International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 4 (2012) pp. 153-160

Adapting the ePUMA Architecture for Hand-held Video Games

Ingemar Ragnemalm[1] and Dake Liu[2]

[1] Information Coding Group, Dept of Electrical Engineering,
Linköping University, Sweden
ingis@isy.liu.se

[2] Division of Computer Engineering, Dept. of Electrical Engineering,
Linköping University, Sweden
dake@isy.liu.se

***Abstract*: The ePUMA architecture is a novel parallel architecture being developed as a platform for low-power computing, typically for embedded or hand-held devices. It was originally designed for radio baseband processors for hand-held devices and for radio base stations. It has also been adapted for executing high definition video CODECs. In this paper, we investigate the possibilities and limitations of the platform for real-time graphics, with focus on hand-held gaming.**

***Keywords*:** DSP, parallel processing, SIMD, embedded, low-power**.**

I. Introduction

The ePUMA architecture (embedded Parallel DSP with Unique Memory Architecture) is a master-SIMD DSP platform which was primarily designed for communication infrastructures such as the DSP subsystem for radio base stations. It has been shown that ePUMA is a good architecture for HDTV [6].

The ePUMA architecture is a work in progress. It exists as a simulator where currently assembly language programs can be run and benchmarked. Higher level programming as well as hardware implementations are planned in the near future.

In this paper, we report our investigations of the potential for the architecture as a low-power platform for video games. We analyze its strengths and bottlenecks for such applications, as well as outlining extensions to the assembly language.

II. Related work

Mobile gaming and mobile computing are rapidly growing fields with growing performance demands combined with demands on low energy consumption. Battery technology, as noted by Mochocki et. al. [8], is not evolving as fast as computing demands, thus much effort must be made to provide more performance per Watt. One approach to this is to use multi-core systems, reviewed by van Berkel [3].

For gaming, GPUs for hand-held systems are important, and the dominant commercial products are the PowerVR line from Imagination Technologies [11].

Woo et. al. [13] propose a low-power architecture with dedicated hardware subsystem for graphics as well as video coding. Earlier, the same group [12] proposed a fixed-functionality hardware for low-power graphics.

The problem of parallel processing for rendering graphics is central to our work, and has been extensively studied in the past, for other architectures. A review of the field was done by Eldridge [4].

III. The ePUMA architecture

The ePUMA is a master-SIMD architecture, based on a star-ring connectivity as shown in Figure 1. A master CPU acts as controller for 8 SIMD processors, each with 8 processing lanes, providing 8 16-bit data paths or 4 32-bit data paths per SIMD processor. The SIMD processors are the DSP subsystem of the hardware and its main processing unit. The main memory of the system is accessible from all SIMD cores, forming a star. In addition to this, a ring bus connects the eight cores for fast data communication.

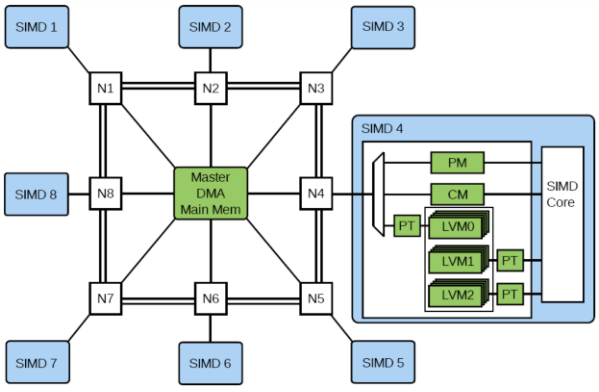

Figure 1: *Master and SIMD architecture*

In the DSP subsystem shown in Figure 1, the DSP application code is divided into the "stream" and "kernels". The stream is the top level code and it is executed in the master, a DSP controller or an MCU. Kernels are subroutines that can be executed in parallel as threads. They are executed in the eight

SIMD (Single Instruction Multiple Data) machines. The SIMD machines can be programmed using both SIMD and SIMT (Single Instruction Multiple Tasks) instructions. SIMT is for function acceleration of loops, where one layer of an FFT can be performed as one instruction. By using SIMT instructions, SIMD local control overheads are minimized.

The Master and SIMD structure, shown in Figure 1, consists of 8 SIMD processors controlled by one master. The main memory, the DMA controller, and the master are the center of the star connection network. The main memory is connected to all nodes by DMA channels controlled by the master. Each node, N1-N8, connects its SIMD machine to both the centre of the star bus and to neighbor SIMD nodes. The star bus is used to perform DMA access and data broadcasting. The ring bus can be configured into sections to locally connect SIMD pairs for stream computing.

To conclude, a complete DSP application program can be divided into three parts to achieve the highest efficiency (performance over silicon cost): The part that cannot be executed in parallel, such as the top level stream and tree-type code, will be executed in the master. The part of the code that can be data level parallel and require programmability (need to change HW in each clock cycle) will be executed on one or several SIMD machines. Finally, the part of the code that can be data level parallel and do not require programmability (do not change HW in a period of time, such as entropy coding) will be executed on an rDPA (reconfigurable Data Processing Array).

A. The SIMD data path

The SIMD machines execute all data level parallel and vector level parallel functions as well as function solver accelerations. A SIMD machine has an 8-way data path for parallel computing (8-in and 8-out, we call it "v" mode), accumulative computing (8-in and 1 out, we call it "t" mode), and Taylor function acceleration. Each SIMD performs an 8-way data parallel execution in one clock cycle.

The following local storage devices are built into each SIMD machine:

- A register file, accessible as an 8x128 bit vector register file or 8x8x16 scalar register file.
- An 8x40b accumulator built in the SIMD data path.
- 3 80kB LVMs (local vector memory), where each LVM supports 8 memory accesses in parallel. Each memory access supplies a 16b data word.
- Constant memory (CM) with 128*16b (16 vectors).
- Program memory (PM) with 1024 instructions.

SIMT (Single Instruction Multiple-Task) task level instructions are used in this solution to accelerate loop functions. Especially, convolution, transformation, and large matrix computing can be conducted in SIMT mode without data access and control overheads when connecting the LVMs directly to the SIMD data path. When connecting two input ports to two LVMs, the SIMD data path can support iterative loops; that is SIMT instructions. Most vector and triangle instructions can be carried by repeating micro code and become SIMT instructions.

Each SIMD machine handles:

1. 8-way parallel computing (v-vector mode) per clock cycle
2. 8-way accumulative function (t-triangle mode) per clock cycle
3. One 7th order (power) Taylor series output per clock cycle
4. 8 Real MAC functions per clock cycle
5. 4 complex data MAC per clock cycle
6. Two radix-2 FFT butterflies per clock cycle
7. One radix-4 FFT butterflies per clock cycle

Branch instructions and register value conditional branch instructions are supported by the SIMD processors. However, many conditional operations are rather controlled by a special-purpose bit mask register. This mask can be set by vector comparison operations and used for selectively blocking subsequent operations, e.g. copying data.

The architecture uses wide instructions that allow full access to micro code level programming of the SIMD processors. Kernel developers are thus able to formulate their own assembly instructions, creating assembly instructions that are optimized for the application. This ability reduces the number of instructions and provides flexibility for kernel programmers. [6]

The focus of the current research is the architecture exploration. For that, we use a 16b data path for simplicity, enabling the access of 8 16-bit words or 4 32-bit words. An 8*32b data path including floating point support will be investigated later.

B. The SIMD data access path

To support parallel computing in SIMD, parallel load, store and special data manipulation (shuffling) for special parallel algorithms are essential for the SIMD machines [7]. Each SIMD machine has a vector register file as well as 3 LVMs (local vector memory). Each LVM supports 8 memory accesses in parallel. Each memory access supplies a 16b data word.

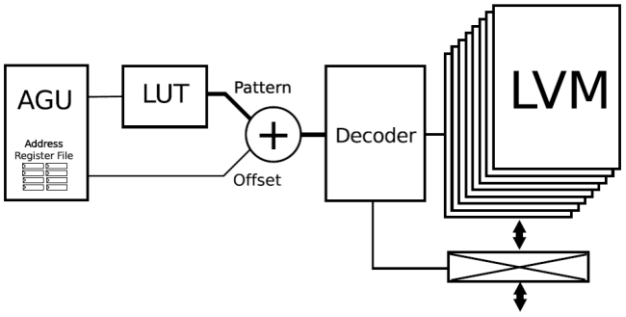

Figure 2: *Memory addressing*

Each LVM has its own address generator shown in Figure 2. For regular access, a vector data address can be generated in compile-time based on a conflict free address generator using integer linear programming. This is, however, of limited use for graphics.

The following data access modes are supported:

1. Normal address generation including ++/-- with address generator logic AGU
2. Modulo ++/-- address generation with address generator logic AGU
3. Address supplied by a permutation table where the permutation table is controlled by a local FSM counter.
4. Permutation table and AGU mixed addressing

Up to three data vectors can be supplied per clock cycle, although only one from each LVM.

The coding of addressing is based on micro code. Micro coded operations for data access are merged with micro code for arithmetic computations, together forming an assembly language instruction.

A SIMD machine gets data from the main memory and uses the LVM as the L2 computing buffer. The vector register file is used as the L1 computing buffer.

The combination of the DMA channels. LVMs and registers for a hierarchy of memory accesses shown in Figure 3. This hierarchy hints how the intermediate stages act as fast temporary storage.

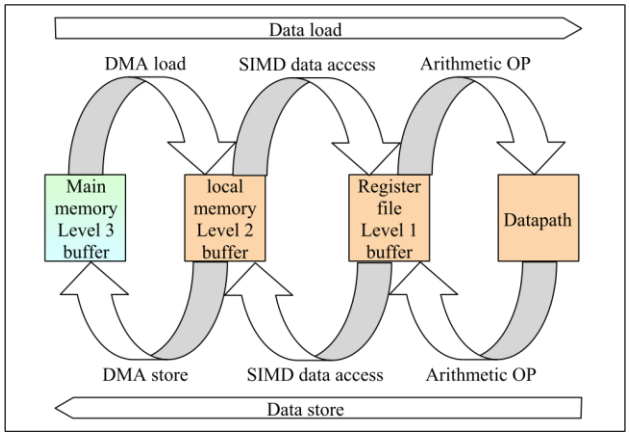

Figure 3: *Memory access and hierarchy*

IV. A graphics rendering model for ePUMA

Our goal at this stage of the project is to prove that ePUMA, in its current form, is not entirely unsuited to real-time graphics and thereby low-power handheld gaming. This does not mean that neither our approach or the architecture are fixed, only that this first study is needed to find any critical limitations and as needed address them, as well as taking advantage of the features ePUMA provides for rendering.

Let us consider a conventional graphics pipeline. Roughly, it consists of the following steps (only considering the traditional pipeline, not the new stages introduced recently, e.g. tesselation stage):

- Vertex transformations
- Primitive assembly
- Clip, cull
- Raster conversion
- Fragment processing
- Frame buffer operations

The bottlenecks are usually in the last stages, fragment processing and frame buffer operations. However, we must first consider memory usage. With the LVMs being the main source of local memory, using them efficiently is essential.

A. LVM usage

Computations need to be kept local to the SIMDs as far as possible, effectively having the SIMDs playing the role of a GPU while the master takes on the tasks of a CPU. However, we have no dedicated frame buffer hardware, and we have no texturing units. (Whether they can be added to ePUMA in some form is a later problem to consider.)

The usage of LVMs is critical. There are three LVMs in each SIMD unit. An LVM can either be available for reading or writing by the SIMD unit, or busy for DMA to and from the main memory. Let us denote these three LVMs by m0, m1 and m2.

The intention by the design is that, during computing, one LVM (m0) should be used for persistent data, data that will be the same for many iterations, one (m1) should be used for temporary input data as well as output data, and the third (m2) should be busy performing DMAs, first for output of results from the previous iteration, then for input of new data. See Figure 4. Thus, for every iteration, m1 and m2 should switch roles while m0 will keep its role.

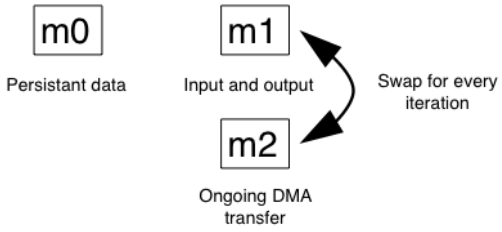

Figure 4: *Typical LVM usage*

This model is, however, flexible. A cyclic usage (see Figure 5) may prove more suitable in some cases, namely where output data is produced in one pass, and should be considered. This model is not suitable for our purposes here but has been used in our studies of other algorithms (to be published).

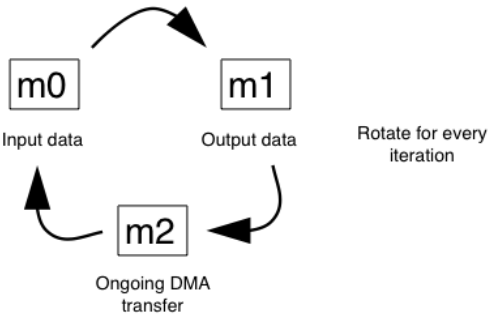

Figure 5: *Cyclic LVM usage*

Another variation of the LVM usage which we will use is to put all output data in m0, the ”persistent data” LVM, and having the other two switching between being read as source of input data and being busy uploading new input data, but with no output until all passes are finished. Thereafter, m0 is downloaded to the host.

In all these cases, one LVM at a time is being occupied for data transfers. This model is central to the ePUMA model. ePUMA algorithms are described as three stage processes, consisting of 1) prologue, 2) computing and 3) epilogue. The prologue is essentially the upload of data to an LVM. Computing takes place within the SIMD unit. The epilogue is the readout of data from an LVM to main memory. Both the prologue and epilogue may include reorganization of data.

For maximum performance, these operations should overlap as much as possible. This is illustrated by Figure 6.

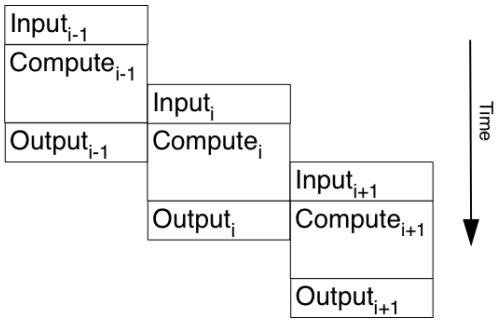

Figure 6: *Data transfer and computing overlap*

The prologue is the upload of texture data and geometry, while the epilogue is the readout of the resulting frame buffer.

B. Frame buffer size

The size of a typical frame buffer of today (iPhone 3GS) is 480x320 pixels. We expect higher resolutions in the future. This means that 480x320, that is 153600 pixels (150k pixels), is to be considered a minimum.

Every frame buffer pixel needs output color data (3 bytes in ”true color” mode) plus a depth buffer value (at least 1 byte, preferably more). The depth buffer value must be stored throughout the rendering process, but does not have to be read back to the host. We do not consider any alpha value for the frame buffer, which is supported by desktop GPUs. Also, we do not consider any stencil buffer.

For the output color, 16 bits per pixel can be considered. Lower than that is hardly acceptable today. Since we must have a depth buffer, the cost is still 3 bytes per pixel or more. So 4 bytes per pixel is very close to the minimum, and there may be a demand for more, like a 16-bit depth buffer.

At 4 bytes per pixel at 480x320 pixels, we need 600 kB for the frame buffer. The current size of an LVM is merely 80kB (40k words), and can only be scaled up to a maximum of 128kB without major design changes. This means that the entire frame buffer can under no circumstances fit in a single LVM!

This is solved by splitting the frame buffer between the SIMD units. If we split the frame buffer in eight parts, we will need a more modest 75k per part, which fits nicely in an LVM, and leaves some space for other data. For higher resolutions, this can be scaled by splitting in 16 or even 32 parts, rendering in several passes. This way we can scale the system to support 640x480 pixels, a higher depth buffer resolution, or both.

Lower-quality alternatives like 320*240 (quarter VGA) can be considered, and will reduce processing time as well as data transfers, but we will mostly ignore this since the base level chosen above already fits the architecture nicely.

This gives us three major configurations to consider:

1. Low end, 320x240 in 32 bits per pixel or 480x320 in 16 bits per pixel. Split frame buffer in eight parts.
2. Current. 480x320 in 32 bits per pixel. Split in 8 parts.
3. Future high resolution. 640x480 in 32 bits per pixel or 480x320 in 40 bits per pixel (16-bit depth buffer). Split in 16 parts (2 passes).

This approach has the advantage that the amount of information needed to render each sub-image is significantly less than what is needed for the whole scene. On the other hand, the cost of clipping will increase.

Granted that the frame buffer should be split in eight parts, we may do that in different ways, eight horizontal or vertical sections, or 2x4 as shown in Figure 7. The latter is likely to minimize the amount of added overhead for clipping.

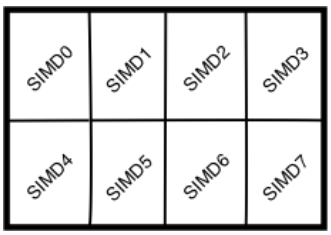

Figure 7: *A possible frame buffer partitioning into eight parts*

C. Texture storage

Let us now consider the problem of textures. Let us take the extreme case where we fill one LVM entirely with texture data. That will give us 80kB of texture data, 20480 pixels for uncompressed 32-bit textures. That means, for example, four 64x64 textures and two 64*32 texture, or 20 32x32 textures. If we want to take mip-mapping into account, which we should, textures cost 25% more (since we only need to load two levels). Then we can still safely fit four 64x64 textures or 16 32*32 ones, or a single 128x128 one.

If we need to fit more texture data at the same time, we need to use texture compression. This problem has been studied [1][5] but needs to be studied specifically for the ePUMA platform. We choose to ignore it for now.

Note that this limited texture capability only refers to a small part of the scene, 1/8 for what we consider the most typical case! That means that only textures that are needed for that particular part of the scene must be loaded in a particular LVM. What we are really doing is to use the LVMs as a replacement for the texture cache in a conventional GPU.

D. Model data

Similar reasoning apply to geometry. We will need at least 16 bits, preferably 32 bits per coordinate in vertex data, which demands 48 to 96 bits for one vertex (6 or 12 bytes). Each vertex will also need texture coordinates, two coordinates (s, t) of 16 bits each. It will be addressed by an index array, roughly four times per vertex, which demands 8 more bytes. This gives us 18 bytes for very modest precision, which allows 4500 vertices, before considering that the memory space needs to be shared with the texture data as well. Compression of vertex data is possible, to some degree. The main source of geometry compression is level-of-detail methods, e.g. as proposed by Purnomo et. al. [10].

We conclude that model data is no major problem. Model data can be uploaded as needed in small amounts, and the most critical LVM usage will come from textures.

E. Texture access

Texture access is a major issue, with several important considerations. First and foremost, we need to support texture filtering, so we must access several texels at once. These texels will always be neighbors, on adjacent rows. For uncompressed RGBA textures, we can read two neighbor texels in one instruction, as long as data is in different memory banks. After reading the pixels, they will have to be unpacked from byte components to word components, as illustrated in Figure 8.

This operation can be performed in one clock cycle using the half word extension instruction already defined for ePUMA.

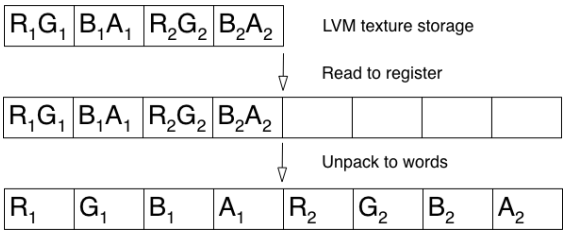

Figure 8: *Texture access and unpacking*

Another consideration of texture access is perspective correct texturing. If we do not make any optimizations, perspective correct texturing costs one division for every pixel. That division is accelerated using the Taylor expansion acceleration mentioned above. This takes eight clock cycles to complete (or more, depending on precision), but is pipelined and parallel. Four of these cycles can be done in parallel to other work, which leaves four effective cycles, that is a single clock cycle if four values are submitted at once. For perspective correct texturing, that is not possible in our model so the effective cost is four cycles, granted that other work can cover the other four, which is certainly the case.

F. Rendering model

We now have enough detail to sketch the overall rendering model. We will suggest an approach based on the hardware dependent modeling above.

The overall rendering model, according to the discussion above, works as follows: The frame buffer is split into eight parts, one stored in an LVM of one of the SIMDs. A scene is rendered by multiple passes, as follows, where the frame buffer is allocated at step 3.

1. Geometry processing. Geometry data (vertices, transformation data) are split in batches that fit in one LVM and processed in the SIMD units. Every vertex is also tagged with the frame buffer region it falls into. The result is downloaded back to the host.
2. Spatial sorting. Every primitive is put in lists for each region it falls into. Note that this processing can be done in parallel with step 1 if step 1 is performed as more than a single pass.
3. Clipping, raster conversion and fragment processing. Each list constructed in step 2 is uploaded in batches (a single polygon at a time or several depending on texturing demands). Note that textures may be uploaded at every pass. Different texture data are uploaded to different SIMDs, depending on their respective content.

According to the classification by Molnar et. al. [9], this is a "sort-middle" algorithm, which is the most common way to parallelize graphics. More specifically, it is a "sort-middle tiled" architecture according to the classification by Eldridge [4].

In our work, we have assumed that step 3 is the dominating one. This is certainly not true for every kind of scene, but a reasonable assumption for many real applications.

We will now discuss more about how to handle the fragment processing in step 3.

The geometry and texture data needed for rendering will be uploaded to LVMs in suitable batches of up to 80 kB. This will usually demand several passes. Although it is possible to render all data for 1/8 of the image in one pass by demanding all texture and geometry data to fit in one LVM, we do not consider that approach realistic. Rather, one pass may consist of as little as a single polygon and texture data for it.

This will require many memory transfers, especially for texture data, but as we will see below, the memory access is not so expensive on the ePUMA.

Thus, throughout the rendering, one LVM is used for frame buffer data though several passes, while the other two LVMs alternate as current source of texture and model data memory or being loaded by the data for the next pass. For the more detailed models, we may upload one single model with textures, and in other cases we may upload several models sharing the same textures.

We will still be limited to less than 80kB of texture data at a time. This may be a serious limitation in some cases, but then we should consider getting around it with texture compression. We may also consider uploading parts of textures. This is, however, a limited possibility since we must work with consecutive blocks of memory to do so efficiently. Thus, textures will need to be split horizontally, or preprocessed into smaller sections. The latter is particularly suitable for detailed backgrounds, like sky boxes.

V. Graphics performance assessment

In this section, we will analyze the graphics performance that the ePUMA will have with the models given above.

There are two bottlenecks to consider: memory transfer and fragment computing. Potential other possible bottlenecks to consider are the preprocessing of geometry on the host to produce data to pass to each subpart of the image, as well as other geometry processing in the SIMDs (e.g. clipping and raster conversion) which requires more processing for the same reason. However, we choose to ignore that at this time, since texture access, part of fragment computing, is considered the major bottleneck [1].

Yet another potential bottleneck in all parallel computing is the communication between the processors. In our case, this is not a problem since each SIMD can perform much work independently, and the only communication needed is a signal once one pass is completed. The number of passes should be fairly low, with many (thousands or more) fragments computed in one pass. Thus, we can safely count on zero overhead for communication.

A. Memory transfer

Memory transfers of textures, model data and frame buffer data will be performed by DMA. This will cost around 40 cycles of setup time, and then it will transfer one 128-bit vector per clock cycle over the 128-bit wide DMA channel. For one full 80kB LVM, that means 655360 bits in 5120+40 cycles = 5160 cycles, which will take 10.4µs at 500 MHz. For every rendering pass, we need eight uploads, resulting in a memory transfer time of 8*10.4 = 83.2 µs. For data uploads, there will be more than one block of memory, for which we must expect more overhead, which will, however, be variable. On the other hand, data uploads may just as often be smaller than a full LVM.

A number of passes will be computed, where every pass makes uploads, but only the last pass for a frame will download data. For computing a number of passes, denoted P, each frame will cost P*83.2 µs for uploads (plus some additional overhead if there are many separate memory blocks) plus a download cost of 83.2 µs = 83.2 (P+1). This gives us the following timings:

4 passes: 416 µs
8 passes: 749 µs
11 passes: 1 ms
200 passes: 17 ms

Thus, memory bandwidth will allow over 200 passes for 50 fps animation, a comfortable capacity.

B. *Fragment calculation*

Fragment computing is a harder part to measure, being more application dependent and also subject to many approximations for balancing performance and quality. There are benchmarks available, like the GraalBench benchmark [2], but they are not applicable at this stage. Instead, we have to outline a realistic scenario.

We assume the following situation:

- Multiple coverage, so all pixels are rendered M times on the average.
- All surfaces are textured.
- Perspective correct texture mapping.
- Two light sources.
- Phong shading with specular reflections.
- The geometry detail is coarse enough to make the fragment processing the dominant computation, not geometry processing.

The number M, the multiple coverage number, is vital. If we can't get an M of 1 or more, we can not render full-screen scenes fast enough with the method at hand. If M is near 1, we can fill the screen with polygons, but no significant overlap is allowed, and the time spent on other tasks, like vertex processing and clipping, must be totally negligible, which is unrealistic. If M is significantly larger than 1, the demand for optimization is relaxed.

With 480*320 pixels, 50 fps animation will require 7680000 pixels to be produced per second, which, split over 8 SIMD units will require 960000 pixels per second per SIMD unit. This leaves us slightly more than 500 cycles per pixel, which means 500/M cycles per fragment. We must be able to produce fragments at a rate that allows a reasonable value of M.

We will now outline a typical fragment processing algorithm. This is a well-known procedure (described in any serious computer graphics textbook) included here in order to make a realistic cycle count. We cannot make the cycle count without deciding on what approximations to make. We choose a reasonably ambitious but still basic model. We include perspective correct texture mapping, but linear interpolation for Phong shading, using two light sources. One texture look-up is done, with tri-linear texture interpolation. No anti-aliasing. Directional light only (so Blinn-Phong specular lighting can be used).

1) *Variables*

Most of the following variables are provided by the outer layer, the raster conversion process. This includes the fragment coordinates, Phong shading vector, texture coordinate over z, light vectors, normal vector, half-vector, viewing direction and material specularity,

Fragment coordinates (x, y)
Phong shading vector: **p** = (px, py, pz)
Phong shading vector increment: **dp** = (dpx, dpy, dpz)
Texture coordinates (over z): **tex** = (sz, tz, iz)
Texture coordinate increments: **dtex** = (dsz, dtz, diz)
Light vectors **i1**, **i2**
Light colors **c1**, **c2**
Blinn-Phong half-vector **h1, h2**
Normal vector **n**
Normal vector increment **dn**
Viewing direction/surface position **view**
Material specularity constant

Other variables are intermediary. For simplicity, some are entirely implicit, while others are given defined symbols:

Texture coordinates: s, t, z
Normalized Phong shading vector: **np** = (npx, npy, npz)

We can now outline the fragment processing in more detail:

2) *Algorithm inner loop outline*

Increment Phong shading vector: **p** = **p** + **dp**
Increment texture coordinates: **tex** = **tex** + **dtex**
Increment normal vector **n** = **n** + **dn**
Inspect depth buffer: if depth(x, y) < iz then break
  else write iz to depth buffer
Calculate np: **np** = normalize(**p**)
Calculate z = 1/iz
Calculate s = sz * z
Calculate t = tz * z
Calculate interpolation weights and indices for texture memory access
Calculate texture memory locations (for two mip-mapping levels)
Get two texels from higher texture level, including unpacking from 8-bit to 16-bit components with half word extension
Get two texels from higher texture level, unpack
Interpolate texels
Get two texels from lower texture level, unpack
Get two texels from lower texture level, unpack
Interpolate texels
Interpolate between the levels
Calculate light level 1:
Diffuse shading: Dot product (**np** • **i1**), multiplication with material constant
Specular shading: Dot product (**view** • **h**), specularity calculation, multiply by material constant
Calculate light level 2: as above
Calculate final pixel value
Blend pixel value with old pixel value using the alpha value
Pack pixel to 8-bit components
Write pixel value to frame buffer

3) Timings

The timings below call for some clarifications. Optimized ePUMA code must be rearranged to hide pipeline latency, which is generally done by overlapping several passes or independent part of the same pass. Operations like 1/x and 1/sqrt(x) are done using Taylor expansion, combined with a range check, so we can resort to other solutions outside the most vital interval. The specularity calculation, a power function, is also performed with the Taylor expansion acceleration.

The calculations of lighting allows more parallelism than other parts. The timing for a lighting calculation is 1 cycle for **np • i1** and **np • i2**, 1 cycle for **view • h1** and **view • h2** and 4 cycles for two specularity power functions, all executed for both light sources in parallel. Finally, we need two cycles to sum the result and multiply with light source colors (**c1**, **c2**).

Now we can count the total number of cycles:

Increments: 2 cycles (by aligning vectors so two can be updated at once)
Inspect depth buffer, write depth: 2 cycles
np normalization: 6 cycles
z: 4 cycles
texture indices, interpolation weights: 1 cycle (multiply, shift of s and t)
memory locations: 2 cycles
Get two texels: 1 cycle * 4
Interpolations: 1 * 3
Calculate light level: 8 cycles for both
Calculate final pixel value: 2 cycles
Read old pixel, blend, pack and write to frame buffer: 4 cycles

This sums to 38 cycles.

When calculating the cycle cost, we can take advantage of the parallel lanes to a varying degree. Most data are vectors with 3 or 4 components. In some cases we can calculate two vectors at once, while some operations must be limited to one vector. The light sources are done in parallel as described above. Although we only perform one vector normalization, we can perform two for the same cost.

The timings require the addition of some new operations to the ePUMA instruction set, as described in the next section.

Thus, a single fragment will cost 38 cycles, in which time all the eight SIMD units will produce one fragment. In addition to this cost, there is an overhead per vertex, per polygon and per texture. Assuming (for now) that the overhead is negligible, we can produce as much as 13 million fragments per second per SIMD, for a total of 105 million fragments per second. This means that we may render each pixel more than 13 times per frame ($M \leq 13$) and still produce 50 frames per second.

This is a modest number compared to what a dedicated GPU can output. However, this is using a generic architecture, only modified by adding new assembly instructions.

C. Instructions added for improved graphics support

In order to make the ePUMA more suited for a certain application, we may add certain additional machine-code instructions. This is possible on the ePUMA architecture, where the micro code is exposed to the kernel programmer.

Table 1 (below) is a list of instructions that have been added for image and video signal processing. These instructions are also suitable for video games.

Table 1. Accelerated operations for image and video signal processing

Algorithms	Kernel operations
SAD	Result = $\sum \lvert A_i - B_i \rvert$
Interpolation	Result = round $((a_1x_1 \pm a_2x_2 \pm a_3x_3 \pm a_4x_4 \pm a_5x_5 \pm a_6x_6) /32)$
De-blocking filter	Result = round $((a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5 + a_6x_6) /8)$
8×8 DCT	Result = Integer butterfly computing and data access
Color transform	Result1 = $a_1x_1 \pm a_2x_2 \pm a_3x_3$; Result2 = $a_1x_4 \pm a_2x_5 \pm a_3x_6$;

For graphics, the instructions in Table 2 have been identified as relevant. In particular, a fast division operation is vital for perspective correct texture lookup. Vector normalization is equally important.

Table 2. Accelerated operations for graphics and video games

Algorithms	Cycle cost	Kernel operations
1/x	4 cycles	Result = 1/x[31:0];
Short convolution	1	Result1 = saturate(round(a1x1 ± a2x2 ± a3x3 ± a4x4)); Result2 = saturate(round(a1x5 ± a2x6 ± a3x7 ± a4x8));

More optimizations are possible through defining more special-purpose instructions tailored for certain situations. That is a task for future work.

D. Low-power computing options

For the case where we design for extreme low-power, we should consider lower frequencies. At 80 MHz with 65 nm technology and 0.6 V power, the power consumption will drop to about 0.14 W. This would produce 17 million fragments per second, still allowing 2 times overdraw (more on lower fps rates or simpler rendering cases, e.g. less texture filtering or Gouraud shading).

Table 3. Graphics performance for the given model in two different clock frequencies (480x320 pixels)

ePUMA clock frequency	80 MHz	500 MHz
Fragments/second	17 million	105 million
Overdraw at 50 fps	2	13
Estimated power	0.14W	3.5W
Supply voltage	0.6V	1.2V
Vector data memory cost	80kB*3*8	80kB*3*8

The two cases are compared side-by-side in table 3. Writing games with the limitations of the low-power system would certainly be a challenge, but we should remember that this is not at the lowest possible graphics quality, but rather fairly high. There is room for application-dependent optimizations, like using Gouraud shading in parts of the scene as well as reducing the interpolation to bilinear or even nearest-neighbor, to improve the fragment rate even more.

VI. Conclusions and future work

We have reported function decomposition, mapping and scheduling for the graphics part of the ePUMA project, where the goal so far has been to show that the ePUMA is feasible for real-time graphics, even if not yet optimized for the purpose. We have found strategies for rendering with the existing architecture and by cycle counts demonstrated that performance is possible that allows graphics to be rendered in real time under given circumstances.

The quantitative results show that the inner loop of polygon rendering with linear texture filtering and Phong shading allow a fill rate of 13 fragments per pixel. With stricter demands on the rendering, minimizing the overdraw, the system can use a lower clock frequency in order to produce a low-power gaming system processor consuming as little power as 0.14 Watts.

Future work include implementation of a complete graphics pipeline as well as investigations of algorithms and architecture extensions for improved graphics performance. These future simulations will, like the present work, motivate new assembly instructions as well as hardware modifications.

Furthermore, we are investigating texture compression methods suitable for the ePUMA that will make it possible to render higher resolution textures, reducing the need for a texture cache.

Acknowledgements

Special thanks to Olof Kraigher, Jian Wang and Andréas Karlsson for fruitful discussions and technical assistance. This work was funded by the Swedish strategic research foundation (SSF).

References

[1] T. Akenine-Möller, J. Ström, ”Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones”, SIGGRAPH '03, 2003.

[2] I. Antochi, B. Juurlink, S. Vassiliadis, P. Liuha, ”Graal-Bench: a 3D graphics benchmark suite for mobile phones”, Proceedings of the 2004 ACM SIGPLAN/SIG-BED

[3] C.H. van Berkel, ”Multi-Core for Mobile Phones”, DATE09, 2009

[4] M. Eldridge, 2001, Designing Graphics Architectures Around Scalability and Communication, Dissertation, Stanford University.

[5] S. Fenney, ”Texture Compression using Low-Frequency Signal Modulation”, Graphics Hardware, 2003

[6] D. Liu, Embedded DSP Processor Design, Application Specific Instruction set Processors, Elsevier (Morgan Kaufmann) 2008 ISBN 9780123741233

[7] D. Liu, J. Sohl, J. Wang, ”Parallel computing and its architecture based on data access separated kernels”, IGI IJERTCS March 2010.

[8] B. Mochocki, K. Lahiri, S. Cadambi, ”Power Analysis of Mobile 3D Graphics”, DATE '06: Proceedings of the conference on Design, automation and test in Europe, 2006

[9] S. Molnar et. al. 1994, A Sorting Classification Of Parallel Rendering, IEEE Computer Graphics and Applications, 14(4): pp 23-32.

Author Biographies

Ingemar Ragnemalm was born in Linköping, Sweden, 1962. PhD 1993 in image processing at Linköping University at the Department of Electrical Engineering. Worked outside the university 1993-1999, with interactive television and game programming. Teacher and researcher at Linköping University since 1999, teaching computer graphics and game programming. Research interests include image processing, computer graphics, parallel processing and user interfaces.

Dake Liu is the professor of the chair and the Director of Computer Engineering Division, Department of Electrical Engineering at Linköping University, Sweden. He is IEEE senior member, Founder and chief scientist of Coresonic AB Sweden, founder and CTO of FreehandDSP AB Sweden. His research interests are application specific instruction set processors (ASIP), on-chip multiprocessors, and multiprocessor programming for embedded especially streaming signal processing applications. Head of the ePUMA project.