Inside a GPGPU Managed Platform with an Auto-tuning JIT Compiler

Chris Jang
fastkor@gmail.com

http://fastkor.wordpress.com/
Three questions

Auto-tuning on the GPU...

1. *What* is it?
2. *Why* is it important?
3. *How* is it done?
What is GPGPU auto-tuning?

► **Automated performance tuning**
► Without programming...
  ▶ Find code that runs fast
  ▶ Adapt code to hardware
  ▶ Optimize code for SDK/driver
Why is auto-tuning important?

Same GPGPU algorithm... over 100x speed difference

DGEMM 400x400 memory buffers (ATI HD 5870) histogram from EM auto-tuning over model family
Why is auto-tuning important?

Same GPGPU algorithm... **adapts** to GPU hardware

DGEMM 400x400 memory buffers (**NVIDIA** GTX 480) histogram from EM auto-tuning over model family
Why is auto-tuning important?

*Same GPGPU algorithm... optimizes for SDK/driver*

DGEMM 400x400 memory buffers (ATI SDK 2.1 and 2.2) histogram from EM auto-tuning over model family
Why is auto-tuning important?

*Same GPGPU algorithm... no programming, fast code optimization, adaptation*

- ATI HD 5870 *(Evergreen)*
- NVIDIA GTX 480 *(Fermi)*
- SDK 2.1 and SDK 2.2 for ATI OpenCL
How is auto-tuning done?

This is really two questions . . .

1. How are fast math kernels found?
2. How to integrate with a virtual machine JIT?
How are fast math kernels found?

1. Parameterized kernel family
   - Designed by hand
   - Compiler generated

2. Search using statistical optimization
   - Expectation-Maximization works!
   - Maximize expected throughput (GFLOPS)
   - Examples: GEMM and GEMV
How to integrate with a VM JIT?

Why ask this question...

- Fast GPGPU kernels are **not** enough
  - Memory management - I/O can dominate performance
  - GPGPU programming is still too hard
- Natural solution is managed platform for GPGPU
  - Application virtual machine
  - Garbage collection
  - Just-in-time *auto-tuning* compiler
  - High-level programming language
GPGPU kernels are not enough

Let’s see some examples... 

- Memory management - I/O can dominate performance
- GPGPU programming is still too hard
Memory management is important

Memory objects can be very expensive

SGEMM 1600x1600 AB images, C buffer (ATI HD 5870) histogram from EM auto-tuning over model family
Memory costs can be huge

Order of magnitude difference in peak throughput!

- 1340 GFLOPS if no I/O, all memory on GPU
- 660 GFLOPS if read back result
- 140 GFLOPS if send input data
- 120 GFLOPS if send input data and read back result

SGEMM 1600x1600 AB images, C buffer (ATI HD 5870)
What are the memory costs?

- Data transfer over the PCIe bus
  - Bus slower than RAM
  - DMA not always available
  - Bus controllers not all the same (mainboard/GPU)
  - May need to copy application to driver memory

- Memory allocation on GPU
  - Textures (images) expensive
  - Memory buffers cheap
Add two arrays on the CPU...

```c
float a[1000], b[1000], c[1000];
void fooCPU(float* c, float* a, float* b) {
    for (int i = 0; i < 1000; i++)
        c[i] = a[i] + b[i];
}
```
Add two arrays on the GPU...

```c
__kernel void fooGPU(__global float* c,
                     __global float* a,
                     __global float* b) {

    c[get_global_id(0)]
        = a[get_global_id(0)]
        + b[get_global_id(0)];
}

size_t global[1], local[1];
global[0] = 1000;
local[0] = 1;
clEnqueueNDRangeKernel(..., global, local, ...
```
That looked easy!

It is easy for two reasons...

- Simple algorithm (entry-wise addition)
- No performance tuning

Productivity issues...

- GPGPU code too verbose
- Need to hide boilerplate in libraries
- Additional resource management for GPGPU
GPGPU as a managed platform

GPGPU auto-tuning solves some problems...

- finds **fast code** and **adapts** without programming
- ...does not **manage memory** for us
- ...**low productivity** in C-like GPGPU languages

Natural solution is

- Application virtual machine
- Garbage collection
- Just-in-time **auto-tuning** compiler
- High-level programming language
An implementation of these ideas

Chai… a managed platform and language for GPGPU

- Free, open source
- Inspired by PeakStream
- Array programming language as C++ DSL
- Under development since 2010
Chai language summary

- Array is fundamental type
- Arithmetic and predicate operators
- Selection, gathering, indexes
- Data creation
- RNG (CPU only for now)
- Auto-tuned matrix multiply
- Reductions
- Math, common, relational functions (POSIX/OpenCL)
- Extensible at runtime
Sample: Data parallel reduction

double cpuA[\(P\)][N * N], cpuB[\(P\)][N * N];
vector<double*>* dataA, dataB; // parallel
for (int i = 0; i < \(P\); i++) {
    dataA.push_back(cpuA[i]);
    dataB.push_back(cpuB[i]);
}

Arrayf64 D;
{
    Arrayf64 A = make2(N, N, dataA);
    Arrayf64 B = make2(N, N, dataB);
    D = sum(matmul(A, B));
}
const vector<double> d = D.read_scalar(\(P\));
for (size_t i = 0; i < 5; i++)
{
    // managed on GPU
    Arrayf64 B;
    {
        Arrayf64 A = make1(10, cpuA);
        B = exp(A);
    }
    B.read1(cpuB, 10 * sizeof(double));

    // unmanaged on CPU
    cpuB[i] += i;
}
Sample: Conjugate-gradient (1/2)

```cpp
Arrayf32 A = Arrayf32::make2(N, N, cpuA);
Arrayf32 b = Arrayf32::make1(N, cpuB);
Arrayf32 residuals = b - matmul(A, x);
Arrayf32 p = residuals;
Arrayf32 newRR
    = dot_product(residuals, residuals);
for (iter = 0; iter < N; iter++) {
    Arrayf32 oldRR = newRR;
    Arrayf32 newX, newP, newResiduals;
    Arrayf32 Ap = matmul(A, p);
    Arrayf32 dp = dot_product(p, Ap);
    newX = x + p * oldRR / dp;
```

(PeakStream demo code)
newResiduals = residuals - Ap * oldRR / dp;
newRR = dot_product(newResiduals, newResiduals);
newP = newResiduals + p * newRR / oldRR;
p = newP;
residuals = newResiduals;
float oldRRcpu = oldRR.read_scalar();
if (oldRRcpu <= TOLERANCE)
    break;
x = newX;

(PeakStream demo code)
Basic concepts

- An odd couple: CPU and GPU
- Arithmetic intensity
- GPU memory hierarchy
- Problem blocking
An odd couple: CPU and GPU

Software design rules must be different...  

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUs</td>
<td>few</td>
<td>many</td>
</tr>
<tr>
<td>memory</td>
<td>cache controller</td>
<td>program code</td>
</tr>
<tr>
<td>control flow</td>
<td>MIMD</td>
<td>SIMD</td>
</tr>
<tr>
<td>programming</td>
<td>mixed paradigm</td>
<td>kernel/functional</td>
</tr>
<tr>
<td>dominant cost</td>
<td>time</td>
<td>space</td>
</tr>
<tr>
<td>effective</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>memory bandwidth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arithmetic intensity

Number crunching relative to amount of data:

\[
\frac{O(\text{time})}{O(\text{space})}
\]

GPUs have:

- Huge number of ALUs
- Low effective memory bandwidth

which means:

- \( O(\text{time}) \) should be large
- \( O(\text{space}) \) should be small
GPU memory hierarchy

Six kinds of memory...

- **global**: RAM on the GPU
- **texture**: Very fast but accessed with coordinates
- **local**: Fast scratchpad for running kernels
- **private**: Register variables for running kernels
**Problem blocking**

**inner blocking**  kernel registers, arithmetic intensity

**outer blocking**  GPU core work groups, local memory
Basic concepts summary

GPGPU performance needs:

1. High arithmetic intensity
2. Efficient data movement through memory hierarchy
3. Optimal blocking
   - Maximize arithmetic intensity
   - Minimize register pressure (GPU thread scheduling)
Auto-tuning

1. In isolation...
   - Expectation-Maximization
   - Curse of dimensionality
   - Memoization and journaling

2. With the JIT...
   - Everything fails
   - Dynamic auto-tuning
   - Auto-tune everything?
Expectation-Maximization

\( \alpha \) parameters outer and inner blocking

\( \beta \) parameters loop order, miscellaneous
Curse of dimensionality

Auto-tuning *doesn’t* find everything...

<table>
<thead>
<tr>
<th></th>
<th>Auto-tuned</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>problem blocking</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>loop order</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>miscellaneous</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>variable types</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>dimensions</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>data layout</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Why not? *Curse of Dimensionality!*
EM search fails to converge, so use *brute force*…

- *Fixed parameters* in outer loop
- Tuned parameters found with EM auto-tuning
- Expensive, but only have to do once
Memoization and journaling

Memoization...

- Faster search (typical recursion trick)
- Auto-tuning takes a long time (hours, days)
- Thousands of kernel variations

Journaling...

- Don’t lose time if something hangs or crashes
- Resume search from where left off
Everything fails

Will crash or hang sometime: compiler, driver, kernel

► failures impossible to predict accurately
► auto-tuning is a stress test for technology stack
► auto-tuning even more important:
  ▶ learn the good kernels ahead-of-time
  ▶ application VM and JIT avoid bad ones at runtime
Dynamic auto-tuning

Cold start ahead-of-time:

- Learn structure of compute device
- Expensive statistical optimization
- Pay this cost once up-front and re-use results

Warm start just-in-time:

- Adapt to runtime state of GPU
- Relatively cheap as structure already known
- Acceptable JIT compiler warm up overhead
No, only auto-tune *high arithmetic intensity* kernels…

- Good return on investment
  - JIT work is not cheap
  - Too much JIT limits throughput (Amdahl’s law)
- Converges easily as convexity is strong enough
  - Side-effect of good ROI
- Low arithmetic intensity kernels will always be slow
  - Side-effect of bad ROI
There’s not enough time…

- Application virtual machine
  - C++ domain specific language
  - Bytecode stack machine
  - Tracing JIT and interpreter
  - Gather/scatter scheduler
  - JIT generates OpenCL

- Memory management
  - Reference counted garbage collection
  - Tracing JIT enqueues, interpreter swizzles
  - Sticky continuation

- JIT middle-end…