CSE 599 I Accelerated Computing - Programming GPUS

Intrinsic Functions

Objective

- Learn more about intrinsic functions
- Get a sense for the breadth of intrinsic functions available in CUDA
- Introduce techniques for tuning performance-critical parts of compute-bound applications

Review: What is an Intrinsic Function?

From the programmer point of view:

- a library function

From the compiler's point of view:

- a special instruction, or an inline-able set of highly optimized instructions

The use of intrinsics can allow:

- Faster operations than it would be possible to construct with "regular" instructions
- Operations that are impossible to do with "regular" instructions, perhaps by making use of specialized hardware

Standard vs Intrinsic Floating-Point Math Functions

Standard Functions:

- Callable from both host and device

CUDA Intrinsic Functions:

- Callable only from the device
- Often offer an alternative to a standard function that is faster but has less numerical accuracy

An Example: pow!

```
__global__ void standardPow(float * x) {

    *x = powf(*x, 2.0f);  // compute x squared

}
```

Remember to use `powf` and not `pow` for floats!

```
__global__ void intrinsicPow(float * x) {

    *x = __powf(*x, 2.0f);  // compute x squared

}
```

There is no such thing as __pow

Inspecting Compiled CUDA Code

NVCC can be used to compile a *.ptx file from a *.cu file:

```
nvcc --ptx -o pow.ptx pow.cu
```

A PTX file can be read with any text reader / editor

This allows one to see how nvcc translates high-level CUDA C code into device instructions

Standard (powf) PTX

```
        // .globl   _Z8standardPf

.visible .entry _Z8standardPf(
        .param .u64 _Z8standardPf_param_0
)
{
        .reg .pred  %p<18>;
        .reg .f32   %f<100>;
        .reg .b32   %r<10>;
        .reg .b64   %rd<3>;


        ld.param.u64            %rd2, [_Z8standardPf_param_0];
        cvta.to.global.u64      %rd1, %rd2;
        mov.f32     %f19, 0f3F800000;
        cvt.rzi.f32.f32         %f20, %f19;
        add.f32     %f21, %f20, %f20;
        mov.f32     %f22, 0f40000000;
        sub.f32     %f23, %f22, %f21;
        abs.f32     %f1, %f23;
        ldu.global.f32          %f2, [%rd1];
        abs.f32     %f3, %f2;
        setp.lt.f32 %p2, %f3, 0f00800000;
        mul.f32     %f24, %f3, 0f4B800000;
        selp.f32    %f25, 0fC3170000, 0fC2FE0000, %p2;
        selp.f32    %f26, %f24, %f3, %p2;
        mov.b32      %r1, %f26;
        and.b32     %r2, %r1, 8388607;
        or.b32      %r3, %r2, 1065353216;
        mov.b32      %f27, %r3;
        shr.u32     %r4, %r1, 23;
        cvt.rn.f32.u32          %f28, %r4;
        add.f32     %f29, %f25, %f28;
        setp.gt.f32 %p3, %f27, 0f3FB504F3;
        mul.f32     %f30, %f27, 0f3F000000;
        add.f32     %f31, %f29, 0f3F800000;
        selp.f32    %f32, %f30, %f27, %p3;
        selp.f32    %f33, %f31, %f29, %p3;
        add.f32     %f34, %f32, 0fBF800000;
        add.f32     %f16, %f32, 0f3F800000;
        // inline asm
        rcp.approx.ftz.f32 %f15,%f16;
        // inline asm
        add.f32     %f35, %f34, %f34;
        mul.f32     %f36, %f15, %f35;
        mul.f32     %f37, %f36, %f36;
        mov.f32     %f38, 0f3C4CAF63;
        mov.f32     %f39, 0f3B18F0FE;
        fma.rn.f32  %f40, %f39, %f37, %f38;
        mov.f32     %f41, 0f3DAAAABD;
        fma.rn.f32  %f42, %f40, %f37, %f41;
        mul.rn.f32  %f43, %f42, %f37;
        mul.rn.f32  %f44, %f43, %f36;
        sub.f32     %f45, %f34, %f36;
        neg.f32     %f46, %f36;
        add.f32     %f47, %f45, %f45;
        fma.rn.f32  %f48, %f46, %f34, %f47;
        mul.rn.f32  %f49, %f15, %f48;
        add.f32     %f50, %f44, %f36;
        sub.f32     %f51, %f36, %f50;
        add.f32     %f52, %f44, %f51;
        add.f32     %f53, %f49, %f52;
        add.f32     %f54, %f50, %f53;
        sub.f32     %f55, %f50, %f54;
        add.f32     %f56, %f53, %f55;
        mov.f32     %f57, 0f3F317200;
        mul.rn.f32  %f58, %f33, %f57;
        mov.f32     %f59, 0f35BFBE8E;
        mul.rn.f32  %f60, %f33, %f59;
        add.f32     %f61, %f58, %f54;
        sub.f32     %f62, %f58, %f61;
        add.f32     %f63, %f54, %f62;
        add.f32     %f64, %f56, %f63;
        add.f32     %f65, %f60, %f64;
        add.f32     %f66, %f61, %f65;
        sub.f32     %f67, %f61, %f66;
        add.f32     %f68, %f65, %f67;
        mul.rn.f32  %f69, %f22, %f66;
        neg.f32     %f70, %f69;
        fma.rn.f32  %f71, %f22, %f66, %f70;
        fma.rn.f32  %f72, %f22, %f68, %f71;
        mov.f32     %f73, 0f00000000;
        fma.rn.f32  %f74, %f73, %f66, %f72;
        add.rn.f32  %f75, %f69, %f74;
        neg.f32     %f76, %f75;
        add.rn.f32  %f77, %f69, %f76;
        add.rn.f32  %f78, %f77, %f74;
        mov.b32      %r5, %f75;
        setp.eq.s32 %p4, %r5, 1118925336;
        add.s32     %r6, %r5, -1;
        mov.b32      %f79, %r6;
        add.f32     %f80, %f78, 0f37000000;
        selp.f32    %f81, %f79, %f75, %p4;
        selp.f32    %f4, %f80, %f78, %p4;
        mul.f32     %f82, %f81, 0f3FB8AA3B;
        cvt.rzi.f32.f32         %f83, %f82;
        mov.f32     %f84, 0fBF317200;
        fma.rn.f32  %f85, %f83, %f84, %f81;
        mov.f32     %f86, 0fB5BFBE8E;
        fma.rn.f32  %f87, %f83, %f86, %f85;
        mul.f32     %f18, %f87, 0f3FB8AA3B;
        // inline asm
        ex2.approx.ftz.f32 %f17,%f18;
        // inline asm
        add.f32     %f88, %f83, 0f00000000;
        ex2.approx.f32          %f89, %f88;
        mul.f32     %f90, %f17, %f89;
        setp.lt.f32 %p5, %f81, 0fC2D20000;
        selp.f32    %f91, 0f00000000, %f90, %p5;
        setp.gt.f32 %p6, %f81, 0f42D20000;
        selp.f32    %f98, 0f7F800000, %f91, %p6;
        setp.eq.f32 %p7, %f98, 0f7F800000;
        @%p7 bra    BB0_2;

        fma.rn.f32  %f98, %f98, %f4, %f98;

BB0_2:
        setp.lt.f32 %p8, %f2, 0f00000000;
        setp.eq.f32 %p9, %f1, 0f3F800000;
        and.pred    %p1, %p8, %p9;
        mov.b32      %r7, %f98;
        xor.b32     %r8, %r7, -2147483648;
        mov.b32      %f92, %r8;
        selp.f32    %f99, %f92, %f98, %p1;
        setp.eq.f32 %p10, %f2, 0f00000000;
        @%p10 bra   BB0_5;
        bra.uni     BB0_3;

BB0_5:
        add.f32     %f95, %f2, %f2;
        selp.f32    %f99, %f95, 0f00000000, %p9;
        bra.uni     BB0_6;

BB0_3:
        setp.geu.f32%p11, %f2, 0f00000000;
        @%p11 bra   BB0_6;

        cvt.rzi.f32.f32         %f94, %f22;
        setp.neu.f32%p12, %f94, 0f40000000;
        selp.f32    %f99, 0f7FFFFFFF, %f99, %p12;

BB0_6:
        add.f32     %f96, %f3, 0f40000000;
        mov.b32      %r9, %f96;
        setp.lt.s32 %p14, %r9, 2139095040;
        @%p14 bra   BB0_11;

        setp.gtu.f32%p15, %f3, 0f7F800000;
        @%p15 bra   BB0_10;
        bra.uni     BB0_8;

BB0_10:
        add.f32     %f99, %f2, 0f40000000;
        bra.uni     BB0_11;

BB0_8:
        setp.neu.f32%p16, %f3, 0f7F800000;
        @%p16 bra   BB0_11;

        selp.f32    %f99, 0fFF800000, 0f7F800000, %p1;

BB0_11:
        setp.eq.f32 %p17, %f2, 0f3F800000;
        selp.f32    %f97, 0f3F800000, %f99, %p17;
        st.global.f32           [%rd1], %f97;
        ret;
}
```

Intrinsic (__powf) PTX

```
     // .globl _Z9intrinsicPf
.visible .entry _Z9intrinsicPf(
     .param .u64 _Z9intrinsicPf_param_0
)
{
     .reg .f32 %f<5>;
     .reg .b64 %rd<3>;


     ld.param.u64    %rd1, [_Z9intrinsicPf_param_0];
     cvta.to.global.u64   %rd2, %rd1;
     ld.global.f32   %f1, [%rd2];
     lg2.approx.f32 %f2, %f1;
     add.f32    %f3, %f2, %f2;
     ex2.approx.f32 %f4, %f3;
     st.global.f32   [%rd2], %f4;
     ret;
}
```

Some Other CUDA Single Precision Intrinsics

Standard Code	Intrinsic
`r = cosf(x);`	`r = __cosf(x);`
`r = expf(x);`	`r = __expf(x);`
`r = x / y;`	`r = __fdividef(x,y);`
`r = logf(x);`	`r = __logf(x);`
`r = max(0.f,min(x,1.f);`	`r = __saturatef(x);`
`r = cosf(x);`	`r = __cosf(x);`
`r = sinf(x);`	`r = __sinf(x);`
`r = tanf(x);`	`r = __tanf(x);`
`s = sinf(x);` `c = cosf(x);`	`float s, c;` `__sincosf(x, &s, &c);`

CUDA Single Precision Intrinsics with Rounding

Suffix	Function
`rd`	Round down to the next representable value
`rn`	Round to the nearest representable value
`ru`	Round up to the next representable value
`rz`	Round to the next representable value in the direction of zero

For example, to add two floats in round-towards-zero mode, you could use:

```
r = __fadd_rz(x,y);
```

Standard Code	Intrinsic
`r = x + z;`	`r = __fadd_*(x,y);`
`r = x / z;`	`r = __fdiv_*(x,y);`
`r = x * y + z;`	`r = __fmaf_*(x,y,z);`
`r = x * y;`	`r = __ful_*(x,y);`
`r = 1.f / x;`	`r = __frcp_*(x);`
`r = 1.f / sqrtf(x);`	`r = __frsqrt_rn(x);`
`r = sqrtf(x);`	`r = __fsqrt_*(x);`
`r = x - y;`	`y = __fsub_*(x);`

Some CUDA Integer Intrinsics

Intrinsic	Function
`r = __brev(x);`	Reverse the bit order of x
`r = __clz(x);`	Count the number of consecutive high-order zero bits in x
`r = __ffs(x);`	Find the position of the least-significant set bit in x
`r = __popc(x);`	Count the number of set bits in x
`r = __sad(x,y,z);`	Compute | x - y | + z
`r = __mulhi(x,y);`	Compute the most-significant 32 bits of the result of multiplying x and y
`r = __rhadd(x,y);`	Compute the average of x and y using (x + y + 1) >> 1

There are also 64-bit version of many of these (append “`ll`”), and unsigned versions (prepend “`u`”)

Some Intrinsics are Automatically Used by nvcc

For example, fused-multiply-and-add (FMAD) is used automatically by the compiler:

```
__global__ void standard(float * x, float * y, float * z) {
     *x = (*x) * (*y) + (*z);
}

__global__ void intrinsic(float * x, float * y, float * z) {
     *x = __fmaf_rn(*x, *y, *z);
}
```

Both result in the following PTX code:

```
...
ldu.global.f32  %f1, [%rd6];
ldu.global.f32  %f2, [%rd5];
ldu.global.f32  %f3, [%rd4];
fma.rn.f32      %f4, %f1, %f2, %f3;
...
```

Tuning Instruction Generation with Compiler Flags

Now we'll compile this file passing the flag "`--fmad=false`" to nvcc:

```
__global__ void standard(float * x, float * y, float * z) {
     *x = (*x) * (*y) + (*z);
}

__global__ void intrinsic(float * x, float * y, float * z) {
     *x = __fmaf_rn(*x, *y, *z);
}
```

Intrinsic:

```
...
ldu.global.f32  %f1, [%rd6];
ldu.global.f32  %f2, [%rd5];
ldu.global.f32  %f3, [%rd4];
fma.rn.f32      %f4, %f1, %f2, %f3;
...
```

Faster

Standard:

```
...
ldu.global.f32  %f1, [%rd6];
ldu.global.f32  %f2, [%rd5];
mul.rn.f32      %f3, %f1, %f2;
ldu.global.f32  %f4, [%rd4];
add.rn.f32      %f5, %f3, %f4;
...
```

More accurate

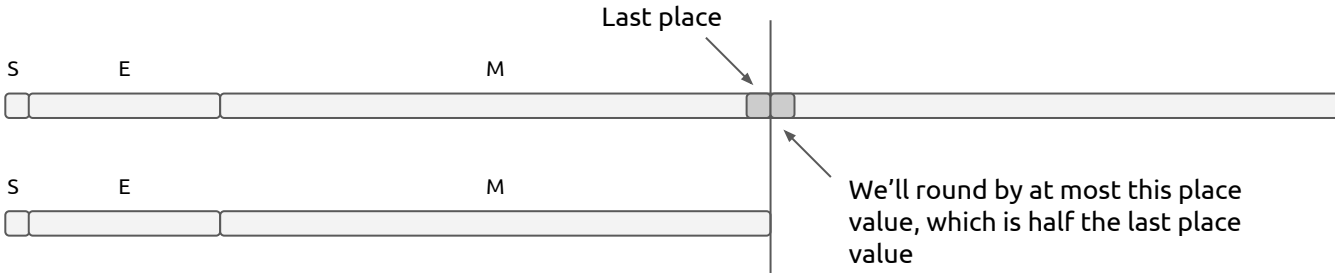
nvcc Performance vs Accuracy Flags

Flag	Description	Default
--ftz=[true,false]	“Flush-to-zero”; All de-normalized floating point values are set to zero. This allows the code to avoid checking for and specialized handling of de-normalized floats	false
--prec-div=[true,false]	“Precision division”; If true, divisions and reciprocals are IEEE compliant. Otherwise, they are faster	true
--prec-sqrt=[true,false]	“Precision square root”; If true, square roots are IEEE compliant. Otherwise, they are faster	true
--fmad=[true,false]	“Fused multiply-and-add”; If true, the compiler will use the specialized fmaf instruction	true
--use_fast_math	Equivalent to setting “--ftz=true”, “--prec-div=false”, and “--prec-sqrt=false”. Also replaces all standard functions with intrinsics, where possible	not set

Quantifying the Loss of Accuracy

Accuracy bounds for CUDA hardware are well documented

For example, floating point division is IEEE-compliant, meaning it is computed to within 0.5 ULP



`__fdivf_[rn,ru,rd,rz](x,y)` is also IEEE-compliant, it just allows you to specify the rounding mode

`__fdividef(x,y)` has error within 2 ULP for $2^{-126} \leq y \leq 2^{126}$

See the CUDA Programming Guide for error bounds on other intrinsics

Some Conversion Intrinsics

`r = __double_as_longlong(x);`	Reinterprets double-precision value x as a 64-bit int; equivalent to: `r = *((unsigned long long int *)&x);`
`r = __longlong_as_double(x);`	Reinterprets 64-bit int x as a double-precision value; equivalent to: `r = *((double *)&x);`
`r = __float2half(x);`	Converts single-precision value x to half-precision
`r = __floats2half2_rn(x, y);`	Converts two single-precision values x and y to half-precision, returning them as a single __half2
`r = __half22float2(x);`	Converts __half2 x to a float2

Half-precision Math!

Half-precision representations (i.e. 16-bit floating point values) are becoming more popular, especially with deep learning

Recent devices have been pushing efficient half-precision support

CUDA provides a half-precision type, called `half`

However, for making efficient use of a system optimized for 32-bit words, it's generally better to use pairs of half-precision values, i.e. `half2`

Half-Precision Math is Heavily Intrinsic-Dependent

```
__global__
void haxpy(int n, half a, const half *x, half *y) {
      int start = threadIdx.x + blockDim.x * blockIdx.x;
      int stride = blockDim.x * gridDim.x;

#if __CUDA_ARCH__ >= 530
      int n2 = n/2;
      half2 *x2 = (half2*)x, *y2 = (half2*)y;

      for (int i = start; i < n2; i+= stride)
            y2[i] = __hfma2(__halves2half2(a, a), x2[i], y2[i]);

      // first thread handles singleton for odd arrays
      if (start == 0 && (n%2))
            y[n-1] = __hfma(a, x[n-1], y[n-1]);

#else
      for (int i = start; i < n; i+= stride) {
            y[i] = __float2half(__half2float(a) * __half2float(x[i]) + __half2float(y[i]));
      }
#endif
}
```

Source: https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/

Other Intrinsics We've Seen So Far

__syncthreads
__threadfence
tex1D
atomicAdd
atomicSub
atomicExch
atomicMin
atomicMax
atomicInc
atomicDec
atomicCAS
atomicAnd
atomicOr
atomicXor

Intrinsics Allow Access to Specialized Hardware Instructions

CUDA C Code	PTX
`__syncthreads();`	`bar.sync  0;`
`__threadfence();`	`membar.gl;`
`tex1D(textureName,1);`	`mov.f32  %f1, 0f3F800000;` `tex.1d.v4.s32.f32  {%r1, %r2, %r3, %r4}, [textureName, {%f1}];`

Roll Your Own Atomic Add

atomicCAS can be used to implement any custom atomic operation

From nVidia documentation:

```
unsigned int atomicCAS(unsigned int * address, unsigned int compare, unsigned int val);
```

> “reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old == compare ? val : old) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap).”

```
__device__ int diyAtomicAdd(int * address, const int increment) {
	// see what's there already
	int expected = *address;
	int oldValue = atomicCAS(address, expected, expected + increment);
	// keep trying until it goes through
	while (oldValue != expected) {
		expected = oldValue;
		oldValue = atomicCAS(address, expected, expected + increment);
	}
	return oldValue;
}
```

DIY Atomic Add PTX

```
     ld.param.u64    %rd2, [_Z18diyForcingFunctionPii_param_0];
     ld.param.u32    %r4, [_Z18diyForcingFunctionPii_param_1];
     cvta.to.global.u64   %rd1, %rd2;
     ld.global.u32   %r5, [%rd1];
     add.s32    %r6, %r5, %r4;
     atom.global.cas.b32  %r8, [%rd1], %r5, %r6;
     setp.eq.s32     %p1, %r8, %r5;
     @%p1 bra   BB1_2;

BB1_1:
     mov.u32    %r2, %r8;
     add.s32    %r7, %r2, %r4;
     atom.global.cas.b32  %r8, [%rd1], %r2, %r7;
     setp.ne.s32     %p2, %r8, %r2;
     @%p2 bra   BB1_1;

BB1_2:
     ret;
```

Intrinsic Atomic Add

code:

```
__device__ int intrinsicAtomicAdd(int * address, const int increment) {
     atomicAdd(address,increment);
}
```

PTX:

```
     ld.param.u64    %rd1, [_Z24intrinsicForcingFunctionPii_param_0];
     ld.param.u32    %r1, [_Z24intrinsicForcingFunctionPii_param_1];
     cvta.to.global.u64   %rd2, %rd1;
     atom.global.add.u32  %r2, [%rd2], %r1;
     ret;
```

This is because of UVA! It converts a generic virtual global device address into an address that can be used to access device memory on this GPU

Loop is implemented in hardware!

Some Additional Intrinsics

`__ldg()`	Used to load data that will be read-only for the duration of the kernel (often the compiler can detect this condition, but not always)
`__shfl()`	Allows threads to share values in local registers; we'll cover this topic in detail in the next lecture!
`__threadfence_block()`	Like threadfence, but only ensures that memory reads and writes are visible to before the call are visible to other threads in the block (rather than all threads in the grid)
`__syncthreads_count(predicate)`	In addition to synchronizing, broadcast to all threads the number of threads where the predicate is true
`__syncthreads_and(predicate)`	In addition to synchronizing, broadcast value 1 to all threads if predicate was true for all threads, otherwise 0
`__syncthreads_or(predicate)`	In addition to synchronizing, broadcast value 1 to all threads if predicate was true for any thread, otherwise 0
`__vadd4(a,b);`	Performs per-byte addition of unsigned 32-bit integers a and b. There are a variety of other SIMD intrinsics for integer values

Data-Parallel Graph Search Revisited

```
do {
      if (threadIdx.x == 0) {
            sharedAnyVisited = false;
      }
      __syncthreads();

      bool justVisited = false;
      for (int edge = incomingEdgesStart[vertex]; edge < incomingEdgesStart[vertex] ++edge) {

            // check if source was visited in the last round
            bool sourceVisitedLastRound = ...

            justVisited |= sourceVisitedLastRound;

      }

      if (justVisited) {
            // set distance, mark as visited
            ...

            sharedAnyVisited = true;
      }

      __syncthreads();

      // swap buffers, etc.
      ...

} while (sharedAnyVisited);
```

Data-Parallel Graph Search Revisited

```
bool anyVisited;
do {

      bool justVisited = false;
      for (int edge = incomingEdgesStart[vertex]; edge < incomingEdgesStart[vertex] ++edge) {

            // check if source was visited in the last round
            bool sourceVisitedLastRound = ...

            justVisited |= sourceVisitedLastRound;

      }

      if (justVisited) {
            // set distance, mark as visited
            ...
      }

      anyVisited = __syncthreads_or(justVisited);

      // swap buffers, etc.
      ...

} while (anyVisited);
```

Notes on PTX

PTX is actually an intermediate assembly language

To check instructions in a binary, use:

```
nvcc -Xptxas=-v -cubin *.cu
```

Then, inspect the resulting binary using:

```
cuobjdump -sass *.cubin
```

There may be additional optimization that happens in targeting a particular device

Conclusion / Takeaways

- There are MANY intrinsic functions in CUDA
  - More generally, specialized hardware and intrinsic functions tend to go hand-in-hand
- There are a variety of ways to trade off performance and accuracy of floating point computation in CUDA
  - Representation-wise: float vs. double. vs half
  - Computation-wise: standard vs. intrinsic functions
- Intrinsic functions allow one to do things that wouldn't be possible otherwise, or to do things with a small instruction count that would otherwise take a number of instructions
- One can inspect compilations using *.ptx files and a text editor or *.cubin files and cuobjdump

Further Reading / Reference

CUDA Math API Documentation:

http://docs.nvidia.com/cuda/cuda-math-api/index.html#axzz4fxtDexzg

CUDA Programming Guide Appendix D:

http://docs.nvidia.com/cuda/cuda-c-programming-guide/#mathematical-functions-appendix

Sources

http://docs.nvidia.com

Cheng, John, Max Grossman, and Ty McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.

Wilt, Nicholas. The cuda handbook: A comprehensive guide to gpu programming. Pearson Education, 2013.