The Importance of Kernels for Performance Portability,

Or, How I Learned to Stop Looping and Love the Kernel

Tom Scogland, David Richards

September, 2020

LLNL-PRES-813854

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

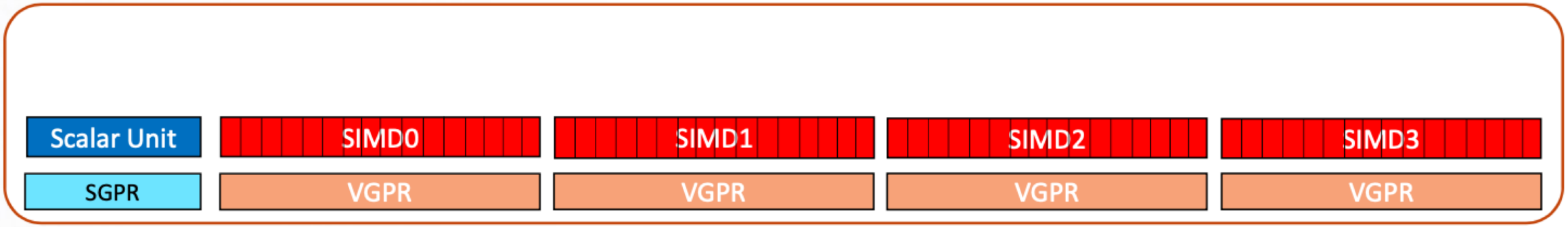
The Volta Compute Unit

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

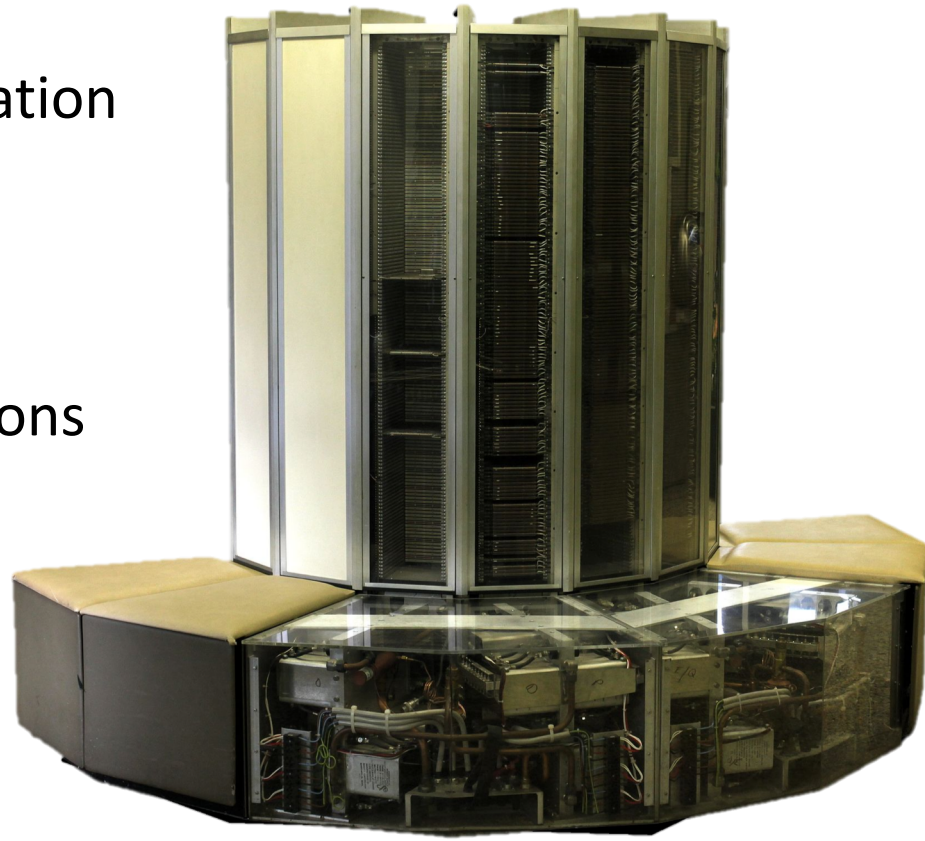
The GCN Compute Unit (CU)



- 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.

Cray 1 at EPFL: By Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr, CC BY-SA 2.0 fr, https://commons.wikimedia.org/w/index.php?curid=14619982

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

SIMD code can be a nightmare to write and maintain

Original Code

```
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;
  }
}
```

27 lines of code

After SIMDization

```
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);
   sum1[ii] = mhu_a[0]     + VM[ii]*sum1[ii];//BODYCOM0
   sum2[ii] = mhu_a[10]    + VM[ii]*sum2[ii];//BODYCOM0
   sum2[ii] = mhu_a[9]     + VM[ii]*sum2[ii];//BODYCOM0
   sum3[ii] = tauRdt_a[13] + VM[ii]*sum3[ii];//BODYCOM0
   sum3[ii] = tauRdt_a[12] + VM[ii]*sum3[ii];//BODYCOM0
   v_mhu_A1    = vec_lds(0, &mhu_a[0]);//BODYCOM0
   v_mhu_B1    = vec_lds(0, &mhu_a[9]);//BODYCOM0
   v_mhu_B2    = vec_lds(0, &mhu_a[10]);//BODYCOM0
   v_tauRdt_C1 = vec_lds(0, &tauRdt_a[12]);//BODYCOM0
   v_tauRdt_C2 = vec_lds(0, &tauRdt_a[13]);//BODYCOM0
     .
     .
     .
}
```

537 lines of code

Even Simple Code, Like DAXPY

Sequential

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

After SIMDization

```
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

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

OpenCL

```
__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];
}
```

CUDA

```
__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];
}
```

Brook+

```
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

Sequential

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

OpenCL

```
__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];
}
```

CUDA

```
__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];
}
```

Brook+

```
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

```
__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];
}
```

```
exmemcpy(char*, char*, unsigned long):
        mov     rcx, rdx
        test    rdx, rdx
        je      .L8
        xor     eax, eax
.L3:
        movzx   edx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], dl
        inc     rax
        cmp     rax, rcx
        jne     .L3
.L8:
        ret
```

*With reasonable flags: -O2 –march=haswell -fopenmp

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

What Happens With an OMP parallel for Loop?

```
void myMemCpy4(char * dst,
               char * src,
               size_t n) {
  #pragma omp parallel for
  for(int i = 0; i < n; ++i)
    dst[i] = src[i];
}
```

Loop over chunks

Déjà vu. Same serial code executed for a chunk

```
        call    omp_get_thread_num
        mov     ecx, eax
        mov     rax, QWORD PTR [rbx+16]
        cdq
        idiv    ebp
        cmp     ecx, edx
        jl      .L2
.L5:
        imul    ecx, eax
        add     edx, ecx
        add     eax, edx
        cmp     edx, eax
        jge     .L7
        mov     rdi, QWORD PTR [rbx+8]
        mov     rsi, QWORD PTR [rbx]
        movsx   rdx, edx
.L4:
        movzx   ecx, BYTE PTR [rdi+rdx]
        mov     BYTE PTR [rsi+rdx], cl
        inc     rdx
        cmp     eax, edx
        jg      .L4
.L7:
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
.L2:
        inc     eax
        xor     edx, edx
        jmp     .L5
```

What PTX does nvcc generate for a similar kernel?

```
__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];
}
```

```
8   ld.param.u64    %rd1, [_Z8mymemcpyPcS_m_param_0];
9   ld.param.u64    %rd2, [_Z8mymemcpyPcS_m_param_1];
10  ld.param.u64    %rd3, [_Z8mymemcpyPcS_m_param_2];
11  mov.u32         %r2, %ctaid.x;
12  mov.u32         %r3, %ntid.x;
13  mov.u32         %r4, %tid.x;
14  mad.lo.s32      %r1, %r3, %r2, %r4;
15  cvt.s64.s32     %rd4, %r1;
16  setp.ge.u64     %p1, %rd4, %rd3;
17  @%p1 bra        BB0_2;
18
19  cvta.to.global.u64      %rd5, %rd2;
20  add.s64         %rd7, %rd5, %rd4;
21  ld.global.u8    %rs1, [%rd7];
22  cvta.to.global.u64      %rd8, %rd1;
23  add.s64         %rd9, %rd8, %rd4;
24  st.global.u8    [%rd9], %rs1;
25
26  ret;
```

- 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

```
__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

```
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

```
myMemCpy3(char*, char*, unsigned long):
        mov     r8d, edx
        test    edx, edx
        jle     .L24
        lea     r9d, [rdx-1]
        cmp     r9d, 30
        jbe     .L20
        mov     ecx, edx
        xor     eax, eax
        shr     ecx, 5
        sal     rcx, 5
.L15:
        vmovdqu ymm0, YMMWORD PTR [rsi+rax]
        vmovdqu YMMWORD PTR [rdi+rax], ymm0
        add     rax, 32
        cmp     rax, rcx
        jne     .L15
        mov     ecx, edx
        and     ecx, -32
        mov     eax, ecx
        cmp     edx, ecx
        je      .L25
        vzeroupper
.L14:
        sub     r9d, ecx
        sub     edx, ecx
        cmp     r9d, 14
        jbe     .L17
        vmovdqu xmm1, XMMWORD PTR [rsi+rcx]
        vmovdqu XMMWORD PTR [rdi+rcx], xmm1
        mov     ecx, edx
        and     ecx, -16
        add     eax, ecx
        cmp     edx, ecx
        je      .L24
.L17:
        cdqe
.L19:
        movzx   edx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], dl
        inc     rax
        cmp     r8d, eax
        jg      .L19
.L24:
        ret
.L20:
        xor     ecx, ecx
        xor     eax, eax
        jmp     .L14
.L25:
        vzeroupper
        ret
```

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

```
std::enable_if<__pstl::execution::v1::is_execution_policy<std::remove_cv<std::remove_referen
        mov     r8, rdi
        cmp     rsi, rdi
        ja      .L37
        mov     rax, rdi
        mov     rdi, rsi
        sub     rax, rsi
        sub     rdi, r8
.L38:
        test    rax, rax
        jns     .L50
        test    rdi, rdi
        mov     r9d, 1
        cmovg   r9, rdi
        cmp     rdi, 31
        jle     .L46
        mov     rsi, r9
        add     rcx, r8
        add     rdx, r8
        Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise
        instructions will #UD.
        More information available in the context menu.
.L41:
        vmovdqu ymm0, YMMWORD PTR [rcx+rax]
        vmovdqu YMMWORD PTR [rdx+rax], ymm0
        add     rax, 32
        cmp     rax, rsi
        jne     .L41
        mov     rax, r9
        and     rax, -32
        and     r9d, 31
        je      .L51
        vzeroupper
.L45:
        movzx   esi, BYTE PTR [rcx+rax]
        mov     BYTE PTR [rdx+rax], sil
        inc     rax
        cmp     rdi, rax
        jg      .L45
.L50:
        ret
.L37:
        mov     rdi, rsi
        mov     rax, r8
        sub     rdi, r8
        sub     rax, rsi
        jmp     .L38
.L46:
        xor     eax, eax
        add     rcx, r8
        add     rdx, r8
        jmp     .L45
.L51:
```

C++17 execution policies can express parallel unsequenced kernels as well

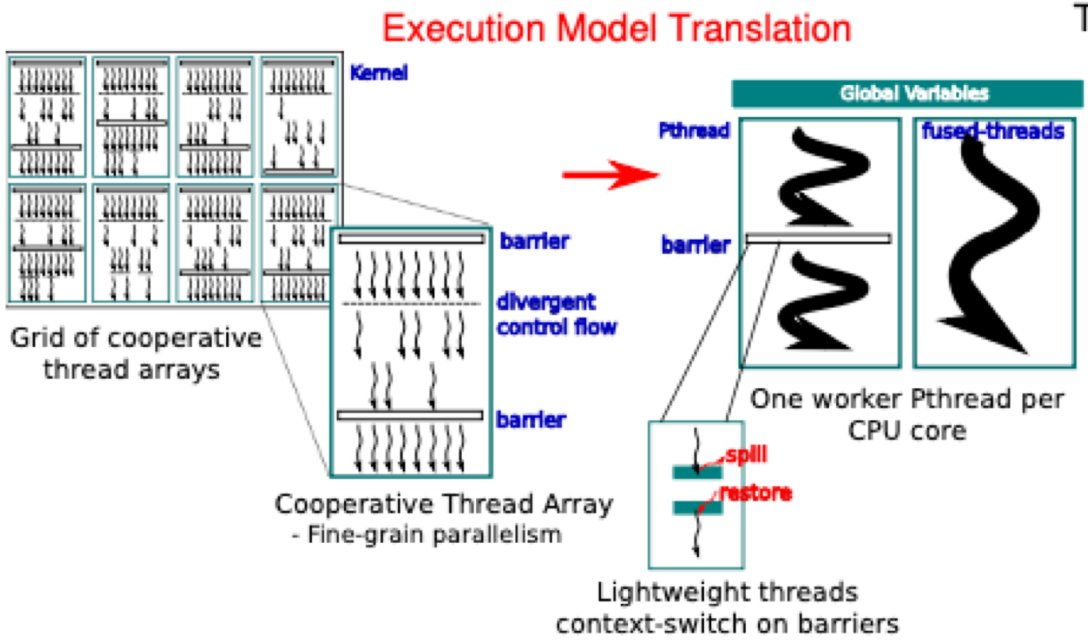
C++17 and RAJA give flexibility through *policies*

- 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



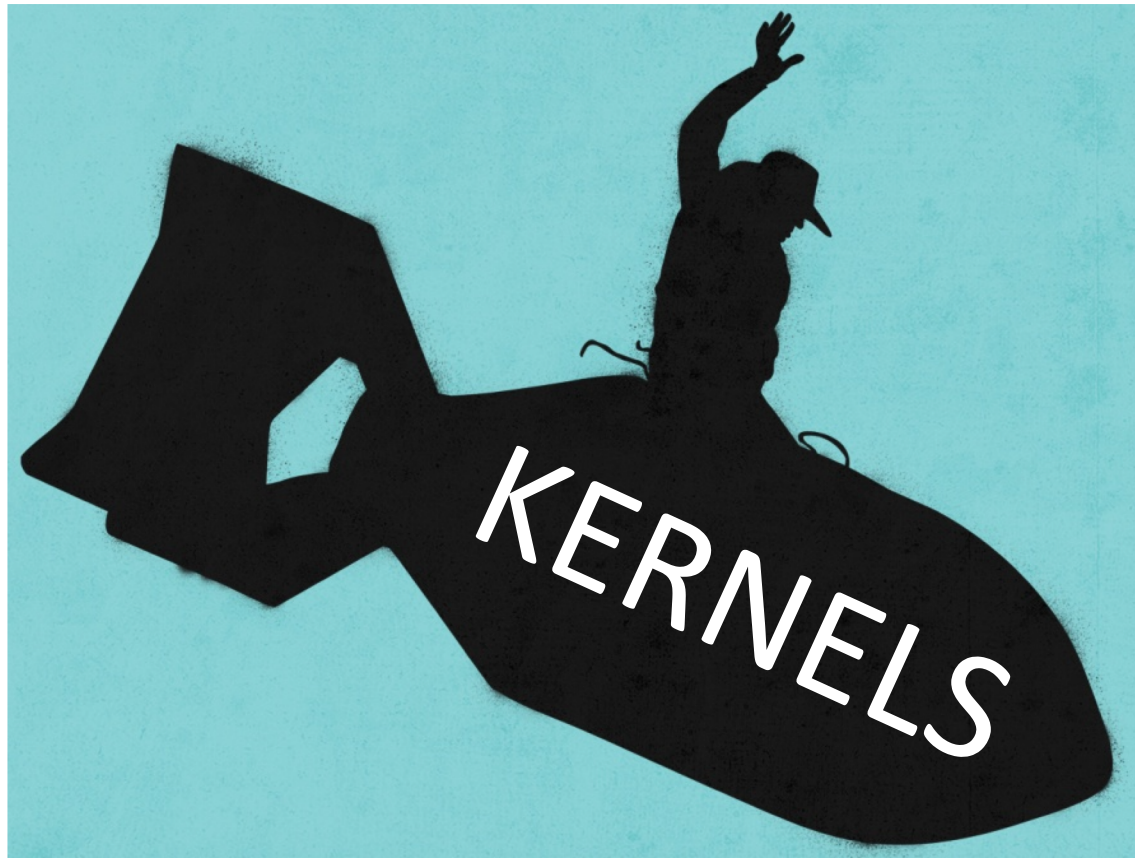
Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems." The Nineteenth International Conference on Parallel Architectures and Compilation Techniques. September 2010.

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



Lawrence Livermore National Laboratory

Disclaimer
This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

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

Godbolt links for examples

- Several host:
  https://godbolt.org/#z:OYLghAFBqd5QCxAYwPYBMCmBRdBLAF1QCcAaPECAM1QDsCBlZAQwBtMQBGAFlICsQAJlKtmtUMgCkggELSZpAM6Z2yAnjqVMtdAGFUrAK4BbWiACpbegAyeWpgByJgEaZilbgGZSAB1SKhBq0%2BkamFn4BQXR2Ds7Gbh7eSiqYasEMBMzEBKEmZpbKqup0mdkEsU6u7p4%2Bilk5eeGFDRX2VQk13gCUSqiGxMgcAOTSXvbIRlgA1JJeu
  vXoWFQAdAhz2JIADACCYxNTmLPzLqgBBAD0xGLAmBd41%2BKYa76%2BG9t7guO0k4Yzc7o2MASIQEMZ3rt9j9DsddJgAB5pQwlWgQnYAN1QeHQ02MAE8ALKYYy6Xx4iDIBDZaYAKmmWHqpGmlOpdMUgyZgQAXpgAPoEaa0bqzSEAdhk00lH0l0wuF2m%2BjJzLoBSQGq0m6WSmjEaYQewCvDHI1bOYSvAAwVe2ayeR4
  YV2mWShkEK0yH3G906wZRmNG/0fSSi5OQzMYrE4/FEklkrwUqmOukRpks0vx5CcvA8/mCkNiiVS3YyuUK1BKtD0NUarXVqM202oc2W63G0NjXzXYDGZia4y%2BHV4YzoUMOp0u6ZuuYe/0%2B%2BZ%2BrwS%2BRB7ghsMRxPDveDg1JlNijN7XaY7G4wnE0l47jF1jgQFYlrS1a1vWApCiKezijKoYdoqeLKr29D9tqupDrGZrXhOL5wV8M7MHOC6oEu0y%2BNkbDsK
  w0wOhuJBbvQO7up6J47r6qJsWesjBpeMo4dGt5eCaEPjaT4wS%2BHzvrmX4Fnier/lW5bMiBblciuEGNtBaYttB7byohyGqqhmroQmuEjoBmF4VKBGzvOi7Lg0twCqqzDGBq%2BD1MQeAuMiRwUdcrDUbRDGBGu9GOs6TG7SrHehxRaBjxF5tmG4YWpGYmxneomCZOXHPqmb45p%2B%2BY/gAbEpoEqZWoG6uBfKQU2MF6ZezDIqOjp3qc5wgCADw3JgEDcs1E
  BbL0QpevBhldkhPYmeqZn3oaJrYVlNmhosA0OrymDMJSo0EOgA0IkiKIDcoACOTLECsbjAPYEC9NM93WC9TJDsJNqjXWzU7he4p8Zl44FbawmrY%2BRU7HB6YXmxOnpiVOwdjJ5XfmSoo1WWWXAQBjWaQDUEfB2ulwbsHZhp1Zo9ZDfX1ANQ1PH9EETVNCNyJT8oGZ23Yqn2K35cOG1g%2BJ3NhgASjsABSOy7SQVEAtLctXau6D7YiUheNgEBkLM1o/carMA7xwM
  SxloPZeDcb5dDp4W1K8P%2BtBZPpsMvSsCAwx6sMpBmMMWx%2B6g3u6NxEqKP0gxHGMgh%2BwQ3tB70EBIGgWC4lQJDkJQND0EwbAcDw/BcCINwoOHKTFMEWg6E0ZheJVVg6JU8SJCAXhbJEgQonX7eN/43fBC31QeB3ldpCiZSNAY%2BR9%2BP6SIK0w%2BdKPnf1OUvcN0oS/tK3NQd70kcDEM3tQr8/zzAgBAEL4iggHKT31CsT0EAghj%2BcogwC/QKxoMY
  FwAHc8AhQANbANYFsUUFxuCCAABxUEENwWBzBRRUCoB3LwzBMCVVFIIFwWwvBeEEEMbgnBzCwPMJwK4zAAEXGYNwSqsD0CimQFsRh3BMDmC4UsKg6BkDIHMAdbBrDRReCWKwzgsDMGVSoVgVUagSAvDeDrVMXwDh/Bjpfa%2Bt974XEfgQZ%2BoJ36GE/otNUv9SKAPAWAkkKCLh4PMFsehSC4GYDgU4pxWBKp6nwVsFwepODoCoMwcwUDri0PQaK
  PxnA9ROMEOgLYPAXAuEqswSRVBMDcACYIQQVAcGVUEKKbxSxYHEIuD2IYvgCC8kpGkEBSi0QfFVEuUQqpYSTGYIoDUAAVNEJ7DAGmFSRQvJHr2DjFgSYBA8S%2BBGjtEAkzWDojYACXpOsXoPUwE9WgL1OZNOJL4VpmjdAdK6dMNZmxdj9PEEMzp%2B0dATLSKwaZszjqnQWU85ZrBVkbA2R9boeyszNMOcwNpAIXnaA8kcC5HxrmDIABKdJkFs%2Bw2AHl3l0Ag
  Op%2BgfiYCqQCO0OwdAEvSpKHFlTIyXzuWM2gTILmkEvOSvFILMV3OsHS94KiuVc1fDsYFRzwUzMhcYaFTJ%2BWguOacxQKwZUbGZKIM5jgoUVgVRqGVKwdjEGAIoNEPlDBqE7Liqp2B4SHQFLpFGaZkwnn2S0iV7TVXnIZaSy2poDIHIdZ06Vsqdbyq9dMJVIrnWw1dZKKV0x1XTE1dq3VBBiD6oFJi7FdAKUAiZca01ahVlMkDZgJk0bvUrE5dgWY5sQ3r3UMgZC9QEQz
  mmH1GiXzDCaJNHgEZciJ6YA1giM1vJ0TZtzfmrVhbOU8qtTNIF7r7WCtmbQKF5y%2BmBBuYixQyLtIor0HQdEcYk3IBAemlIhLiXzEZSm5IJqzUAm3DSJkwzRkotpQunWwawwHovVm%2BY17b1sp0Byrlmxu Uo3FWC%2BYEK50iqfZcnYeqDUyE6THMtM5UDyNVKdO03T6SguYBOkN3S6Q0peqWnlYZiCYAIAMWgAZKroCwzhp2tk8PTH%2BURy8pHyPEEo9IajtGbXpX
  HcVXjvLgPHLA/OmFuxw1GncHgdEXbpggHIsQZDnacRweUD831Frdi%2BHfqwPAyAQB2jhdMNTmANPYAGjSuj0xjOmfM2dY9Y7kaCY%2BDBgUZQdBsDoAh4jO56D4YfYRrTIbJRsYo4KQwIUqnEGs/xkNLo6TMeCyRsj4XaCRdYNF2LyMBNjt2G56YDBVyHJ8x8JDKGu2GfSgl%2BtgWQxlpS%2Bxyj6Wotxuy7ZGrSWGsyjCxxiLrWYuCfo6mZzeXoNxoTZ2FpCJ5OKeU2o
  OTRXpsse07p/TVWQ3GaWyVyzD7rNbeK%2BwBz64XOjZRgVgAag8cjbBTMrZDTplwemDN2nRNdwwbA/MEAC9soLkMEqvfe59zrOg/smgB7l87E2DVXZyB91gUnfKyZxApx7z3piw5u6wO7yXyJrZe%2BlN7cPgf%2Bdq794UqBZPEF8v8broXUt9Za5ltrQ3S22SJ1jr7iXQcU6pzT0rLrevNYy1l1ncXx0uYlrQEg84aIdoXo%2BmkHlmBcgGawPE%2BGlMgOOK5rllbeRevcA
  KCAK612ouPboDzNHWDeY2ANJtea7TSEENgKneJX4DJ3BqKg7Rnecx0vlvX%2BmDddKN06MAYBTcPo3QCOzn76DFvt2wZtL6AyCGE3NirOIXDwb9zhitwfDc5HD5HpF0eLfbfYHbkADvU/O%2Bl7QAAtBn8rKn6255yYC3IBfkAh8/sbqP66LeI5k12q9Cf/1J6MI7vjOSM9YCR3JqgSnjCZ7bzn9Tnf89B970XgfZeh8nd0Poab8Jq%2B16dzkhvzep1tIX6PnEy/SJr4W9njvggu
  %2B69BYX0PxeTcH/NyPOx3hzjwsxr2TxnxDWdw53h3b03w/232/131/331XXLyAKBwR2k2R3PwgLrxyRgM%2B3v2R01D52xDMy30l1RnlAAHUSAQFsh%2Bhf1VI6laQIU6QBxa1SMulgg3pQUsVHRX4xBpgvlsRCA8Qv99c98nRB9ACcV%2B15grcvMHBcDp98Cxc3cPcbk21aJfdKCHZu8d8%2B8w8IAI9ZDaAN15DY9c949KUAMwCL9Z909b9AolMs84CKCEDBMe9jC
  /8zCACLDa4t0ARK8zMdYp8U9L9Br8W83D1939P9A8kDfDUCzdAjN1aAFDdAR8cDbDE9wC1Coj59sCl8V8X9UMPC89vCjDpD/8DDD8rD5gT8Ssz9wiCjIinCYiXDzRF9H8yjW9X9Kj9CodkjajzDLDgj5hgDbsbCHC8CojCCaIN9PCu8fCxiAiJjMiARpisDejVCOioCCDMCeiH9SD3B%2BcqixtLVdgPYRBvZfZ/ZE5g5Q5w4dQo4hgAwvBOB44niAVSAQF25uAVg9RBBHFRE2E
  pFzAYE9ReAvZhhuA/ZjAuAthO4A4g5SAQ5hg/Y75O4E5A5bi4BYBU5SJfBgF3Bs4KQSSySPBgBOAthhAfdnl3A74IAXAnjSAXB7Bsg8RvZvjSA/4RV6AAB5WgdXdkrAeccQI7fE0gfAUjdIWTO%2BGU86ZAAKXkv2F0FQdkvTFwa4YgPEfQLAdU0gONVcdU3oXORgFgI7IuAQTgUucQcuFKBQHUu%2BWAcDDgBmICKnXk0RAKj0kAR4W4RQdEHwH0wOOkpOXoLsFEJUxv
  ecQYdYYSYZABFQGiRvIUwQaYRvRYPcKQZ0zgUUbMmgWdJcbE1IBXGuTdMIMwe06wZeNue0geaEIEGcIJsqIFEBsmoe0ooCeDIVoXuXsisyeHeOIEeEudeaeGsycscjoRsw%2Bd4jgD/O4n2P2NE544YeEWBSqRvSqbgaYYAfhaYOklYLMiADOIgHqL4YQKbUk9ga8r44UMOZ0n4/E5OJAfoG%2BZECkv%2Be8nsqwfAK8oQUgS0/OI7SwABa4Xwc01ch4jcjE0%2BL4TgaYI
  BV%2BaYbc3c/cw84808uOE0343oLFZgBfSgXoAEohFYTgLwUUPUcwLwOi0pXBGBVchE0gJEyMx4mUzE7EkAXE340gQkxAFAKgYACkrtTOGnECu0jEqpYIFkudN7YAb/TQYYPk3wBAPEQIZAJU743oAeSgXkIy%2BEPi0gIy3kHkxvL44Ew%2Ba0wueeVDAAMQDjzExl/CUhvVUmyE8sMFoECGAAcBxBt3EBDANEmG8wjV5BJN5CoFoBWE9GNG9k9nuPXPZMxMwr3IPK
  PKrTwumAAHFoKEAMc8BMBUzopLyGJY4mRmjqTPjOBug3yoz/j24vAVhCF2qOrOrWLUruLvZeL%2BL3zkrhh8KEKeKCLBrSAqdAhNBuAgA%3D%3D%3D

- Single host:
  https://godbolt.org/#z:OYLghAFBqd5QCxAYwPYBMCmBRdBLAF1QCcAaPECAM1QDsCBlZAQwBtMQBGAFlICsQAJlKtmtUMgCkggELSZpAM6Z2yAnjqVMtdAGFUrAK4BbWiADMpbegAyeWpgByJgEaZiIAOykADqkWEGrT6RqYWvv6BdHYOzsZuHt7Kqup0DATMxAQhJmaWyZhqQemZBDFOru5eShlZOWH5tWX2FfFVngCUSqiGxMgcAOTS5vbIRlgA1JLmuooE6
  FhUAHQlO9iSAAwAggBuqHjoE5gAHsaYxsg%2BAJ4QyAiZEwBUAPrPxJhzxHhqE1hzpBM7g8Xm8PgQvj9FH0AQEAF6YZ4ECa0DpTbaSTwyCbYzZbbETAD0BIm%2BmugLoBG0BEUE1QVAmUKkAFY5EyACITIi/MGSFm8jm4/E0YgTCD2JF4aZsjbTLGSma0WUTSWyeSowX4/F/Ai8mSS9lTcwcxm6/VspW4jEcgZdVggAZMgakMwDDZO1D23TyeQMnp9TBTQTmQROgj
  2t1dCBINBYXCEEjkSg0ehMNgcHj8LgiMQSb2yJQqQqpMwQaz1PIANisONKcQSFg2EQCxfLFirfmbQVrlQ85kbBSKaSarfMVYHxZKWW7bV7/eHBlybZqpWn9b7XUUfv69uGo3GAemugQBAIPkUICJwDwcyWV4ICEMLkMyj6FKpSzQxgJAHc8KxWAA1n%2BrAbJ4BLcIIAAcVCCNwkHMJ4VBUH25jMJgFaeIIlgbOYwb9NwnAAJyQYRnAEsQzDfgSzDcBWkHoJ4yAbHR3C
  YIR7ELFQ6DIMghGYGhFZMZ45gLExnCQahFZkVglJqCQKw%2BD4ayWkGe6GJMh7Hqe56XteBC3oQD5Pi%2BaD0O%2Bn4/sBQH/qBBJYYRGw0XBUGYFBjmOVgFZMthGwuEynDoFQzCEWBFFUchni%2BZwTKOYI6AbDwLguBWzDiVQmDcP5giCFQGEVoInheQskGCMgBKmf0PgEM8dyFABClKeY6zotsILGD4oiUoauhjMwig0gAKsp2zPvYwATPcijPG4V6OlaHJYGM
  BBXD4mAQHM6AgCAi2sDsbCHkNTUQB0SwzfYx0dLKuJtR1zBdYevX9RMh3NVso3iBNfXPNY83cktK1rRtW07XtrAHWsx1LNYF1XS1Ww3Z1B4zMtq20MwZzPcNb0BB9AASfUyJgs3YDov1HnV%2BiOJVOozBqWw6leGrYpT1OaV9Z20ACL2kEzJJ0KzMyTd9Ohc8pTVi3IcMI3dmCHij2jowGA0AtL90zI9IhLFrayAqIT2OIrAIaxMWtLFsxDAIoWOfIYPws5gVXYMczA/Bikt
  bJanjmuY7uq0jPV64NPPbJqoecuct1qwHfWa9rTW6zHEwG2cwd4mH%2BLG6bEzm5b1vgrbSLk8gAH21Vh6lwQTsuzTujK0nhvZxbsdLGL2BTJiGpzHd3zkrQcwnD4IouKgBgTKDhhIxy17PDJRaYOg33O2ozw7AdpDJ5gAI583Eue977vXRHiPdfLaMYy9uLveN%2BKITxM6JTOXwkgFP8w7Nd0wztMh/iFdV2oh5xRPABELDmosmqp01H/JeNcgGPBAV9aw4D1ji1QQfV
  qR8ZYnwBmfJWediAFwmDIPqB4O7bEHqgWSlJNoagGr8O6zBYZpwGo8CYHNjrt3dpqd4BBehzWkBWdADCmE4i9rQ1h0N1RkLTtiHhfDAyCOET7DUVo95MMPu1Y%2BcscGK0xuLbYxs2TuDwDseeEwQATAoVQsxxDIDg3jm7XEPhHysG%2BCADUV8iEkPsdgLaHMRETE8bY2WMwXpbWsOosRyi4Y2x%2BCUHQbA6CkK4cqegrD2FSJSbIzAvDiBzVOIYf8VViABNUT/
  cUEidAcMcT/bJuT8mFNYMUQ0pYj0RRPQVsWJSIGB4EOckpxxBKFzxoeUtJbCibnUybzQReTkSNOadEtOZS04VKOFUqZtSJgzIaUU8ELTzRtP3pabYXS%2BZ9OOOYyxgzrGHB6X0zhTiXFuI8TjcadyOocBAP4xZ2JPHvPYOEr%2BB92nHM6fnH4AA1PAWRDBsGCQ8n%2BziXCuOQO4n%2BOxoW8LYKkgg6SJm0GqUaCYMofIj0xbC1gOLKnoEJRyElwKjxKP2BGK%
  2BKYw4FikUoomKyrFrB4U1LTry556LyXYtWRk2kpjiBfA0tI7hOT5EFN2SU0lyzsQYrZZS1ZkjZXuAVf0zZ2y5nqv2WoxZuliTlhIMYbFs9BycyeOjZgsIxqsCuOkwZAftC4i7uoZAzwY7uCRBAG%2Bd97Ak3QIeeJQjWBJLWFtcem8VHZWwHKq494xrKhpFQFo0hBCXR9miD2JyMhBpDf1MNoowBgEjfimNh5gmAPoK3FNbAJ6QMDIIP2VyhlqDMS4bx2US3u0Dd8atL5w31sb
  ffWNMx/khN8SAVNPai20DoAAWn7VY4ZbDR3FvURW7uwbQ1ZFFPO6NX9dAcpMfPNtNMUGrvXemvtmCupYE5WYqggzjADpuYeuxY6mGTvPTWy9EaCZNtvfoc5ya11drTT/TdO7%2B3fofYcP9qAAP7qHYcEdIHj3WtPVWi94br20GbTMAVFLW1NU7UYFDSzso6sFcB2WoHFngenbW6Dt9YOLt0HR9lxiuWIbfahtjkrKWYa5Ya%2BVBwuMkY6bagA6iQACmQegi0B
  C/YuTx5asLpEcY4g8PgBDoFsu6L8RT3jEGPNgBxCBXADZWqdFGr0wYXY/ONGQEmJocJJ5DG6M1ZpzR9a8EwC3Be41iUFvGvMQDnT5m9eg6CrxmAx3Q4oO1IeY2Fj9misH4eoZxot473Nnr41B1LgnfOZbjb0j5IXCvvq3bQXdn6AxleHUeqrTKavJaozGvzMx70SZmHl1Br7Qvvow%2BJ39/7AMHqIypwbYLhuQco2l6jD8mszHgx844bXu0dfQz17kP7sMrb64RgbJ6tvkZ29
  5hr6Xxsidkwxub7XpOCHYxSir8XqsvZnW9qN%2B2Mu0Cy193Vk3H2MYK%2Bd/7gPHVLcOKgOVxrKtPdBbiG0Ih7SOmdOGd0no8xYk3L0fogZzCcFDGTjoXQAlgBiksJkIVIIVmlhhJkfZaJE4GNwJ0xguAbEbC6N0pAPDQGdOeRsYZXSE7gLAGAiAUC4Z8H%2BdwiZbha51x4YAnANjCALawSkxBzwQBcGT0gLh7CZCuPaBnpBPxnHoAAeVoF6u3WAHXiABcr0g%2BB3hFF
  MeeYPJxCiGEpC7p04oVB29cS4CixArj6CwPH0g4Jenx66MmRgLAAUZgEMIUQ4gUCU5EHgFw55IBdFQFVIlkft0Or6Ksl0k1vwqEpduz3IYCwpCCFoA7oQzCCEbNYVcVRLAdiiMEBcYRJ9NgXzP3sQ%2BizFHnOPoQ/ZCzOsnM0WIPZwhdzqEvifc4VwtDrLPjcW4ODFqFyTqX5OBjHG59uis3AJjAB4hMCbksIIKKHGEQCKMMMMIGctruwBAUGMWiSJTozsrszqQKzrhEsLhFg
  dgTgULiLqOGLuzp4JBm3FNFgMqsW6SPM2wAEmzRKYML2CaMuWjLKyOhzZgJNjputyaEXypdnnxMGtqEJ21trBrLH2CkJu5KVwX1fZwLpiNzTcf93WIeAobQ1jpHtDR6Mayh0b4cs6I7PnEvk3zAvHPWNH8Jr0MS/nLcxDAHIXO3HUxCSMoQztyT2DHBfHa07uG768PYfM6bHMjC35ddj/Eov7Pb8167xXu/IYG8ARf3rL/LsoDd5DH3ofY%2Bp9z7YmzhTK%2BncME32YMo
  FWaxwGrHgQfOQwD%2B7YCTJ2FcyQVDIHmHQOoRw3BzDsKgXQnw2CSHDgyQy9xBosUwOge2t9SYfDwQQtYCCSHLwHo6Akzon7B2fgzABY9JYTwBObS2twE582TnIVO6d0J0A%2BFnHOLIMfI3rCXdAIif5YOEWOfsUI1i10FmnSwvcLaQLsXQKu2DcHaicS47umhPFTTIUHIeyjjYRzUZPJupJkEXyJGzRayY7hMNQENZQAiQCbhGEMGQyQg7khGPmeu6sKGtyGNKAg
  XQRaliwkU5gJStz5PzAHds6SoE1MwHU4gDTBBNKNi0hR5Jdw7jkZEpRI857j3icfUBdJHB4A%2BAIoYIAhiZOyUNXYKSFC%2BwtIUjIZlWB4GQHk8uOihj7MwIc7Act66tOubc%2B5ctTCRKmQoxJBAz4XwKJoNgdA0mUMpOU9xtBKn32/r0/pItaCSFciTYgrSJn1jBQcTQULuk/1qfUnEiLWDItRVMyZ8jB53CSdiBgQozggtJNs/hstNwYoqV0yh5I8UDIJUi35JLtzovoB0rF7Lqlc
  vhYS4lYy2lkumSfX5KDL5HTpTyTZjKcm7BpQzelJymTnMufWa5Wq6WPIhc862fNjWQRAB875XyKXyr%2BdiAAangVlkg2C3OOfWNwpz9Wbg%2BG6upbAKRCsZBCqF9lhjDmIYG91IaMWdOXDGwe9raZUqGK6%2BNrBlnEFWestVfraFzuDawL10KfVFoNeSONpbQ0EHBcLSpqA1nEDzepHFXZxU8qJXy6VaKa1Bo9awetwr0DNtbe2nVPTu0lt5Si/tpLg5pp3HcQYOIS
  D6BDXwniDdoQnGYGiXmrAxjlOIKgFkGgfIG3OfcNmlIsQQD0QYywGcbHagBTILYVhJpy29pw7xopBDYFbWMSUvMKQcyoI0Usghda1nxKu%2BVN7kB3vZo4R9YAwDPohW%2BscrzAn0D7g8kA/7ViblgzPJm6rdmMlSbB%2BDdYRI0LQw%2B9UWGcOGPfVoK1dyLZ/o4eR8usHaB0AALRUamDR9ZrwDmo0Y%2Bm6ht770YfVJx19pdt5SvzdxykxGBN6CE/WSj/8mZY
  DzWs3YVBz36C2eenZMn6PyaDsx5T6HUhqZTrhzTWgdDaprCRsjgGRPick4UlZlmhjWaOnZrJTK6Nybg3K1zqGVMeafV5rjY4S3DoI4FwTwXUa1uHQlu5zmFEpdY6pjL%2BjvPcZy2wbTlnf2kYKxRorQ7t0RfWS2xwU6GNyvXQAdRICyFInRQmRgptCSO%2BZUBqkYdKdmPgalG2WHMSU%2BxvY/EIGMa9LG0uPvU7QN9edREfuSIC79fH8uGcK8B0D4GbaGSizB8r4pFM
  oaq%2BljjmWNPaGMfh%2BjhHq7hNuwB9rggwvSd2JsrSWKtKdS%2B5zD2G/snaMbQc7PHaXsBa0FyHonaASdM1J%2Bz8W4cDe%2BZSpH32jvo9O4DrT3XdNEbIQZiHwnUZhfMzpqLNnYsOdh05hHTHaeHc87VrjZ2xx%2BZVfjtrXPBBE5J9EqYvPIvRdszDOrVOHWVYlzVI9GOAdY%2By51stwPwdGfGR17Nuv3v7bc2xo3dWZfagazmlnCu7uQ%2BK11izPXJ0/Hhwp
  pDD87j7hEEeE8Z4vxXhvlhMsT55KLRmpwD88fWjtBZCAZG5wCSF%2BcCjbg2NrheQJNH3o3BTz6C4OXuPF4E%2B9FPLpEin4m8/hgIgFAyqXKOHIJQeybh%2B9OCx01Tg1xhDQedo4X5EA7Dx9IHYSwKQxhHgz6QeyAJ6AAHlaAnqX1gLd4hJN9IPgaUeQ1m6XPOcXlkghob9PJSRQz%2BRB4DsLlYgYwdBYHf78kKM/u0DQPQEwGwBwDwPwMYHBBINILIGcnYLpJAO0
  IGAkLfmJrvs4K3jkLumhADv4EYEMrhOYI0ARC0NRGRAkJRMQTRAkFJIRJwDhHECxPkPUDQcwbgaxPUAwRQRJOxIQUIMwTwWQYJEwe0Cnt0BwHBlXrHueJeKQNeL0NYM6r5uSBALgIQBsEpMIEqqcKPq%2BJwHiHeA%2BJnp3qQL%2BCAJ0Npl/oPmCH3uwNECYIBCQEIKQKAeBBASAGIKQI5LIG4MAbIaePIc3kpJwEMM5JKGAqoTqGYd%2BO0MsDsGUOrKQLnoa
  NwOcKjEMsFASlaDNDVEFFXjXqQHXpPiRCEYoUeK3iAO3IngeEeO%2BI3goUoXEfuO0K2l4GoNwEAA%3D%3D%3D

- Cuda:
  https://godbolt.org/#z:OYLghAFBqd5QCxAYwPYBMCmBRdBLAF1QCcAaPECAM1QDsCBlZAQwBtMQBGAFlICsQAJlKtmtUMgCu6ZqQDOmdsgJ46lTLXQBhVK0kBbWl1Ib0AGTy1MAOQMAjTMS4BmUgAdUcwqto69h4w8vFToLK1t9BydOVwUlENoGAmZiAj8DI055RUxlHySUgjCbe0cXeWTU9ICsuUqiyxLIspiASnlUSWJkDgByAHp%2BgGoAFQBPN0whsc7iIbQsI
  YRHTFIhkiHWVGZOIbEhzAAPZn03dgA6AFIABgBBAH174C27NkehgDdUPF39Mf1MPpkG4xhBkAgUkMAFRYOprcGQqFybprLwAL0w9wIQ1orSGlwA7AAhG63IZDSzYlS7S7OAAiQzsW2QAGsAJLoQ7nQ7QxnMll0vD6bn4wREoYEBDETA7Dlcw60kl3cl4KhDCDU/HOLQ4vGwgiXACsROpRoZtlZyOQRpNPzNitJhLjprujzeqFerHeXx%2BQwBQNQIMEYIhcxhcPmoehVt
  ReAxWN1%2BOJpJV9AlvotfNQrLIiqhWdZguFPMuYolUpl6FzCucSrJFLVGoz9Mz1zxhLr5PJNDmEEpFK1DOuioHtJ1tBHpZJYrwergNrwZsHQytC/ttZTSeddydpN67VYIF6ht6pCMvWup9QR60AFU6WS5LMeqLnIJTwQj5f2hAkAscPgRBkBQ1B0IwLDsFwvACFkojiCgU5Ttk8Q%2BOomjVEYzgAGwmJoxQRFEIDONc7ieN4dAYUROFBORtD4aUTjEchuQJAUV5BkVHMX
  kdBsQ04QMURJF1IUlHYRUhT0c0jFth0XQ9EepbOJYyB6lsY4IAQBBuHIICDMAeB1OcBmSpIdiSAo3RgRoBDnGg%2Bj9AA7ngrCsCyLmsNcBL9NwggABxUII3B%2BcwBJUFQxHOMwmBYQSgh2Nczhvj03CcAAnH5aWcP0xDMI5/TMNnWF%2BegBLlNcxXcJgaU1eg6BUOgyDIGlMoxeVBLOHV5WcH5UVYdlWAECxJDnAgbhuLS2COoIym0KpOhTBpWk6Xp/QGUZJkI
  GZFmOGg9A2XZqAOc5rnua5Xn9PFaXXIVwX%2BZg/k3TdWBYYaCXXHYhqcPVzBpd5uX5RFBlfZwho3YI6DXDwdh2FhzA9VQmDcF9giCFQsVYYIBKvXVfmCMg/R7T0bgEPc4K5Cyo3jZNLq3ENpyiENWpaKpzByHIow03cFmWMASxs/cDgGbQy5YKpBATJgEB1OgIAgGLrAfGwY4jJNECtOcQuWOrrQOnc9NnMwTNjqz7Oc84U3c144j83I9ymKLuSsBLkzSwQsvy07SusC
  rasa6YOt67cpIG4zi3ai7GgnFMqsW6SPM2wAEmzRKYML2CaMuWjLKyOhzZgJNjputyaEXypdnnxMGtqEJ21trBrLH2CkJu5KVwX1fZwLpiNzTcf93WIeAobQ1jpHtDR6Mayh0b4cs6I7PnEvk3zAvHPWNH8Jr0MS/nLcxDAHIXO3HUxCSMoQztyT2DHBfHa07uG768PYfM6bHMjC35ddj/Eov7Pb8167xXu/IYG8ARf3rL/LsoDd5DH3ofY%2Bp9z7YmzhTK%2BncME32YMo
  FWaxwGrHgQfOQwD%2B7YCTJ2FcyQVDIHmHQOoRw3BzDsKgXQnw2CSHDgyQy9xBosUwOge2t9SYfDwQQtYCCSHLwHo6Akzon7B2fgzABY9JYTwBObS2twE582TnIVO6d0J0A%2BFnHOLIMfI3rCXdAIif5YOEWOfsUI1i10FmnSwvcLaQLsXQKu2DcHaicS47umhPFTTIUHIeyjjYRzUZPJupJkEXyJGzRayY7hMNQENZQAiQCbhGEMGQyQg7khGPmeu6sKGtyGNKAg
  XQRaliwkU5gJStz5PzAHds6SoE1MwHU4gDTBBNKNi0hR5Jdw7jkZEpRI857j3icfUBdJHB4A%2BAIoYIAhiZOyUNXYKSFC%2BwtIUjIZlWB4GQHk8uOihj7MwIc7Act66tOubc%2B5ctTCRKmQoxJBAz4XwKJoNgdA0mUMpOU9xtBKn32/r0/pItaCSFciTYgrSJn1jBQcTQULuk/1qfUnEiLWDItRVMyZ8jB53CSdiBgQozggtJNs/hstNwYoqV0yh5I8UDIJUi35JLtzovoB0rF7Lqlc
  vhYS4lYy2lkumSfX5KDL5HTpTyTZjKcm7BpQzelJymTnMufWa5Wq6WPIhc862fNjWQRAB875XyKXyr%2BdiAAangVlkg2C3OOfWNwpz9Wbg%2BG6upbAKRCsZBCqF9lhjDmIYG91IaMWdOXDGwe9raZUqGK6%2BNrBlnEFWestVfraFzuDawL10KfVFoNeSONpbQ0EHBcLSpqA1nEDzepHFXZxU8qJXy6VaKa1Bo9awetwr0DNtbe2nVPTu0lt5Si/tpLg5pp3HcQYOIS
  D6BDXwniDdoQnGYGiXmrAxjlOIKgFkGgfIG3OfcNmlIsQQD0QYywGcbHagBTILYVhJpy29pw7xopBDYFbWMSUvMKQcyoI0Usghda1nxKu%2BVN7kB3vZo4R9YAwDPohW%2BscrzAn0D7g8kA/7ViblgzPJm6rdmMlSbB%2BDdYRI0LQw%2B9UWGcOGPfVoK1dyLZ/o4eR8usHaB0AALRUamDR9ZrwDmo0Y%2Bm6ht770YfVJx19pdt5SvzdxykxGBN6CE/WSj/8mZY
  DzWs3YVBz36C2eenZMn6PyaDsx5T6HUhqZTrhzTWgdDaprCRsjgGRPick4UlZlmhjWaOnZrJTK6Nybg3K1zqGVMeafV5rjY4S3DoI4FwTwXUa1uHQlu5zmFEpdY6pjL%2BjvPcZy2wbTlnf2kYKxRorQ7t0RfWS2xwU6GNyvXQAdRICyFInRQmRgptCSO%2BZUBqkYdKdmPgalG2WHMSU%2BxvY/EIGMa9LG0uPvU7QN9edREfuSIC79fH8uGcK8B0D4GbaGSizB8r4pFM
  oaq%2BljjmWNPaGMfh%2BjhHq7hNuwB9rggwvSd2JsrSWKtKdS%2B5zD2G/snaMbQc7PHaXsBa0FyHonaASdM1J%2Bz8W4cDe%2BZSpH32jvo9O4DrT3XdNEbIQZiHwnUZhfMzpqLNnYsOdh05hHTHaeHc87VrjZ2xx%2BZVfjtrXPBBE5J9EqYvPIvRdszDOrVOHWVYlzVI9GOAdY%2By51stwPwdGfGR17Nuv3v7bc2xo3dWZfagazmlnCu7uQ%2BK11izPXJ0/Hhwp
  pDD87j7hEEeE8Z4vxXhvlhMsT55KLRmpwD88fWjtBZCAZG5wCSF%2BcCjbg2NrheQJNH3o3BTz6C4OXuPF4E%2B9FPLpEin4m8/hgIgFAyqXKOHIJQeybh%2B9OCx01Tg1xhDQedo4X5EA7Dx9IHYSwKQxhHgz6QeyAJ6AAHlaAnqX1gLd4hJN9IPgaUeQ1m6XPOcXlkghob9PJSRQz%2BRB4DsLlYgYwdBYHf78kKM/u0DQPQEwGwBwDwPwMYHBBINILIGcnYLpJAO0
  IGAkLfmJrvs4K3jkLumhADv4EYEMrhOYI0ARC0NRGRAkJRMQTRAkFJIRJwDhHECxPkPUDQcwbgaxPUAwRQRJOxIQUIMwTwWQYJEwe0Cnt0BwHBlXrHueJeKQNeL0NYM6r5uSBALgIQBsEpMIEqqcKPq%2BJwHiHeA%2BJnp3qQL%2BCAJ0Npl/oPmCH3uwNECYIBCQEIKQKAeBBASAGIKQI5LIG4MAbIaePIc3kpJwEMM5JKGAqoTqGYd%2BO0MsDsGUOrKQLnoa
  NwOcKjEMsFASlaDNDVEFFXjXqQHXpPiRCEYoUeK3iAO3IngeEeO%2BI3goUoXEfuO0K2l4GoNwEAA%3D%3D%3D

Lawrence Livermore
National Laboratory