The Importance of Kernels for Performance Portability,
Or, How I Learned to Stop Looping and Love the Kernel

September, 2020

Tom Scogland, David Richards
The Volta Compute Unit

Major Features include:

- New mixed-precision Tensor Cores purpose-built for deep learning matrix arithmetic, delivering 12x TFLOPS for training, compared to GP100, in the same power envelope
- 50% higher energy efficiency on general compute workloads
- Enhanced high performance L1 data cache
- A new SIMT thread model that removes limitations present in previous SIMT and SIMD processor designs

NVIDIA TESLA V100 GPU ARCHITECTURE – whitepaper WP-08608-001_v1.1
The GCN Compute Unit (CU)

- **SIMD Units**
  - 4x SIMD vector units (each 16 lanes wide)
  - 4x 64KB (256KB total) Vector General-Purpose Register (VGPR) file
    - Maximum of 256 registers per SIMD – each register is 64x 4-byte entries
  - Instruction buffer for 10 wavefronts on each SIMD unit
    - Each wavefront is local to a single SIMD unit, not spread among the four (more on this in a moment)
This is all Seymour’s fault*

- Vector machines begat the SIMD machines of today, even if they were pipelined

- Early compilers either added explicit extensions for vectorization
  - Vector LRLTRAN
  - Vectra
  - Etc.

- Or vectorized sequential loops and ultimately array expressions descended from the extensions

- Trouble was, vectorizing sequential code is hard

*It’s not really Seymour Cray’s fault. He didn’t invent vector processing. But he did create the successful vector processor.
The “Killer Micros” soon gave us SIMD vectors

- SIMD instruction sets were widely introduced in mid-90’s (MMX, 3DNow, SSE, etc.)

- Instead of pipelining, early microprocessor SIMD:
  - Added wide SIMD registers
  - Processed elements in those registers in parallel (usually)
  - Had little support for cross-lane operations (reduction, swizzle, gather, scatter)

- Heavily reliant on data contiguous in memory

- Sequential code still hard even harder to vectorize
  - Relied heavily on programmers to manually vectorize
  - Compilers are notoriously conservative
  - Slow code is better than incorrect code
  - C is more popular than fortran by this point, C is much harder to autovectorize

Many programmers ignored SIMD units for many years because they were just so hard to use
void update_jGate(double dt, int nCells, double *VM, double *g, double *mhu_a, double *tauR_a)
{
    int gateIndex=2;
    int mhu_l= 7 ;
    int mhu_m= 9;
    int tauR_l= 1;
    int tauR_m=13;
    double tauRdt_a[tauR_m];
    for (int j=tauR_m-1;j>=0;j--) tauRdt_a[j] = tauR_a[j]*dt;
    int mhu_k = mhu_m+mhu_l-1;
    int tauR_k = tauR_m+tauR_l-1;
    for (int ii=0;ii<nCells;ii++)
    {
        double x = VM[ii];
        double sum1=0;
        double sum2=0;
        double sum3=0;
        for (int j=mhu_m-1;j>=0 ;j--)sum1 = mhu_a[j] + x*sum1;
        for (int j=mhu_k ;j>=mhu_m;j--)sum2 = mhu_a[j] + x*sum2;
        for (int j=tauR_m-1;j>=0 ;j--)sum3 = tauRdt_a[j] + x*sum3;
        double tauRdt= sum3;
        double mhu= sum1/sum2;
        g[ii] += mhu*tauRdt - g[ii]*tauRdt;
    }
}

SIMD code can be a nightmare to write and maintain

**Original Code**

```c
void update_jGate(double dt, int nCells, double *VM, double *g, double *mhu_a, double *tauR_a)
{
    int gateIndex=2;
    int mhu_l= 7 ;
    int mhu_m= 9;
    int tauR_l= 1;
    int tauR_m=13;
    double tauRdt_a[tauR_m];
    for (int j=tauR_m-1;j>=0;j--) tauRdt_a[j] = tauR_a[j]*dt;
    int mhu_k = mhu_m+mhu_l-1;
    int tauR_k = tauR_m+tauR_l-1;
    for (int ii=0;ii<nCells;ii++)
    {
        double x = VM[ii];
        double sum1=0;
        double sum2=0;
        double sum3=0;
        for (int j=mhu_m-1;j>=0 ;j--)sum1 = mhu_a[j] + x*sum1;
        for (int j=mhu_k ;j>=mhu_m;j--)sum2 = mhu_a[j] + x*sum2;
        for (int j=tauR_m-1;j>=0 ;j--)sum3 = tauRdt_a[j] + x*sum3;
        double tauRdt= sum3;
        double mhu= sum1/sum2;
        g[ii] += mhu*tauRdt - g[ii]*tauRdt;
    }
}
```

**After SIMDization**

```c
void update_jGate(double dt, int nCells, double *VM, double *g, double *mhu_a, double *tauR_a)
{
    vector4double v_xa, sum1,sum2,sum3.
    .
    .
    v_sum1a = vec_madd(v_xa, v_sum1a, v_mhu_A2);
    v_sum1a = vec_madd(v_xa, v_sum1a, v_mhu_A1);
    v_sum2a = vec_madd(v_xa, v_sum2a, v_mhu_B2);
    v_sum2a = vec_madd(v_xa, v_sum2a, v_mhu_B1);
    v_sum3a = vec_madd(v_xa, v_sum3a, v_tauRdt_C2);
    v_sum3a = vec_madd(v_xa, v_sum3a, v_tauRdt_C1);
    for (int ii=0;ii<nCells;ii++)
    {
        double x = VM[ii];
        double sum1=0;
        double sum2=0;
        double sum3=0;
        for (int j=mhu_m-1;j>=0 ;j--)sum1 = mhu_a[j] + x*sum1;
        for (int j=mhu_k ;j>=mhu_m;j--)sum2 = mhu_a[j] + x*sum2;
        for (int j=tauR_m-1;j>=0 ;j--)sum3 = tauRdt_a[j] + x*sum3;
        double tauRdt= sum3;
        double mhu= sum1/sum2;
        g[ii] += mhu*tauRdt - g[ii]*tauRdt;
    }
}
```

27 lines of code vs. 537 lines of code
Even Simple Code, Like DAXPY

**Sequential**

```c
for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
```

**After SIMDization**

```c
while (n >= 2) {
    n -= 2;
    _mm_store_pd(y + n, _mm_add_pd(_mm_load_pd(y + n),
                              _mm_mul_pd(_mm_load_pd(x + n), _mm_load1_pd(&a))));
}
```

Even the simplest code becomes tricky, and non-portable, with simd intrinsics
GPUs gave us wider vectors but also a new programming model

- Step 1 of GPU porting is converting loops to kernels

**Sequential**

```c
void daxpy(int n, double a, double *x, double *y){
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

**CUDA**

```c
__global__void daxpy_knl(int n, double a,
                          double *x, double *y){
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```

**OpenCL**

```c
__kernel void daxpy(int n, double a, _global double *src, _global double *dst)
{
    int i = get_global_id(0);
    if (i < n) y[i] = a * x[i] + y[i];
}
```

**Brook+**

```c
kernel void daxpy(int n, double a,
                  double x<> , out double y<> ) {
    int i = instance().x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```
GPUs gave us wider vectors but also a new programming model

- Step 1 of GPU porting is converting loops to kernels

```c
void daxpy(int n, double a, double *x, double *y){
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

```c
__global__ void daxpy_knl(int n, double a, double *x, double *y){
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```

```c
__kernel void daxpy(int n, double a, __global double *src, __global double *dst){
    int i = get_global_id(0);
    if (i < n) y[i] = a * x[i] + y[i];
}
```

```c
kernel void daxpy(int n, double a, double x<>, out double y<>){
    int i = instance().x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```

It is easy to overlook the significant differences between kernels and loops (and the execution model that goes with them).
GPU Kernels Always Vectorize

- A GPU model has no scalar context
- A CPU model has a native scalar context but allows vector instructions in that context
- Kernel code always vectorizes
  - For some meaning of vectorize
  - No compiler intelligence necessary

```c
__global__ void daxpy_knl(int n, double a,
                           double *x, double *y)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) y[i] = a * x[i] + y[i];
}
```
What instructions does gcc generate for a basic loop?

```c
void naive_memcpy(char * dst, char * src, size_t n){
    for (int i=0; i<n; i++)
        dst[i] = src[i];
}
```

One operation, one byte at a time (move, test, branch)

*With reasonable flags: -O2 -march=haswell -fopenmp
What Happens With an OMP parallel for Loop?

```c
void myMemCpy4(char * dst, char * src, size_t n) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i)
        dst[i] = src[i];
}
```
What PTX does nvcc generate for a similar kernel?

```c
__global__ void mymemcpy(char *dest, char *src, size_t n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n)
        dest[tid] = src[tid];
}
```

- Same memcpy as a cuda kernel
- PTX looks similar to sequential loop
- **But** these are *vector* instructions across warps
- There are no truly sequential instructions here
Making a kernel behave sequentially

- If we want sequential behavior it has to be **explicit**

- Parallel and *unsequenced* by default

```c
__global__ void sequential_mc(char *dst, char *src, size_t n)
{
    if (thread_id == 0) {
        for(int i = 0; i < n; ++i)
            dst[i] = src[i]
    }
}
```

- Far more obvious this is a Bad Idea™

- Generates **74** PTX instructions to the 17 in the unsequenced
GPU models aren’t the only way to write kernels: OpenMP

```c
void mc_omp_simd(char * dest, char * src, size_t n) {
  // Copy contents of src[] to dest[]
  #pragma omp simd
  for(int i = 0; i < n; ++i)
    dest[i] = src[i];
}
```

- Now unsequenced (thanks to `simd` directive)
- Generates three loops
  - 256-bit vectors
  - 128-bit vectors
  - Individual bytes
- Ties them together to do the most efficient thing for a length
GPU models aren’t the only way to write kernels: C++17/RAJA/Kokkos

void myMemCpy6(char * dest, char * src, size_t n) {
    auto r = RAJA::RangeSegment(0, n);
    std::for_each(
        std::execution::par_unseq, r.begin(), r.end(),
        [=](size_t i) {
            dest[i] = src[i];
        });
}

- Similarly vectorized
- Only two loops this time, 256-bit and per-byte

C++17 execution policies can express parallel unsequenced kernels as well
C++17 and RAJA give flexibility through policies

- Serial sequenced or unsequenced:
  - RAJA::simd or std::unseq: unsequenced kernel, vector safe
  - RAJA::loop or std::seq: sequenced, order matters, compiler must prove or provide safety

- Parallelism is another axis entirely:
  - RAJA::omp_parallel_for_exec or std::par: run in parallel but still sequenced in each thread
  - RAJA::omp_parallel_for_simd_exec or std::par_unseq: both parallel and unsequenced execution are safe

Logically a “kernel”, policy determines strength of guarantees on the spectrum from loop to unsequenced parallel kernel
Kernels that use team/block synchronization have weaknesses too

- Full context (stack, registers) must persist for each logical thread across the synchronization
- Huge context switch and flushing/loading cost to make progress for many logical threads (500+)
- This is why Apple OpenCL for CPU only supported one thread per workgroup
- If there are no barriers (syncthreads, workgroup_sync, etc.) there’s no cost, if there is CPU may suffer while GPU does not

Conclusions

- Eliminating sequential scalar loops from your code and writing in “kernel” or an unsequenced parallel form is inherently Performant, Portable, and Productive across vendors and architectures.

- **But** it’s not a silver bullet
  - Data organization and motion are still a concern
  - User must prove the safety and lack of races or data ordering issues, compiler is absolved of dependencies between iterations, lanes, or threads
  - Synchronization can have widely varying costs at different scopes on different architectures

- Overall, learn to love the kernel
We need to keep three promises in this talk

1. Explain why the kernel programming model produces code that always vectorizes
   - Slides 1-6 give background and explain the vectorization challenge

2. Compare RAJA forall, Kokkos parallel_for, and OpenMP omp loop to omp for
   - Omp for is a set of sequential loops in parallel
   - This is where your code examples come in

3. Discuss weaknesses of kernels
   - And why weaknesses are outweighed by advantages

Conclusion: Eliminating sequential scalar loops from your code and writing in kernel form is inherently Performant, Portable, and Productive across vendors and architectures
   - But it’s not a silver bullet. You still need to worry about data organization and motion
Cutting room floor

- We could talk about array programming
  - Fortran 90 looks like an example, but really isn’t.
  - We would talk to John Levesque about why Fortran vector notation isn’t nearly as nice as you think it might be for OpenMP programming.

- 925-577-9005