Dense Semiring Linear Algebra on Modern CUDA Hardware

Vijay Thakkar (thakkarv@gatech.edu)
SIAM CSE ’21 – MiniSymposium on GraphBLAS

Vijay Thakkar, Richard Vuduc
Georgia Institute of Technology

Ramakrishnan Kannan, Piyush Sao, Hao Lu,
Drahomira Herrmannova, Robert Patton, Thomas Potok
Oak Ridge National Laboratory
Related Work

• “Solving path problems on the GPU”
  • Buluç, Gilbert and Budak

• GraphBLAS libraries and toolkits:
  • SuiteSparse : github.com/DrTimothyAldenDavis/SuiteSparse
  • Grunrock Graph Blast : github.com/gunrock/graphblast
  • C++ CombBLAS : github.com/PASSIONLab/CombBLAS

• Key differences:
  • Strict focus on dense semiring kernels
  • Target near speed of light performance (SoL) on Volta and beyond only
  • A backend library for other GraphBLAS frameworks
Motivation
Motivation – Graph Algorithms

• Accelerated all pairs shortest path (APSP)
  • With applications in medical literature mining for drug discovery
  • Tackle knowledge interpretability despite literature flood

• Traditional approaches either:
  • Paint APSP like a dense linear system (Floyd-Warshall, FW)
  • Run Dijkstra’s on every vertex (Johnson’s)

• Supernodal APSP (Sao et. al. PPoPP’20):
  • Exploits graph sparsity for FW
  • Leverage dense linear algebra for supernodes

Fig 1: A Supernodal matrix
Motivation – Ease of Acceleration

• Unstructured sparsity is hard
  • Generality of unstructured sparsity -> degraded average performance

• Dense linear algebra is very well accelerated
  • Well understood data movement strategies

• Embrace dense semiring linear algebra
  • “If you build it, they will come.”
cuASR: CUDA Algebra for Semirings
A GraphBLAS Backend Library

Image from https://github.com/nvidia/cutlass
Features of cuASR

• SRGEMMs implemented in <10 lines of C++
• OOTB tuned configurations for common semirings
• Minimal knowledge of CUDA required
• Extensive test and benchmark infrastructure
• Header only template library

• Based on github.com/nvidia/cutlass

Fig 2: A tropical semiring; the unofficial logo of cuASR.
<table>
<thead>
<tr>
<th></th>
<th>GEMM</th>
<th>Tropical SRGEMM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial Accumulator Value</strong></td>
<td>0</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>(Identity of addition operator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Core Compute Operator</strong></td>
<td>$(+,\times)$</td>
<td>$(\text{min}, +)$</td>
</tr>
<tr>
<td><strong>Epilogue Operation</strong></td>
<td>$C_{i,j} = \alpha \cdot C_{i,j} + \beta \cdot A_{i,<em>} B_{</em>,j}$</td>
<td>$C_{i,j} = \min(C_{i,j} , A_{i,<em>} B_{</em>,j})$</td>
</tr>
</tbody>
</table>
Example Usage: APSP

using cuASR_MinPlus_fp32_SRGEMM_NT =
cuasr::gemm::device::Srgemm<
cuasr::minimum<float>, // Addition op
cuasr::plus<float>, // Multiplication op
float, // A type
ColumnMajor, // A layout
float, // B type
RowMajor, // B layout
float, // C type
ColumnMajor, // C layout
float // D type
>;

float alpha = MultiplicationOp::Identity;
float beta = do_epilogue_min
  ? MultiplicationOp::Identity
  : MultiplicationOp::Annihilator;

// launch SRGEMM kernel
cuASR_MinPlus_fp32_SRGEMM_NT kernel;
kernel({ M, N, K },
  { A, lda },
  { B, ldb },
  { C, ldc },
  { D, ldc },
  { alpha, beta });

Step 1: Stamp out desired SRGEMM template
Step 2: Use kernel

Step 3: ???
Step 4: PROFIT!!
Example Usage: Custom Semiring

```cpp
template <typename T, int N = 1>
struct binary_xor {
    static T constexpr Identity = static_cast<T>(false);
    // expose base scalar operator
    __host__ __device__
    T operator()(T lhs, T const &rhs) const {
        lhs ^= rhs;
        return lhs;
    }
};
```

Step 1: define a 10 line scalar operator struct + identity element

```
using cuASRGaloisFieldSrgemm = cuasr::gemm::device::Srgemm<
    binary_xor<int>,
    cuasr::binary_and<int>,
    // Datatypes and tile sizes
    // not OOTB semiring ...
    // ... must tune tile shapes manually
>
```

Step 2: Stamp out kernel template

Operator identity is used to init registers to elide epilogue source loads
Speeds and Feeds
SRGEMM Performance Implications

• Take (min, plus) SRGEMM as an example
• Reevaluate peak performance
  • No fused min-plus instruction in hardware halves theoretical peak flop/s
  • Performance must be evaluated based on instruction throughput
• Normalize performance w.r.t. instruction count

// GEMM Core instruction (PTX):          // Tropical SRGEMM Core instructions (PTX):

FFMA R55, R75.reuse, R78, R55 ;          FADD R109, R72.reuse, R80 ;
FSETP.GEU.AND P0, PT, R109.reuse, R4.reuse, PT ;
FSEL R109, R109, R4, !P0 ;
SRGEMM Performance Implications

- 6.81 TFlop/s fp32 SRGEMM w/ minimal tuning (V100 on Summit)
- 95% GEMM’s throughput when normalized for instruction count
- High utilization of hardware

Fig 3: Increase in instruction count in the unrolled core compute loop between GEMM and SRGEMM, and the corresponding drop in throughput

Fig 4: Throughput comparison between GEMM and SRGEMM when normalized for the instruction count in core unrolled compute loops. 5% delta likely affected by many 2\textsuperscript{nd} order effects e.g. occupancy drop from increased register consumption, iCache pressure etc.
Table 1: Performance of various SRGEMMs implemented in cuASR on PCIe V100. Input size of 4096x4096, identical layouts and kernel tiling strategies. Note that here, performance corresponds to abstract semiring add/multiply operations.

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Performance (TFLOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+, ×)</td>
<td>13.1</td>
</tr>
<tr>
<td>(min, +)</td>
<td>5.89</td>
</tr>
<tr>
<td>(max, +)</td>
<td>5.89</td>
</tr>
<tr>
<td>(min, ×)</td>
<td>5.89</td>
</tr>
<tr>
<td>(max, ×)</td>
<td>5.89</td>
</tr>
<tr>
<td>(min, max)</td>
<td>3.29</td>
</tr>
<tr>
<td>(max, min)</td>
<td>3.29</td>
</tr>
<tr>
<td>(or, and)</td>
<td>1.79</td>
</tr>
</tbody>
</table>
Split-K GEMM

• GEMM on Skinny-Tall matrices hard to do:
  • Reduction mode (K) dominates
  • 2D decomposition of output tile means very few threads launched

• Solution – 3D GEMM:
  • Compute partial inner products splitting (K) mode
  • Reduce partial inner products to output

• Great for supernodes
  • May have skewed aspect ratios

Fig 5: Performance of split-K (3D) min-plus SRGEMM (input size 128x4096)
Application Integration: Medical Text Mining

Scalable Knowledge Graph Analytics at 136 Petaflop/s – SC’20 Gordon Bell Finalist

Image from https://github.com/nvidia/cutlass
Looking Forward
Take Aways

• “Globally Sparse, Locally Dense”
  • Play to the strengths of hardware

• Open-source libraries from vendors are critical

• For future hardware:
  • Future architectures with ISA level composability
    • Think semiring FMA instruction!
  • Tensor-core and FF hardware do not support GraphBLAS
    • Need better perf communication
  • Tell your vendor about your use case!
Visit cuASR.io for details