the correspondence between slabs of width 4 in the three-dimensional grid and texture images. In order to impose the Neumann boundary conditions for the remaining two directions, the texture wrap mode must be set to GL_CLAMP_TO_EDGE for both texture coordinates.

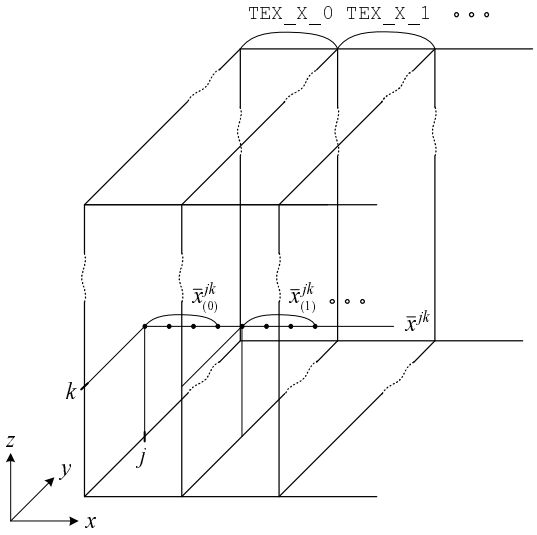
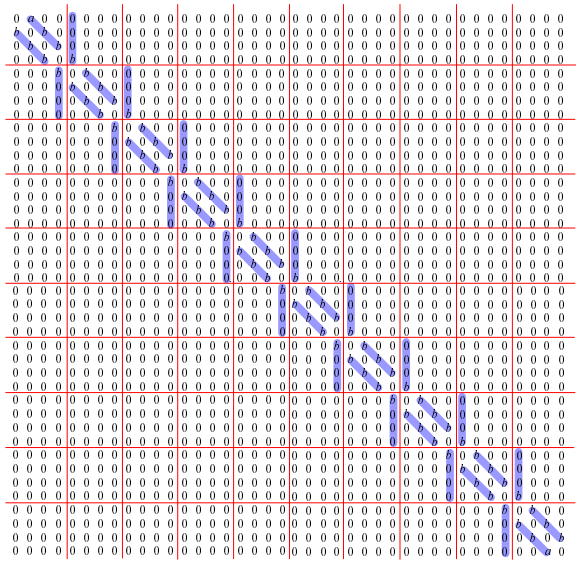


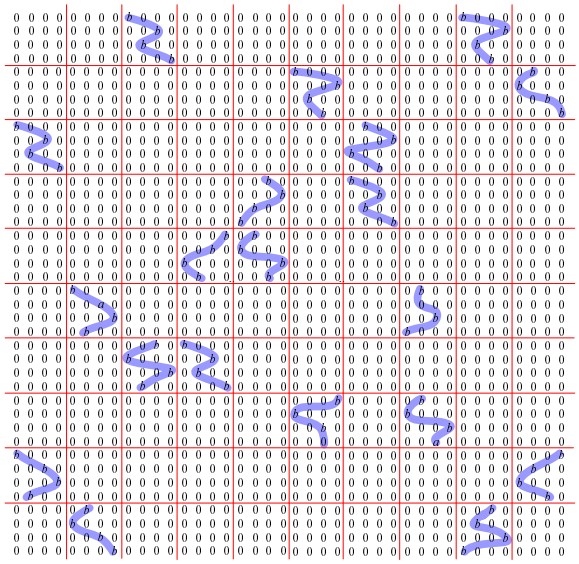
Figure 12: *Partition of 3D grid into 2D texture images. Each slab of width 4 is stored in a 2D RGBA texture. All unknowns in a slab are updated by executing a corresponding pixel shader kernel. The computed values update the unknowns immediately, and are used to compute the unknown values of the remaining slabs. These immediate updates within an iteration step lead to an enhanced convergence rate, compared with the standard Jacobi method.*

Once the shader kernels and textures are prepared, including one for the $\bar{b}_2$ vector, it is straightforward to solve the three-dimensional Poisson equation. In the l-th Jacobi iteration step, a rectangle is drawn 10 times. In every drawing, the correctly bound shader kernels update the unknowns, slab-by-slab. Note that the newly computed unknown values of a slab are immediately used to update the unknowns of the remaining slabs. These slabwise serial updates of the unknown vector within an iteration step enhance the convergence speed of the standard Jacobi method remarkably.

Our sparse linear system solver differs from previous solvers in that only the unknown vector $\bar{x}$ is loaded into texture memory for the matrix-vector multiplication. In the work of Bolz et al.[1], for instance, all nonzero entries of the matrix A and their indirection information are additionally loaded in texture memory. Their formulation is appropriate for *arbitrary* sparse matrices; however, many of the sparse matrices found in engineering applications are structured. In such cases, there is often no need to store all information in the matrix, as the nonzero entries can be 'melted' into the computation kernels instead. Our easy-to-implement pixel shader is optimized in both space and time since only the unknown vector is loaded into texture memory, and no additional indirection operations are required to access the matrix entries and the unknowns.



(a) Before transformation



(b) After transformation

Figure 13: *The diagonal block S of the matrix $-D^{-1}(L+U)$ in the enhanced block Jacobi solver for 3D Poisson equations ($a=-\frac{1}{5}$ and $b=-\frac{1}{6}$). The matrix transformation reduces the cost of SIMD instructions from 38 to 20. Through the optimization effort, we were able to utilize the SIMD capacity of pixel shader hardware with an efficiency of 99.2/elementary blockwise multiplication. Notice that only two 'wasted' zeros are found in the thick line segments of the optimized S.*

		tolerance		
		10^{-3}	10^{-4}	10^{-5}
GS on CPU	time	101.6	364.8	1178.6
	iter	16	61	200
EBJ on GPU	time	62.8	142.4	419.1
	iter	20	84	291
EBJ on GPU (optimized)	time	49.1 (9.8)	111.6 (68.3)	315.9 (272.6)
	iter	19	77	253

Table 3: *Comparison of timing performances. The computation times in milliseconds (time) spent solving a Poisson equation of size* $256,000 \times 256,000$ *are given with the numbers of iterations (iter) required to achieve the specified precisions (tolerance). The figures in parentheses are the pure computation times spent by the optimized enhanced block Jacobi solver. They exclude the times for data transfers between the CPU and the GPU that must be made to store the initial vector in the GPU and to read the solution vector back into the CPU.*

4.1.3. Comparisons between CPU and GPU Implementations

To verify the effectiveness of the presented optimization technique, we have implemented the enhanced block Jacobi method on an NVIDIA's GeForce FX 5900 Ultra GPU. We have also implemented the Gauss-Seidel method on an Intel's 2.66 GHz Pentium 4 CPU. We applied these GPU and CPU techniques to solving the Poisson equation that arises in the projection step of Stam's solver for incompressible viscous Navier-Stokes equations[31]. Although we have described the GPU technique using the Neumann boundary conditions, it is possible to impose either Dirichlet or Neumann conditions to each direction in three-dimensional space. In this experiment, zero Dirichlet conditions were imposed on the boundary plane perpendicular to the *x*-axis, and zero Neumann conditions were applied to the other two directions. We tested Poisson equations of size $256,000 \times 256,000$ with a $40 \times 80 \times 80$ grid.

Table 3 compares the performance of three different implementations of the Poisson solver. First, the Gauss-Seidel method was coded in highly optimized software (GS on CPU). The enhanced block Jacobi method was programmed with and without applying our SIMD optimization technique (EBJ on GPU (optimized) and EBJ on GPU, respectively). To measure the computation time in milliseconds (time), we have evolved the Navier-Stokes equations in time 1,000 times, and averaged the timings required to solve the 1,000 generated equations. In each time frame, the respective solver of each implementation was iterated until a target tolerance was achieved.

The Gauss-Seidel method requires 20 to 30 percent fewer iterations (iter) than the enhanced block Jacobi method in this experiment. However, the experimental results indicate that the 'slower' enhanced block Jacobi method without a SIMD optimization (EBJ on GPU) runs faster on the GPU than the 'faster' Gauss-Seidel method (GS on CPU) does on the CPU. They also reveal that a considerable enhancement in the GPU implementations is obtained when the presented SIMD optimization technique is applied (for example, when comparing the timings of EBJ on GPU and EBJ on GPU (optimized)). We find that about 23 per cent of the computation time is saved on average. It is interesting to note that the optimized GPU implementation demanded slightly fewer iterations to satisfy the same tolerance constraint. For instance, the optimized implementation required only 253 iterations to achieve a tolerance of 10^{-5}, whereas the unoptimized implementation required 291 iterations. As discussed, our optimization technique packs the necessary arithmetic operations of the solver more efficiently into the four-wide SIMD instructions of the GPU. The new arrangement of arithmetic operations produced by the SIMD optimization process produces a more effective ordering of computations that leads to a faster convergence. As a result, the number of required SIMD instructions per iteration is reduced, and the convergence rate is accelerated.

The timings for the two GPU implementations include the data transfer time between the CPU and the GPU. In the initialization stage, the 10 texture images containing the $256,000$ elements of $\vec{x}^{(0)}$ must be moved from the CPU to the GPU. When the Poisson solver finishes iteration, another data transfer from the GPU to the CPU must be performed to get the solution vector back. The figures in parentheses indicate the pure computation times spent by the solver, yet exclude the data transfer time, which is about 40 ms on average in this experiment. Notice that this extra time for data transfer can be saved if the entire solver of the Navier-Stokes equations is implemented completely on a GPU, or at least amortized over the diffusion step that precedes the projection step in Stam's solver. Our optimized GPU implementation can be easily modified to solve the linear equations from the diffusion step. Usually, these equations are more diagonally dominant, therefore only a few (5 to 10) iterations are sufficient to achieve a high precision.

4.2. Applying the SIMD Optimization Technique to Other Problems

One of the most critical factors in mapping an algorithm onto pixel shaders is the design of an efficient texture storage and access pattern. As a result, pixel shader programming tends to be more specific to the problem than vertex shader programming. The implementation of the enhanced block Jacobi method on pixel shaders turned out to be somewhat complicated as the fragment processing units are fully utilized. However, it is clear that the GPU provides a major speedup over the CPU implementation, and that applying

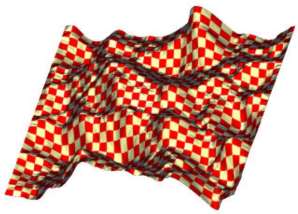
the SIMD optimization technique results in a considerable improvement in timing performance for GPU implementations.

The SIMD optimization technique presented here is very well suited to problems that are based on repeated evaluations of linear expressions. As another example, recall the two-dimensional wave equation (Eq. (3)), described in Subsection 3.1. We have simulated waves on a 512×512 grid, from which a wave matrix of size $262,144 \times 262,144$ is generated. In this experiment, we imposed zero Dirichlet conditions in one direction, and zero Neumann conditions in the other direction, where the application of a similar mapping technique as used in the previous subsection also produces an efficient GPU implementation. Here, we give only a brief description, without repeatedly explaining what is basically the same implementation technique.

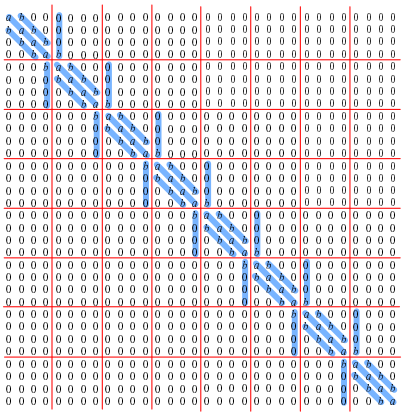
The wave matrix is sparse and has a simple structure, and can be, as before, represented as a repetition of a T–S–T block sequence of size 512×512. Instead of applying our SIMD optimization technique to the somewhat large diagonal S block, we have subdivided it into a sequence of diagonal subblocks S^* of size 32×32 (see Figure 14(b)). After the elementary block S^* is optimized, we find that the SIMD cost reduces from 38 to 24 (see Figure 14(c)). Two coefficients around each corner of S^* are missing, as the S block is partitioned into the S^* blocks. In coding pixel shaders, two extra SIMD instructions are additionally required to handle them, for a total of 26 SIMD instructions after the optimization. Because the T block is a simple diagonal matrix, two partitioning subblocks T^* of size 32×32 can be multiplied using 16 SIMD instructions. Hence, the total number of SIMD instructions for an elementary blockwise multiplication of size 32×32 decreases from 56 to 42 through the optimization effort. Note that the entire $262,144$ variables containing the heights at the grid points are stored similarly as four-tuples in eight two-dimensional texture images, which are updated in each time step using eight separate pixel shader kernels.

The timing measurements summarized in Table 4 again show a quite favorable result for both the GPU implementations and the SIMD optimization technique. Here, the figures represent the computation times in milliseconds for a single integration of the wave equation, averaged over the first 2000 time steps. In the GPU implementations, the height values of simulated waves are stored in texture memory, hence a data transfer from the GPU to the CPU is necessary per time step to move the newly computed values. Currently, this overhead is unavoidable on most GPUs, because the height information must be read back to the CPU for rendering and further processing. In spite of the extra data transfer time, the GPU implementations are shown to be faster than the CPU implementation (compare the timings in the row Total).

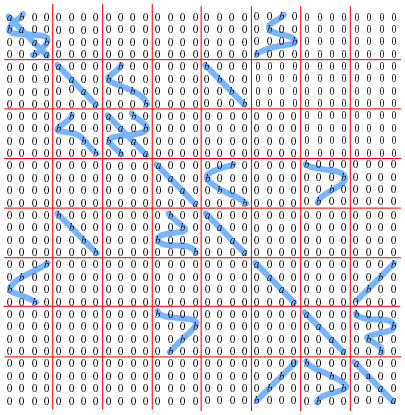
The performance could improve further if it was possible to bind the render target to memory objects like ver-



(a) A snap shot



(b) Before transformation



(c) After transformation

Figure 14: *The simulation of two-dimensional waves. The tested* 512×512 *grid offers a very detailed wave simulation. The matrix transformation again demonstrates the effectiveness of the SIMD optimization technique as in the previous examples. The* 32×32 S^* *blocks are shown before and after the transformation. The cost reduces from 38 to 24 (*$a = 2(1-2\lambda^2)$ *and* $b = \lambda^2$*).*

	On CPU	On GPU	On GPU (optimized)
Total	17.73	8.34	8.08
No data transfer	–	1.28	1.02

Table 4: *Comparison of timing performances. The computation times in milliseconds per time step to integrate the two-dimensional wave equation over a* 512×512 *grid are summarized. The figures were measured by averaging the times spent updating the 262,144 variables over the first 2000 time steps. While the data transfer from the GPU to the CPU clearly harms the performance, this problem is expected to be alleviated in the near future. In any case, the experiment reveals a very favorable result for both the GPU implementations and the SIMD optimization technique.*

tex arrays, as their data can be reinjected into the geometry pipeline directly without data transfers, as proposed recently in the uber buffers extensions[20]. When the data transfer time is excluded, the performance enhancement of the GPU over the CPU is remarkable as indicated in the row No data transfer. It is also evident that the SIMD optimization technique presented here provides a considerable speedup for the GPU implementations. We believe that our SIMD optimization technique can be used effectively in a wider range of graphics-related problems, as GPUs evolve to support more flexible framebuffer architectures in the near future.

5. Conclusion

Linear expressions appear frequently in many scientific and engineering areas, including computer graphics, and their efficient evaluation is often critical in developing real-time applications. We have presented a SIMD code optimization technique that enables efficient codes to be produced for evaluating linear expressions. In particular, our technique exploits the four-wide SIMD computing capacities offered by modern GPUs. We have demonstrated that the new technique can be effectively applied to solving a variety of general mathematical computations on a GPU. Although our emphasis was on code optimization for the GPU, the ideas are valid for any processing unit that supports the simple SIMD processing model as described.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by grant No. R01-2002-000-00512-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

References

1. J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.
2. N. Carr, J. Hall, and J. Hart. GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 51–59, 2003.
3. N.A. Carr, J.D. Hall, and J.C. Hart. The ray engine. In *Proceedings of Graphics Hardware 2002*, pages 1–10, 2002.
4. V. Cerny. Thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Application*, 45:41–51, 1985.
5. M.F. Cohen and J.R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Inc., 1993.
6. K. Engel and T. Ertl. Interactive high-quality volume rendering with flexible consumer graphics hardware. In *Proceedings of Eurographics 2002 - STAR Report*, pages 109–116, 2002.
7. N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of ACM SIGGRAPH 2001*, pages 23–30, 2001.
8. N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 102–111, 2003.
9. M. Harris, W. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 92–101, 2003.
10. M.J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 1–10, 2002.
11. J. Hart. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 87–94, 2001.
12. W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 145–148, 1999.
13. K. Hillesland, S. Molinov, and R. Grzeszczuk. Nonlinear optimization framework for image-based modeling

on programmable graphics hardware. *ACM Transactions on Graphics*, 22(3):925–934, July 2003.

14. W.J. Palm III. *Modeling, Analysis, and Control of Dynamic Systems*. John Wiley & Sons, Inc., 2nd edition, 2000.
15. N. Kapadia and J. Fortes. Block-row sparse matrix-vector multiplication on SIMD machines. In *Proceedings of 1995 International Conference on Parallel Processing*, volume III, pages 34–41, 1995.
16. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
17. J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, July 2003.
18. E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing 2001*, 2001.
19. E. Lindholm, M.J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, pages 149–158, 2001.
20. R. Mace. OpenGL ARB Superbuffers. Game Developers Conference 2004, 2004.
21. C. Maughan and M. Wloka. Vertex shader introduction. NVIDIA Technical Brief, 2001.
22. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.
23. K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 112–119, 2003.
24. A.T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, 1993.
25. M.S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, pages 425–432, 2000.
26. K. Proudfoot, W.R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, pages 159–170, 2001.
27. T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 41–50, 2003.
28. T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2002*, pages 703–712, 2002.
29. L.F. Romero and E.L. Zapata. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, 21(4):583–605, 1995.
30. M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings of IASTED International Conference Visualization, Imaging, and Image Processing*, pages 160–170, 2001.
31. J. Stam. Stable fluids. In *Proc. of ACM SIGGRAPH 1999*, pages 121–128, 1999.
32. J.C. Tannehill, D.A. Anderson, and R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis Publishers, 2nd edition, 1997.
33. C. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings 35th International Symposium on Microarchitecture (MICRO-35)*, November 2002.
34. M.P. Vecchi and S. Kirkpatrick. Global wiring by simulated annealing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-2:215–222, 1983.
35. D. Weiskopf, M. Hopf, and T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proceedings of Vision, Modeling, and Visualization 2001*, pages 439–446, 2001.
36. D.F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic Publishers, 1988.
37. L.H. Ziantz, C.C. Ozturan, and B.K. Szymanski. Runtime optimization of sparse matrix-vector multiplication on SIMD machines. In *Parallel Architectures and Languages Europe*, pages 313–322, 1994.