3D GRAPHICS OPTIMIZATIONS FOR ARM ARCHITECTURE

Gopi K. Kolli†
gopi.k.kolli@intel.com

Stephen Junkins‡
mailto:stephen.junkins@intel.com

Haim Barad†
mailto:haim.barad@intel.com

†Handheld Computing Division
‡Emerging Platforms Lab
Intel Corporation

# Introduction

- Floating-Point Systems Vs Fixed-Point Systems
  - Floating-Point Systems
  - Hardware Coprocessor
  - Floating-point Library
- Fixed-Point System
  - Arithmetic Operations
  - Dynamic Range and Precision
  - Error Checking
  - Arithmetic Approximation Routines
  - Trigonometric functions
- Integer Divide
- Branching and Predication
  - Branching
  - Predication
- Invoking Predication
  - Loops
  - “If” statements
  - Relational or Boolean expression
- Register Allocation
  - Pointer Aliasing
  - Function call overhead
- Memory-Based Optimizations
- Conclusion
- References
Introduction

Embedded and handheld computing devices are rapidly becoming ubiquitous. They are evolving in usage, performance and features and are becoming capable of supporting 3D graphics. The computational performance and display capabilities of these consumer devices are evolving rapidly. Compaq’s iPaq 3800 handheld computing devices has a 206Mhz Intel® StrongARM™ Microprocessor and a 16-bit QVGA display. With such capabilities, handheld computing devices, set-top boxes and even cell phones can now be programmed to support software rendering of immersive 3D Worlds. Once developed, a 3D rendering solution coupled with wireless connectivity capability, and the growing ubiquity of mobile computing devices, provides an exciting new opportunity for 3D game developers.

Many mobile devices such as cell-phones, personal digital assistants and handheld gaming devices use ARM-based processors. ARM architecture is a 16/32-bit RISC architecture designed to allow very small, yet high-performance implementations for low power devices and is becoming an architecture standard for handheld, multi-media computing. Though ARM processor instruction throughput has recently become quite attractive, other aspects of the architecture challenge implementers of software 3D Rendering systems. Specifically:

- Many commercial ARM-based devices do not include dedicated floating-point hardware due to extra cost and power consumption issues.
- ARM architecture does not support integer divide.
- For most ARM implementations, on chip caches are quite small relative to PC architecture caches sizes.
- Display hardware is small and very simple; usually LCD controller memory maps system memory.
- 2D and 3D Rasterization hardware is not commonplace in embedded devices. Cost and power consumption will likely limit the acceptance of dedicated hardware in the future, especially for cell phones. However, leading edge PDAs might accept it for a premium price.

Given these architectural challenges, careful optimization of the 3D engine is the key to achieving rendering performance sufficient for 3D games on ARM-based platforms. In this paper, we will explore these challenges and suggest performance optimization strategies to enable game developers to build software 3D Rendering solutions for ARM-based embedded devices.

Floating-Point Systems Vs Fixed-Point Systems

Flexible 3D engines require real number representation of coordinate space systems to support many of 3D Rendering’s fundamental algorithms. Real number representation is especially relevant for implementation of transform, lighting, clipping, and culling, as they require broad dynamic range and a high degree of precision. Floating-point representation of real numbers is preferred to integer representation due to its ability to provide large dynamic range and very high precision.

Floating-Point Systems

Floating-point support can be provided in ARM-based systems either in hardware or in software.

Hardware Coprocessor

Hardware floating-point implementation typically consists of a floating-point coprocessor and provides very good performance. However, placing additional silicon and power consumption costs on a commercial system is prohibitive. Additionally, the hardware coprocessor limits the performance of the ARM code with its maximum clock speed. Therefore, this implementation is not preferred currently in commercial ARM-based systems.

Floating-point Library

Software floating-point implementation typically consists of a floating-point library. Floating-point operations can be fully implemented in a software library using ARM instructions. While compiling the floating-point application code, the compilers generate function calls to this software library rather than floating-point instructions. Therefore, the application code

- Remains unaffected with future inclusion of floating-point hardware into the system.
- Can instantly take advantage of any improvement in the ARM core.
Choice of the floating-point support in the system depends on various factors such as performance, system cost and system flexibility. ARM inc. recommends use of a floating-point library in embedded systems. [2]

However, a better approach would be to replace floating-point operations with integer operations by using fixed-point representation of real numbers. If the dynamic range and precision requirements can be constrained, fixed-point arithmetic can be more accurate and much faster than floating-point arithmetic.

**Fixed-Point System**

Fixed-point is a way to represent a floating-point number in an integer format with an imaginary decimal point dividing the integer and fractional part. The bits to the right of the imaginary point comprise the fractional portion of the value being represented, and these bits act as weights for negative powers of 2. The bits to the left of the imaginary point comprise the integer portion of the value being represented, and these bits act as weights for positive powers of 2. [4]

For example, a 16.16 fixed-point format specifies that there are 16 bits before and after the decimal point. Therefore, in case of signed numbers, the dynamic range spans the open interval \([-2^{15}, 2^{15}]\) and the precision is \(2^{-16}\). The value represented by this fixed-point format is given by the adding up the weighted powers of 2:

\[
\text{value} = -b_{31}2^{15} + b_{30}2^{14} + ... + b_{17}2^1 + b_{16}2^0 + b_{15}2^{-1} + ... + b_{2}2^{-14} + b_{1}2^{-15} + b_02^{-16}
\]

where \(b_i \in \{0, 1\}\) are the binary bits.

Similarly, for unsigned numbers, the dynamic range spans the open interval \([2^{16}, 0]\) and the precision is \(2^{-16}\). The value represented by this fixed-point format is given by the adding up the weighted powers of 2:

\[
\text{value} = b_{31}2^{15} + b_{30}2^{14} + ... + b_{17}2^1 + b_{16}2^0 + b_{15}2^{-1} + ... + b_{2}2^{-14} + b_{1}2^{-15} + b_02^{-16}
\]

where \(b_i \in \{0, 1\}\) are the binary bits.

**Arithmetic Operations**

Arithmetic operations on fixed-point numbers require certain numerical adjustments. Addition and subtraction are simple. However, the result should be scaled down for multiplication and scaled up for division. Consider the following example.

\[
a = 5.0 \times 2^{16} \\
b = 2.5 \times 2^{16}
\]

Multiplying \(a\) and \(b\) gives us

\[
= (5.0 \times 2^{14}) \times (2.5 \times 2^{14}) \\
= (5.0 \times 2.5 \times 2^{11}) \times 2^{14}
\]

So, to properly format the result in a 16.16 format, the result should be divided by \(2^{16}\).

Dividing \(a\) by \(b\) gives us

\[
= (5.0 \times 2^{14}) / (2.5 \times 2^{14}) = 2.0
\]

So, to properly format the result in a 16.16 format, the result should be multiplied by \(2^{16}\).

Also, it is important to note that the intermediate value extend beyond 32-bit space causing possible data loss.

**Dynamic Range and Precision**

In Fixed-point representation of real numbers, dynamic range and precision are complementary to each other. This poses a unique problem during arithmetic operations.

Consider a multiplication between two unsigned numbers in a 16.16 fixed-point format to get a result in a 16.16 format. Both the multiplicands and the result can be represented in integer space. However, during multiplication, a 64-bit intermediate result in a 32.32 format is produced. The lower 32 bits of the result correspond to the fractional part of the result and the higher 32 bits correspond to the integer part. ARM architecture provides instructions (SMULL, SMLAL, UMULL, UMLAL, etc..) that support 64-bit intermediate results on all arithmetic operations. Therefore care should be taken to use proper data types in C and C++ that would notify the compilers to invoke these special instructions.
```c
int Mul(int a, int b)
{
    return (int) (((__int64) a * (__int64) b) >> 16);
}
```

While 64-bit intermediate results can prevent loss of data, they cannot increase the dynamic range and precision of the numbers. In the previous example, the maximum value that can be represented in the result is in a 16.16 format i.e., the sum of the integer bits of the multiplicands cannot exceed 16. This further brings down the range of the multiplicands. The programmer should constantly keep track of the changes in the range and precision of the dataset to avoid any risk of over-flows and under-flows.

Therefore, the developer should determine the balance between dynamic range and precision based upon the context of the application and the complexity of the 3D content. Also, in a 3D engine, overflow of dynamic range can be avoided by:

- Constraining the world inputs.
- Using localized spaces or normalizing data.
- Allocating less dynamic range during Rasterization stage as it is constrained to a small screen space.

Similarly, underflow of precision can be avoided by:

- Normalizing small vectors for subsequent operations
- Using alternate precision formats for Depth buffers and look-up tables used for math approximation routines.

### Error Checking

3D graphics applications are data-intensive, causing frequent underflow and overflow of data. Also, due to the nature of fixed-point representation, exceptional handling supported by the operating systems for embedded devices might not be suitable. Therefore, an error checking method with status flags should be provided to handle these situations. Also, error checking should be preferably done on the 64-bit intermediate values.

Consider the following example of a multiplication between two numbers a, b in a 16.16 format. Any data represented in the bits 48-63 (other than sign-extension) is considered to be data overflow and any data represented in the bits 0-15 is considered to be data underflow.

```
\[
\begin{array}{c}
a \\
<31 \quad 16 \quad | \quad 15 \quad 0>
\end{array}
\]
```

```
\[
\begin{array}{c}
b \\
<31 \quad 16 \quad | \quad 15 \quad 0>
\end{array}
\]
```

```
\[
\begin{array}{c}
a \times b \\
<31 \quad 16 \quad | \quad 15 \quad \cdots \quad \cdots \quad 0>
\end{array}
\]
```

```
\[
\begin{array}{c}
\leftarrow \text{Overflow} \rightarrow \mid \quad 47 \quad \cdots \quad 16 \mid \quad \leftarrow \text{Underflow} \rightarrow
\end{array}
\]
```
int Mul (int a, int b) {
    __int64 nVal;
    nVal = ((__int64)a * ((__int64)b);

    #ifdef _DEBUG
    if (nVal > (pow(2, 47) - 1)) return ERR_OVER_FLOW;
    if (nVal < (1 - pow(2, 47))) return ERR_UNDER_FLOW;
    #endif

    return (int) nVal;
}

Error checking in fixed-point representation is therefore expensive due to the requirement for 64-bit intermediate value.

**Arithmetic Approximation Routines**

Arithmetic operations such as division and square root are widely used in 3D graphics programming. They are computationally intensive and are supported by calling a C-function library. Considering the limited display properties of the embedded devices, an approximate version of these arithmetic functions would be adequate in certain stages of the graphics. An efficient software implementation of these approximation routines should:

- Consume less CPU cycles and provide high accuracy.
- Avoid lookup tables, where possible, thereby reducing memory access.
- Minimize code, increasing the likelihood of being in the Instruction Cache (IC).

The following comparative data shows the CPU cycle consumption for floating-point, 64-bit integer, 32-bit integer and optimized hand-coded ARM assembly versions of the most common arithmetic operations in 3D graphics. It should be noted here that the 32-integer division is not acceptable for fixed-point division, since either the entire dynamic range or entire precision is lost, depending upon whether the scaling is done before the or after the operation.

<table>
<thead>
<tr>
<th></th>
<th>Floating-Point S/W Emulation</th>
<th>64-bit Integer</th>
<th>32-bit Integer</th>
<th>Optimized ARM Assembly Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Division</strong></td>
<td>524</td>
<td>650</td>
<td>180</td>
<td>235</td>
</tr>
<tr>
<td><strong>Square Root</strong></td>
<td>4855</td>
<td>4855</td>
<td>4867</td>
<td>215</td>
</tr>
</tbody>
</table>

Note: Data for floating-point S/W emulation, 64-bit and 32-bit integer versions was collected using the ARM compiler in MS Embedded Visual C++ 3.0 IDE. Data for optimized ARM library version was collected using Intel® Graphics Performance Primitives 1.00 for the Intel® XScale™ Microarchitecture.
Trigonometric functions

Trigonometric functions, even if approximated, are expensive operations. However, they can be efficiently implemented using lookup tables. A single lookup table of a complete sine loop can be used for both sine and cosine computation. The loop can be broken down into a power of 2 intervals for easy masking. The size of the table depends upon the resolution requirements. For example, a table of 1024 entries has a resolution of 0.0061 radians and requires about 4K of memory.

```c
void CosSin(int theta, int *Cos, int *Sin)
{
    theta &= (TRIG_TABLE_SIZE - 1);
    *Sin = tblSin[theta];
    theta += [(theta + TRIG_TABLE_SIZE >> 2)];
    *Cos = tblSin[theta];
}
```

As a memory-based optimization, the table size can be further reduced by storing the values for the first quadrant and using logic instead to compute the values for other quadrants. Also, for better accuracy, the values in the lookup table can be efficiently stored with high precision such as a 4.28 format.

Integer Divide

The ARM instruction set does not provide integer divide operation. Therefore, compilers support division in the application code by calling a C-library function. Division is an expensive operation and therefore should be avoided where possible.

Consider the following example of vector normalization in a 16.16 format.

```c
v.x = (int)((__int64)(v.x << 16) / (__int64)length);
v.y = (int)((__int64)(v.y << 16) / (__int64)length);
v.z = (int)((__int64)(v.z << 16) / (__int64)length);
```

If more than one number is divided by the same value, then the reciprocal of the divisor should be computed and the numbers should be multiplied by it.

```c
inv_len = (1 << 16) / length;
v.x = (int)((__int64)v.x * (__int64)inv_len);
v.y = (int)((__int64)v.y * (__int64)inv_len);
v.z = (int)((__int64)v.z * (__int64)inv_len);
```

Replacing the additional divisions with multiplications will improve the performance dramatically. However, it will also introduce an additional rounding error.

When a number is to be divided by a known constant, rewrite it as a constant multiplication. Consider the following example of computing the centroid of a triangle in a 16.16 format.

```c
v.x = (v1.x + v2.x + v3.x ) / 3;
v.y = (v1.y + v2.y + v3.y ) / 3;
v.z = (v1.z + v2.z + v3.z ) / 3;
```

This can be rewritten as a constant multiplication with 0x5555 (1/3 in a 16.16 format).

```c
v.x = (int)(((__int64)(v1.x + v2.x + v3.x ) * (__int64)0x5555) >> 16);
v.y = (int)(((__int64)(v1.y + v2.y + v3.y ) * (__int64)0x5555) >> 16);
v.z = (int)(((__int64)(v1.z + v2.z + v3.z ) * (__int64)0x5555) >> 16);
```

Also, where possible, use powers of two as the divisor as it is basically a shift operation. An overflow adjustment is required for signed division though.

Perspective correction requires a division operation for each pixel. Considering the overhead of a division operation and the display characteristic features of embedded devices, care should be taken to avoid perspective correction whenever possible. For instance, the impact of perspective correction is negligible in small triangles.
Branching and Predication

Branching

Instruction-level parallelism improves the performance of the processors. However, branches limit it for several reasons:

- Branches cause pipeline stalls.
- Branch misprediction can hinder performance.
- Branches impose control dependences
- Branches complicate compiler optimization and scheduling.

Predication

Predicated execution is an architectural feature used to exploit instruction-level parallelism in the presence of control flow. It refers to conditional execution of instructions based on the Boolean value of a source operand called ‘predicate’. Compiling for predicated execution involves converting program control flow into conditional instructions. When all instructions that are control-dependent on a branch are predicated using the same condition as the branch, that branch can legally be removed. This reduces branch control dependencies and possible misprediction penalties. [1]

Predication support can be categorized as follows:

- Full Predication: Architectures (IA64, HP-PA, ARM, etc.,) in which all of the instructions are predicated.
- Partial Predication: Architectures (ELF, etc.,) in which only one or two predicated instructions are available such as a conditional move.

Consider the following example.

```plaintext
if ( a > b ) c = a;
else c = b;
```

The code generated for the if-else portion of this code segment using branches is:

```plaintext
CMP R0, R1
BLE _LessThan
MOV R2, R0
B _NextInstruction
_LessThan:
MOV R2, R1
_NextInstruction:
```

Here is the same code generated using conditional statements:

```plaintext
CMP R0, R1
MOVT R2, R0
MOVL R2, R1
```

Conditional statements reduce the number of branches and labels in the code. However, depending upon the operations in the branches, it might result in increased code size. Therefore, compilers use various heuristics to ensure a balance between performance improvement and code size. Simplest being, if the ‘if and else’ blocks have equal probability of executing and the total sum of cycles required is less than the branch misprediction penalty, conditional codes can be used.
Invoking Predication

Loops

3D Graphics applications spend a significant amount of time in loops, especially when the 3D engine supports multi-pass vertex processing. The loop termination condition can cause significant overhead in a loop and can be optimized as follows:

Consider the following example of transforming an object using an incrementing loop.

```c
for ( int i = 0; i < nPoly; i++) { ... }
MOV R0, #0
MOV R1, nPoly
Transform_NextPolygon:
    ...
    ADD R0, R0, #1
    CMP R0, R1
    BLT Transform_NextPolygon
```

Using a decrementing loop and making the loop-exit condition check against the value 0:

```c
for ( int i = 10; i != 0; i--) { .... }  
MOV R0, nPoly
Transform_NextPolygon:
    ...
    SUBS R0, R0, #1
    BNE Transform_NextPolygon
```

Therefore, a slight change in the loop logic resulted in better usage of predication and improved performance.

“If” statements

Conditional execution is applied mostly in the body of “if” statements. It is therefore beneficial to keep the bodies of “if” and “else” statements as simple as possible.

Relational or Boolean expression

A common Boolean expression during Rasterization stage is to check if a screen co-ordinate lies within the display screen limits:

```c
bool PointInRect(Point p)  
{ return (p.x >= XMIN && p.x < XMAX && p.y >= YMIN && p.y < YMAX);  
}
```

This can be optimized by modifying \((x >= \text{min} && x < \text{max})\) to \((\text{unsigned})(x-\text{min}) < (\text{max}-\text{min})\).

```c
bool PointInRect(Point p)  
{ return ((\text{unsigned}) (p.x - XMIN) < XMAX &&  
(\text{unsigned}) (p.y - YMIN) < YMAX);  
}
```

This is especially beneficial if XMIN is zero. [3]
Register Allocation

ARM has a fixed set of registers. If there are more variables than registers available, some of the variables will be stored to memory temporarily. Register allocation is a compiler optimization process that allocates variables to ARM registers, rather than to memory. Efficient register allocation minimizes register-memory swaps, thereby reducing code size and improving performance.

Integers, pointer types, fields of structures and complete structures can be allocated to registers if they are declared locally or passed as function parameters and if their addresses are not taken. [3]

Pointer Aliasing

If the address of a variable is taken, the compiler assumes that the variable can be changed by any assignment through a pointer or by any function call, making it impossible to put it into a register. Instead, the compiler allocates the variable on the stack, resulting in extra loads and stores if the variable is used extensively.

If a function uses global variables in a critical loop, it is beneficial to copy the global variables into local variables so that they can be assigned to registers. Similarly, for variables passed by reference, it is better to create a local copy of the variable and pass the address of that copy. This allows the local variable to be allocated to a register, which reduces memory traffic.

Function call overhead

Under the ARM Procedure Call Standard, up to four words of arguments can be passed to a function in registers. Subsequent arguments, if needed, are passed on the stack. This incurs the additional cost of storing these arguments in the calling function and reloading them in the called function. To minimize the overhead of passing parameters to functions:

- Keep functions small and simple and ensure that they take four or fewer arguments.
- Avoid 64-bit parameters (__int64, long long, double, etc.,) because they take two argument words.
- Avoid functions with a variable number of parameters because they are always passed on the stack.
- If a function needs more than four arguments, consider creating a structure with the related arguments and pass to functions a pointer to the structure. This increases readability.

Memory-Based Optimizations

Embedded devices are characterized by less memory. Also, current implementations of ARM systems have smaller caches compared to the PC processors. Therefore, most of the memory-based 3D graphics optimizations that were done in the PC domain such as single-pass vertex processing, software tiling architecture, deferred rendering etc., would be very appropriate to these devices. These approaches take complete advantage of the cache management provided by the processor. Also, memory hint instructions supported by that processor, can be used to bring data required for subsequent operations into the cache. Smaller texture sizes and formats can lower bandwidth requirements. Depth-Buffers can be avoided wherever possible by using techniques such as depth sorting of geometry.

Conclusion

In this paper, we presented various strategies for performance optimizations given the challenges presented by the ARM architecture. First, we addressed the issue of fixed-point arithmetic as a path for the “traditionally floating-point” parts of the 3D pipeline. Fixed point presents some trade-offs for the programmer between precision and dynamic range, but the rewards in performance over floating point emulation certainly make it worth the effort.

The same approach should be taken with the integer divide. First of all, given the form factor and resolution of the target platform, the programmer should carefully evaluate the requirement for perspective correct rasterization. For situations where it is still critical, we presented approximate methods and code-based optimizations.

We also covered other code-based optimization areas for using new architectural features such as code predication. Register allocation and function call overhead present other issues to consider in the code.
Last and certainly not least were the memory-based optimizations. Certainly the throughput capabilities of ARM-based platforms will make memory strategies even more important than it is in PC 3D engine design.

References


About the Author

Gopi K. Kolli is a senior software engineer at Intel Corporation. He is responsible for performance analysis and optimization of 3D graphics applications for the Intel® PXA250 Applications Processor with Intel® XScale™ technology. He is also responsible for formulating, developing and distributing optimized 3D graphics libraries to support game development on XScale™ microarchitecture-based handheld products. Gopi joined Intel in 1999 after receiving his M.S. degree in Computer Science and Engineering from Arizona State University. He has authored technical publications in the fields of 3D graphics and performance optimization. You may reach Gopi at gopi.k.kolli@intel.com.