Accelerating Algorithms for Physically-Based Modeling

Dave Estes
University of North Carolina at Chapel Hill
COMP 768 Fall 2012
Overview

- GPGPU in Today’s Game Engines
- Rolling Your Own
  - Optimizing C++ for CPU
  - Multicore - OpenMP
  - SIMD - Single Instruction Multiple Data
- GPGPU

Conclusion: What’s the right tool for the job?
Today’s Game Engines

- Rise of General Purpose GPU Computing
  - Moore’s Law is faltering, Parallel Computing is taking up the call
  - Fragment and Vertex Shaders produced sophisticated shader languages
  - GPGPU was the natural evolution of these massively parallel pixel pipelines and shader languages
Today’s Game Engines

- nVidia PhysX
  - First fully GPU accelerated game engine
  - Proprietary and limited to nVidia hardware
- Havok Physics
  - Strong GPU support
- The rest? They’re catching up...
- Bullet 2.81 - OpenCL rigid body pipeline
Rolling Your Own

- Optimizing C++ for CPU
- Multicore - OpenMP
- SIMD - Single Instruction Multiple Data
- GPGPU

1. roll your own
A situation where someone needs to perform a task but the method to complete the task doesn't exist so she has to create the solution herself. It derives from "rolling your own" cigarettes versus buying pre-made cigarettes.

Computer programmer #1: Is there a built-in function to filter the database?
Computer programmer #2: Sorry, man, you're going to have to roll your own.

programming, design, cigarettes, algorithm, engineering
by tofegregg, Feb 3, 2009  share this  add a video
Optimizing C++ for CPU

- Asymptotic Performance vs. Real World Performance
- Managing the Pipelines (Spacing Loads Stores, Branch Prediction, etc.)

ARM's Cortex A57 - www.anandtech.com
Coding Tips

Avoid calculations in loops

```java
for (int j = 0; j < MAX; j++) {
    for (int i = 0; i < MAX; i++) {
        int offset = j * stride + i;
        out[offset] = in[offset] * 2;
    }
}

for (int j = 0; j < MAX; j++) {
    int offset = j * stride;
    for (int i = 0; i < MAX; i++) {
        out[offset] = in[offset] * 2;
        offset++;
    }
}
```

The compiler *should* hoist loop invariants
Coding Tips

Avoid division

\[ i = i / 16; \]
\[ i = i << 4; \]

The compiler *should* replace division by \( 2^x \)

```c
for (int i = 0; i < max; i++) {
    f[i] = f[i] / 16.0;
}
float g = 1 / 16.0;
for (int i = 0; i < max; i++) {
    f[i] = f[i] * g;
}
```

Be careful with floating point precision error accumulation
Coding Tips

Banging bits on ARM

```c
int isOneBitSet(unsigned int v) {
    int ret;

    asm ("clz %[ret], %[v]\n" 
        "rsb %[ret], %[ret], #32\n" 
        "mov r1, #1\n" 
        "and %[ret], %[v], r1, lsl %[ret]\n" 
            : [ret] "=r" (ret) : [v] "r" (v) : "r1");

    return ret;
}
```

Many ARM instruction can pre-shift second operand
A Note About Parallelism

- Task Parallelism
  - Traditional multitasking where multiple, different tasks run in parallel
  - e.g. Running Keynote, listening to Pandora, and getting an email notification

- Data Parallelism
  - Multiple, highly nearly identical tasks each working on unique data
  - e.g. Rigid body collision detection with multiple nodes each with multiple vertices, edges, and faces.
Task vs. Data Parallelism
OpenMP - Multi Processor

- Specification from OpenMP Architecture Review Board
- Strong support in GNU and Clang for Intel and ARM
- Just works on Apple Mac OS X
OpenMP Threads

- Traditional multiprocessor involves creating, dispatching, and monitoring threads
- OpenMP does all of this work through its Fork-Join Model

OpenMP achieves parallelism exclusively through threads.
OpenMP Components

- Compiler Directives
  - Identifies loops
- Run-time Library
  - Forks and joins threads
- Environment Variables
  - Controls threading
OpenMP Simple Example

- Identifies a for loop as a parallel section

```c
#pragma omp parallel for
for (size_t i = 0; i < particle_count; i++) {
    /* Update the position */
    if (runAlgorithm(i, dt)) {
        /* If the particle hits the ground then reset it */
        initializeParticle(i);
    }
}
```
OpenMP Critical Example

- Concurrent access of common data is not advisable
- OpenMP does allow for it

```c
#pragma omp parallel for
for (size_t i = 0; i < particle_count; i++) {
    /* Update the position */
    if (runAlgorithm(i, dt)) {
        /* If the particle hits the ground then reset it */
        initializeParticle(i);
    }
    /* Update particle update counter */
    #pragma omp critical
    count++;
}
```
OpenMP Critical Example

- Reduces the likelihood of critical section collisions

```c
#pragma omp parallel for
for (size_t i = 0; i < particle_count; i += groupSize) {
    for (size_t j = 0; j < groupSize; j++) {
        /* Update the position */
        if (runAlgorithm(i+j, dt)) {
            /* If the particle hits the ground then reset it */
            initializeParticle(i+j);
        }
    }
    /* Update particle update counter */
    #pragma omp critical
count += groupSize;
}```
SIMD

- Single Instruction, Multiple Data
- Unlike OpenMP, SIMD only support true Data Parallelism
- Intel SSE, ARM Neon
- Intel has a great vectorizing compiler
ARM Neon

- Neon Register File can be mapped as single S, double D, or quad Q registers
- Pipeline is 128 bits

- Lane count is based on element size
  - 1 quad, 2 doubles, 4 singles, 8 halves
Using intrinsics is often a simple approach to SIMD

```c
void dotProduct(float *c, float *a, float *b, int n) {
    int i;

    for (i = 0; i < (n & ~3); i += 4) {
        vst1q_f32(&c[i], vmulq_f32(vld1q_f32(&a[i]), vld1q_f32(&b[i])));
    }
    if (i & 2) {
        vst1_f32(&c[i], vmul_f32(vld1_f32(&a[i]), vld1_f32(&b[i])));
        i += 2;
    }
    if (i & 1) {
        c[i] = a[i] * b[i];
    }
}
```
OpenCL

- Specification from the Khronos Group Inc.
- Strong support in GNU and Clang for Intel
- Emerging support for ARM from Chip Manufacturers
- Just works on Apple Mac OS X
OpenCL Overview

- Open standard for general purpose, high power computing in heterogeneous environments
- Kernels are compiled from programs on the host at run-time
- Built-in vector data types leads to strong SIMD support for CPU
- A good deal more complicated to use than OpenMP or even Neon Intrinsics
- OpenGL/CL Interop
Walkthrough - Setup

1. Get Platform ID
2. Get Device IDs
3. Query Supported Extension for devices
4. Create the CL context
5. Create the CL command queue
6. Create the programs then compile into kernels
7. Create CL buffers
8. Set kernel arguments to consts or CL buffers
9. Query the optimum local workgroup size
Walkthrough - Executing

1. Enqueue one or more buffer writes or copies
2. Enqueue kernel over ND range using global and local workgroup size
3. Enqueue one or more buffer reads
__kernel void computePixel(__global int* inputVector,
                           __global int* coefficientPlane,
                           __global uint* frameVolume,
                           __global uint* frameVolumeDims,
                           __write_only image2d_t frameBuffer,
                           const uint cm_width,
                           const uint cm_height,
                           const uint display_width,
                           const uint display_height)
{
    uint x = get_global_id(0);
    uint y = get_global_id(1);
    int2 coord = { x, y };
    uint cm_size = cm_width * cm_height;
    uint cpOffset = (y * display_width + x) * cm_size;
    uint mult = 1;
    uint fvOffset = 0;

    if ((x < display_width) && (y < display_height)) {
        for (uint cmj = 0; cmj < cm_height; cmj++) {
            uint s = 0;
            s += coefficientPlane[cpOffset] * x; cpOffset++;
            s += coefficientPlane[cpOffset] * y; cpOffset++;
            for (uint cmi = 2; cmi < cm_width; cmi++) {
                s += coefficientPlane[cpOffset] * inputVector[cmi]; cpOffset++;
            }
            fvOffset += s * mult;
            mult *= frameVolumeDims[cmj];
        }
        uchar4 colori = as_uchar4(frameVolume[fvOffset]);
        float4 colorf = {
            (float)colori.s0/255.0f,
            (float)colori.s1/255.0f,
            (float)colori.s2/255.0f,
            1.0f};
        write_imagef( frameBuffer, coord, colorf );
    }
}
Sample Kernel Program

```c
__kernel void computePixel(__global int* inputVector,
                           __global int* coefficientPlane,
                           __global uint* frameVolume,
                           __global uint* frameVolumeDims,
                           __write_only image2d_t frameBuffer,
                           const uint cm_width,
                           const uint cm_height,
                           const uint display_width,
                           const uint display_height)
{
    uint x = get_global_id(0);
    uint y = get_global_id(1);
    int2 coord = {x, y};
    uint cm_size = cm_width * cm_height;
    uint cpOffset = (y * display_width + x) * cm_size;
    uint mult = 1;
    uint fvOffset = 0;
    if ((x < display_width) && (y < display_height)) {
        for (uint cmj = 0; cmj < cm_height; cmj++) {
            uint s = 0;
            s += coefficientPlane[cpOffset] * x; cpOffset++;
            s += coefficientPlane[cpOffset] * y; cpOffset++;
            for (uint cmi = 2; cmi < cm_width; cmi++) {
                s += coefficientPlane[cpOffset] * inputVector[cmi]; cpOffset++;
            }
            fvOffset += s * mult;
            mult *= frameVolumeDims[cmj];
        }
        uchar4 colori = as_uchar4(frameVolume[fvOffset]);
        float4 colorf = { (float)colori.s0/255.0f,
                         (float)colori.s1/255.0f,
                         (float)colori.s2/255.0f,
                         1.0f};
        write_imagef( frameBuffer, coord, colorf );
    }
}
```
Sample Kernel Program

```c
__kernel void computePixel(__global int* inputVector,
                           __global int* coefficientPlane,
                           __global uint* frameVolume,
                           __global uint* frameVolumeDims,
                           __write_only image2d_t frameBuffer,
                           const uint cm_width,
                           const uint cm_height,
                           const uint display_width,
                           const uint display_height)
{
    uint x = get_global_id(0);
    uint y = get_global_id(1);
    int2 coord = { x, y };
    uint cm_size = cm_width * cm_height;
    uint cpOffset = (y * display_width + x) * cm_size;
    uint mult = 1;
    uint fvOffset = 0;
    
    for (uint cmj = 0; cmj < cm_height; cmj++) {
        uint s = 0;
        s += coefficientPlane[cpOffset] * x; cpOffset++;
        s += coefficientPlane[cpOffset] * y; cpOffset++;
        for (uint cmi = 2; cmi < cm_width; cmi++) {
            s += coefficientPlane[cpOffset] * inputVector[cmi]; cpOffset++;
        }
        fvOffset += s * mult;
        mult *= frameVolumeDims[cmj];
    }

    uchar4 colori = as_uchar4(frameVolume[fvOffset]);
    float4 colorf = {
        (float)colori.s0/255.0f,
        (float)colori.s1/255.0f,
        (float)colori.s2/255.0f,
        1.0f};
    write_imagef( frameBuffer, coord, colorf );
}
```
Sample Kernel Program

```c
__kernel void computePixel(__global int* inputVector,
                           __global int* coefficientPlane,
                           __global uint* frameVolume,
                           __global uint* frameVolumeDims,
                           __write_only image2d_t frameBuffer,
                           const uint cm_width,
                           const uint cm_height,
                           const uint display_width,
                           const uint display_height)
```

```c
{
  if ((x < display_width) && (y < display_height)) {
    for (uint cmj = 0; cmj < cm_height; cmj++) {
      uint s = 0;
      s += coefficientPlane[cpOffset] * x; cpOffset++;
      s += coefficientPlane[cpOffset] * y; cpOffset++;
      for (uint cmi = 2; cmi < cm_width; cmi++) {
        s += coefficientPlane[cpOffset] * inputVector[cmi];
        cpOffset++;
      }
      fvOffset += s * mult;
      mult *= frameVolumeDims[cmj];
    }
    uchar4 colori = as_uchar4(frameVolume[fvOffset]);
    float4 colorf = {
      (float)colori.s0 / 255.0f,
      (float)colori.s1 / 255.0f,
      (float)colori.s2 / 255.0f,
      1.0f};
    write_imagef(frameBuffer, coord, colorf);
  }
}
```
Managing the PCIe Bus

- Despite the 16 GB/s bandwidth of PCIe v3.0 x16, moving memory of the PCIe bus is painfully slow.
- Without properly managing the PCIe traffic, the CPU will outperform the GPU every time.
  - Minimize traffic
  - Avoid multiple, small transfers
  - Consider using a “packet” and distributing its contents by copying once on the device.
AMD HD 7950

- 28 Compute Units (CU)
- 4 Vector Units (VU) per CU
- 16 Stream Processors (SP) per VU
- = 1792 SPs total
OpenCL Memory Hierarchy

ATI HD 7950
- n = 27
- m = 63
Bank Conflicts

**OK**

- SP 0 → Bank 0
- SP 1 → Bank 1
- SP 2 → Bank 2
- SP 3 → Bank 3
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 63 → Bank 63

**OK**

- SP 0 → Bank 0
- SP 1 → Bank 1
- SP 2 → Bank 2
- SP 3 → Bank 3
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 63 → Bank 63

**Bad**

- SP 0 → Bank 0
- SP 1 → Bank 1
- SP 2 → Bank 2
- SP 3 → Bank 3
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 4 → Bank 4
- SP 63 → Bank 63
Conclusion

- Task Parallel
  - Data Parallel
    - SIMD
      - SIMD Instructions
      - SIMD Registers
    - OpenCL
      - GPU
      - SIMD Instructions
      - SIMD Registers
      - Multi-core
      - Hyper-Threading
  - Task Parallel
    - OpenMP
      - Multi-core
      - Hyper-Threading
    - Posix Threads
      - Multi-core
      - Hyper-Threading

Ease of Use
- Simple
- Difficult
References

- OpenMP Tutorial from Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/openMP/
- ARM (image), http://www.arm.com/images/NEON_ISA.jpg
- MacResearch OpenCL Tutorials, http://www.macresearch.org/opencl_episode4