S4178: Killer-app Fundamentals: Massively-parallel data structures, Performance to 13 PF/s, Portability, Transparency, and more

Rob Farber
Chief Scientist, BlackDog Endeavors, LLC

Contact info at blackdogendeavors.com for consulting, teaching, writing, and other inquiries
int main()
{
    cout << "Hello World" << endl;

    // load data and initialize parameters
    init();

    #pragma acc data \
    copyin(param[0:N_PARAM-1]) \ 
    pcopyin(example[0:nExamples*EXAMPLE_SIZE-1])
    {
        optimize( objFunc ); // the optimizer calls the objective function
    }

    return 0;
}

double objFunc(...)
{
    double err=0.;

    #pragma acc parallel loop reduction(+ : err)
    #pragma omp parallel for reduction(+ : err)
    {
        err = 0.;
        for(int i=0; i < nExamples; i++) {
            // transform
            float d=myFunc(i, param, example, nExamples, NULL);
            //reduce
            err += d*d;
        }
    }

    return sqrt(err);
}
13 PF/s average sustained flops using 16,384 GPUs (On both linear and nonlinear problems using MPI and thousands of GPUs)

EffectiveRate = \frac{TotalOpCount}{T_{broadcast} + T_{objectivefunc} + T_{reduce}}

Note: Always report “Honest Flops”

Acknowledgements:
Oak Ridge Nat. Lab and Fernanda Foertter (PI)
This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
Scalability required to use all those cores (strong scaling execution model)

- Threads can only communicate within a thread block
  - (yes, there are atomic ops)
- Fast hardware scheduling
  - Blks run when dependencies resolved
OpenACC portability exploits strong scaling execution

C

/* matrix-acc.c */
int main()
{
    ...

    // Compute matrix multiplication.
    #pragma acc kernels copyin(a,b) copy(c)
    for (i = 0; i < SIZE; ++i) {
        for (j = 0; j < SIZE; ++j) {
            for (k = 0; k < SIZE; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return 0;
}

Fortran

! program example1
...
!$acc data copyin(a,b) copy(c)
!$acc kernels loop
! Compute matrix multiplication.
!$acc end data
end program example1

C++

int main()
{
    cout << "Hello World" << endl;

    // load data and initialize parameters
    init();

    //pragma acc data \
    copyin(param[0:N_PARAM-1]) \
    pcopyin(example[0:nExamples*EXAMPLE_SIZE-1])
    {
        optimize( objFunc ); // the optimizer calls the objective function
    }
    return 0;
}

Multi-core, coprocessor and GPU versions exist
How to work with complex data structures like graphs?

• Build a parallel stack for complex data structures
  – Must be performance robust (e.g. low-wait)
  – Allow transparent host <-> device <-> host access
  – For performance, must manage memory!

• A stack is required for dynamic parallelism
  – Dynamic parallelism == variable output!
  – Following is a fast intro!
Outline to create a parallel stack

1. A fast, robust ParallelCounter class
   - Counts items in stack ... cannot limit parallelism
     • Even in pathological cases where every thread increments at the same time

2. Transparent data movement between host and devices

3. Combine 1&2 to act as a stack that provides:
   - Fast object allocation
   - Transparent access to data on both host and device
   - Can handle arbitrary numbers of inserts efficiently
Atomics are great (but don’t use them)
Much faster when there is no contention
Minimize the wait by partitioning
(threadIdx.x % WARP_SIZE)

Sum the memory locations for counter value
Pretty simple in C++

  - Note the count array in red

- GPU: N_ATOMICS=32 means zero wait

```cpp
inline __device__ uint32_t operator+=(uint32_t x) {
    return atomicAdd(count + (threadIdx.x % N_ATOMICS), x);
}
```

- Works great on multicore!
  - OpenMP, Intel Xeon Phi

```cpp
inline __device__ uint32_t operator+=(uint32_t x) {
    return atomicAdd(count + (random() % N_ATOMICS), x);
}
```
ParallelCounter performance is great
(lower is better)
Outline to create a parallel stack

1. A fast, robust ParallelCounter class
   – Counts items in stack ... cannot limit parallelism
     • even in pathological cases where every thread increments at the same time

2. Transparent data movement between host and devices

3. Combine 1&2 to act as a stack that provides:
   – Fast object allocation
   – Transparent access to data on both host and device
   – Can handle arbitrary numbers of inserts efficiently
Host/Device(s) object layout compatibility

• Needed for cudaMemcpy() and Intel Xeon Phi data transfers

• Not mainstream!
  – Yet

• Happily addressed by the C++ standards committee for binary read/write
  – File I/O
  – Socket I/O
Ensure host/device object compatibility!

• If your compiler supports the C++11 standard, use the standard!

• Unfortunately NVCC – like most compilers - is not C++11 compliant!
  – Means the programmer must use compiler front end calls
    • Not ideal but it works
  – This is not extremely as the calls are used only for a safety check and not functional reasons

• GNU
  – __is_pod()
  – __is_standard_layout()
  – __has_trivial_copy()

• Microsoft users
  – is_pod()
  – is_standard_layout()
  – has_trivial_copy()
Do the data movement

• Pre-6.0
  – [CUDA, Supercomputing for the Masses: Part 28](#)
    A Massively Parallel Stack for Data Allocation
  – [CUDA, Supercomputing for the Masses: Part 27](#)
    A Robust Histogram for Massive Parallelism
  – [CUDA, Supercomputing for the Masses: Part 26](#)
    CUDA: Unifying Host/Device Interactions with a Single C++ Macro

• CUDA 6.0: TBD
Outline to create a parallel stack

1. A fast, robust ParallelCounter class
   – Counts items in stack ... cannot limit parallelism
     • even in pathological cases where every thread increments at the same time

2. Transparent data movement between host and devices

3. Combine 1&2 to act as a stack that provides:
   – Fast object allocation
   – Transparent access to data on both host and device
   – Can handle arbitrary numbers of inserts efficiently
Allocation times on a GPU

• The CUDA allocator (cudaMalloc) is very heavyweight because it must manage all device memory
  – It’s only going to get slower

• The solution
  – Allocate many objects of a single type at one time
Results of testMalloc.cu

- Comparing individual malloc vs one big alloc
  - 7.86 vs. 0.00001766 seconds

$ nvprof ./testMalloc 1000000 128
====== NVPROF is profiling testMalloc...
====== Command: testMalloc 1000000 128
Using: nElements 1000000 128
====== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00</td>
<td>7.86s</td>
<td>1</td>
<td>7.86s</td>
<td>7.86s</td>
<td>7.86s</td>
<td>performIndividualAlloc(int, int)</td>
</tr>
<tr>
<td>0.00</td>
<td>17.66us</td>
<td>1</td>
<td>17.66us</td>
<td>17.66us</td>
<td>17.66us</td>
<td>performBlockAlloc(int, int)</td>
</tr>
</tbody>
</table>
Modify ParallelCounter

• Need unique counts that are never duplicated
  – ParallelCounter can potentially increase by warpsize in one instruction!
**BoundedParallelCounter**

1. Index off individual counters in the count array
2. Calculate the index so there is no overlap between counters: [0,N) [N,2N) [2N,3N) {where N = maxColCount}

3. Each index represents a unique stack structure/class location
   - Find unused data structures quickly and in parallel even when all counters increment at the same time!
Use transparent object movement

- Transparent object movement allows
  - Host and device allocation
  - Host and device traversal

- Create complex structures on the host
  - Compute with them on the device
  - Read variable results on the host 😊
A GPU is actually many separate computers!
(Counter to popular thought – GPUs are great for small problems)
ParallelCounter is also an accumulator!

Actual code (C++ template arg)

```
template <class ACC_TYPE, uint32_t N_ATOMIC=32>
struct ParallelAccumulator {
    private:
        ACC_TYPE accum[N_ATOMIC];

    public:
        __device__ ParallelAccumulator<ACC_TYPE, N_ATOMIC>() {
        }
        __device__ ~ParallelAccumulator<ACC_TYPE, N_ATOMIC>() {
        }
        inline __device__ ACC_TYPE operator-(ACC_TYPE x) {
            return atomicSub(accum + (threadIdx.x % N_ATOMIC), x);
        }
        inline __device__ ACC_TYPE operator+=(ACC_TYPE x) {
            return atomicAdd(accum + (threadIdx.x % N_ATOMIC), x);
        }
        // spread the values across the accumulator
        __device__ void set(ACC_TYPE x) {
            for(int i=0; i < N_ATOMIC; i++) accum[i]=x;
        }
        inline __device__ ACC_TYPE getValue() {
            // simplest slow method for right now.
            ACC_TYPE sum=0;
            for(int i=0; i < N_ATOMIC; i++) { sum += accum[i]; } return sum;
        }
    }
```
Parallel Concurrent Sorting

Actual code

```c
__global__ void bitonic_sort_step(float *data, int j, int k, int n)
{
    //unsigned int i, ixj; /* Sorting partners: i and ixj */
    for(unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;
        i < n; i += blockDim.x * gridDim.x) {
        unsigned int ixj = i^j;
        /* The threads with the lowest ids sort the array. */
        if ((ixj)>i) {
            if ((i&k)==0) {
                /* Sort ascending */
                if (data[i]>data[ixj]) {
                    /* exchange(i,ixj); */
                    float temp = data[i];
                    data[i] = data[ixj];
                    data[ixj] = temp;
                }
            } else {
                /* Sort descending */
                if (data[i]<data[ixj]) {
                    /* exchange(i,ixj); */
                    float temp = data[i];
                    data[i] = data[ixj];
                    data[ixj] = temp;
                }
            }
        }
    }
}
__global__ void k_bitonic_sort(float *data, int n, int powOfTwo)
{
    int nThreadsPerBlock = 512;
    int nBlocks = (powOfTwo-n)/nThreadsPerBlock;
    nBlocks /= 64;
    nBlocks = (nBlocks==0)?1:n Blocks;
    if(n < powOfTwo)
        bitonic_init<<<nBlocks, nThreadsPerBlock>>>(data, n, powOfTwo);
    nThreadsPerBlock = (powOfTwo > nThreadsPerBlock)?nThreadsPerBlock:powOfTwo;
    nBlocks = powOfTwo/nThreadsPerBlock;
    nBlocks /= 64;
    nBlocks = (nBlocks==0)?1:nBlocks;
    int j, k;
    /* Major step */
    for (k = 2; k <= powOfTwo; k *= 2) {
        /* Minor step */
        for ((j>>1); j>0; j>>1) {
            bitonic_sort_step<<<nBlocks, nThreadsPerBlock>>>(data, j, k,powOfTwo);
        }
    }
}
```
Concurrent Parallel Sorting
(10k floats, 15 streams, k40c)

- Simple Bitonic: 4.5 ms
- Simple Qsort: 246.9 ms

Great job NVIDIA!

- Nice scheduling trapezoid!
- Recursion is expensive

Recursion is expensive (not surprising)
Lots of heterogeneity, variable sampling, varying certainty of measure: Umm... what can we trust?

Protein differences between states cause disease
- Proteins are the structures and machines of life
- Biology varies: heterogeneous observations with large scatter

Mass spectrometry (MS) reads out protein differences
- MS data are not Gaussian and change (calibration, temp., sample, etc.)
- Non-linear measurement reliability as a function of intensity
- Unpredictable number of observations
- => Control everything internally and translate all observations to p-values.
  - Work in those.

High variability at low S/N or at saturation

Low variability within linear range of measure

People can and do look at a single overall measure (e.g. p=0.08)
- Then go and do 3 years and $500k of research
- Only to come back and say “Hmm, can I look at that spectrum again?”
- Often they then find out they are looking at highly variable, low-intensity observations. (OUCH!)

Cold Spring Harbor Laboratory MS analysis pipeline
POC: Dr. John Wilson (jwilson at cshl.edu)
Public URL forthcoming
• **Bootstrapping**: non-parametric resampling technique that works for all distributions and small $n$; allows determination of certainty of measure
  • Steps: sample observations, combine $p$ values, repeat $\geq 10,000$ times, sort, find CIs
• **Example**: 18 heterogeneous measurements
  • $p$ from 0.007 (down regulated) to 0.14 (up reg.)
  • Most say down reg.; overall $p = 0.08$ (maybe down reg.)
  • HOWEVER, 95% CI = 0.001 - 0.558!
  • In ~30-40% of observation sets, upper 95% CIs exceed choice of alpha

<table>
<thead>
<tr>
<th>p-val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.028534</td>
</tr>
<tr>
<td>0.262530</td>
</tr>
<tr>
<td>0.286691</td>
</tr>
<tr>
<td>0.609363</td>
</tr>
<tr>
<td>0.783723</td>
</tr>
<tr>
<td>0.378550</td>
</tr>
<tr>
<td>0.252444</td>
</tr>
<tr>
<td>0.067699</td>
</tr>
<tr>
<td>0.859116</td>
</tr>
<tr>
<td>0.292653</td>
</tr>
<tr>
<td>0.834993</td>
</tr>
<tr>
<td>0.471238</td>
</tr>
</tbody>
</table>

GPU's for interactive answers
Timeline running on k40c + k20c

Cold Spring Harbor Laboratory MS analysis pipeline
POC: Dr. John Wilson ([jwilson at cshl.edu](mailto:jwilson@cshl.edu))
Public URL forthcoming
SurveyFit™

Find the distribution that best fits your data

info at statperfect.com

- Parametric modeling can be extremely helpful
  - Forecasting, interpolation, understanding, speed, simplification
- **Assumptions** and/or the wrong model can have drastically terrible effects
  - “Recipe for Disaster: The Formula That Killed Wall Street” (Wired 2/09)
  - Oops, financial markets are not Gaussian distributed

  - Many hundreds of probability distribution functions exist
  - Many techniques exist to fit data
  - Many goodness of fit tests
    - Measure how well the model describes the data
  - Techniques to determine the relative quality of a model also exist

SurveyFit™ solves every distribution (in library)

😊 With every technique (in library)

😄 Ranks everything against everything

😊 Enabled by massive parallel processing

Multicore + K40c + k20c + GT640
Kriging Interpolation and Graph Algorithms

Kriging Interpolation
• Heavily used by the Air Force and GIS services
• Widely used in the domain of spatial analysis and computer experiments

Graph Algorithms (sadly no time! 😞)
• Heavily used in social media analysis
• See mpgraph & Graphlab
Parallel Kriging
(single K20c delivers a 55x speedup over a quad core Xeon)

Three kriging steps

1. Estimation of the semivariogram
2. Semivariogram model fitting.
3. Expression of the N_TILE x N_TILE solution of the kriging weights and creation of the raster points.

Parallel Implementation
(one kriging result() call per thread)

Strong scalability of Magma sgetrf() by GPUs

4-core Xeon | 825 ms
Nvidia K20c  | 15 ms

2x GPU = 2x faster
4x GPU = 4x faster
16x GPU = …
Thank you!
(Although there is so much more to say!)

Rob Farber
Chief Scientist, BlackDog Endeavors, LLC

Contact info at blackdogendeavors.com for consulting, teaching, writing, and other inquiries
Looks sort of Gaussian
SurveyFit™ animation: Gaussian (sort of fits)

- And a Gaussian sort of fits
SurveyFit™ animation: General Logistic

• General logistic looks better...
Hypersecant is yet better
SurveyFit™ animation: Johnson SU

• Johnson SU is by far the best fit
SurveyFit™ animation: Results (Johnson SU vs Gaussian)

- Gaussian compared to Johnson SU
- The actual SurveyFit™ product examines hundreds of distributions and many methods
Thank you!
(Although there is so much more to say!)

Rob Farber
Chief Scientist, BlackDog Endeavors, LLC

Contact info at blackdogendeavors.com for consulting, teaching, writing, and other inquiries