Optimization Case Study for Kepler K20 GPUs: Synthetic Aperture Radar Backprojection

Thomas M. Benson [1] Daniel P. Campbell [1] David Tarjan [2] Justin Luitjens [2]

[1]Georgia Tech Research Institute – {thomas.benson,dan.campbell}@gtri.gatech.edu

[2]NVIDIA Corporation – {dtarjan,jluitjens}@nvidia.com

GPU Technology Conference, Session S3274, March 19, 2013

SAR Backprojection Overview

- Synthetic aperture radar (SAR) is a radar-based imaging modality
- Backprojection (BP) is one form of image formation – it requires $O(N^3)$ operations (N pulses, $N \times N$ image)

https://www.sdms.afrl.af.mil/index.php?collection=ccd_challenge.

Backprojection Kernel with Linear Interpolation

```
for all voxels v do
    I_v = 0  % Initialize complex voxel to 0
    for all pulses p do
        R = ||(p_vox − p_plat)||  % Distance from platform to voxel
        bin = ⌊(R − R_0)/ΔR⌋  % Range bin (integer)
        if bin ∈ [0, L − 2] then
            w = (R − R_0)/ΔR − bin  % Interpolation weight
            % Phase history data sampled using linear interpolation
            s = (1 − w) · X[bin, p] + w · X[bin + 1, p]
            % exp(j · 2k_u · R) represents ideal reflector response
            I_v += s · exp(j · 2k_u · R)
        end if
    end for
end for
```

What is Good Enough?

- Double precision (FP64)? Single precision (FP32)? Mixed precision? Intrinsics? Approximations? Texture sampling?
- BP optimization involves mixed precision and approximations
- We do not focus on numerical requirements here, but do note that it has been widely reported that the range calculation requires double precision
- The sine and cosine requirements are more lax given an accurate argument

Error Metrics

We use a dB-scale signal-to-error ratio to judge numerical approximations:

$$SER_{dB} = 10 \log_{10} \left(\frac{\sum_i g_i^2}{\sum_i |g_i - t_i|^2} \right)$$

where g is the double precision reference image and t is the test image. We have also evaluated the results qualitatively and look for SER_{dB} values higher than 50.

Optimization Phases

- Algorithmic and numerical optimization
  - High-level algorithmic optimization
  - Numerical approximations
- Architecture-specific optimization
  - Incorporate architecture-specific instructions
  - Exploit memory hierarchy
  - Profiling, occupancy and register utilization, loop unrolling, autotuning, etc.
- The above are inter-dependent – architecture features guide appropriate algorithmic and numerical optimizations
- We focus on the latter phase for this talk

Methodology

- Start with double precision implementation and apply incremental optimizations
- Track the impact of successive optimizations
- This is not perfect
  - Optimizations are inter-dependent, so ordering matters
  - We autotune at certain stages, but that finds local rather than global optima
- We use CUDA 5.0 and driver 310.32 for all experiments
- We report all results in giga backprojections per second (GBP/s)

Algorithmic and numerical optimizations

- V1: Baseline – FP64 for all intermediate calculations
- V2: Mixed precision – FP64 for range calculation, FP32 for linear interpolation and accumulation
- V3: Incremental phase calculations[1] – High fidelity phase lookup table and intrinsic sincos instead of FP64 sincos
- V4: Two-step Newton-Raphson (NR) for square root
- V5: One-step NR with pulse blocking for square root

	K20c GBP/s	C2050 GBP/s	SER
V1	5.2	2.1	–
V2	5.9	2.3	118.7 dB
V3	9.2	3.8	112.1 dB
V4	10.7	4.6	77.7 dB
V5	11.0	5.4	62.9 dB

[1] T. M. Benson, D. P. Campbell, D. A. Cook, "Gigapixel Spotlight Synthetic Aperture Radar Backprojection Using Clusters of GPUs and CUDA", 2012 IEEE Radar Conference, pp. 853–858.

Read-Only Data Cache

- With CC 3.5, we can directly access the read-only data cache without using textures
- The compiler may use such reads for const __restrict__ pointers, but we directly use the __ldg() intrinsic. For example,

```
const float2 lutEntry = __ldg(lutPtr + index);
```

  instead of

```
const float2 lutEntry = lutPtr[index];
```

- Minimal code change ⇒ easy empirical evaluation

Read-Only Data Cache Results

	Baseline	X	LUT	Plat	LUT/Plat
V1	5.2	5.3	5.2	5.7	5.7
V2	5.9	5.8	5.9	6.0	6.0
V3	9.3	9.1	9.2	9.3	9.2
V4	10.7	10.6	10.6	12.0	11.5
V5	11.0	11.0	11.7	12.5	12.9

All results in GBP/s.

- X := phase history data, Plat := platform positions
- V5 has the lowest arithmetic intensity ⇒ memory optimization more important
- We will ultimately use a combination of constant, shared, and texture memory, but quickly evaluating read-only cache impact is very useful

Texture Sampling

- Backprojection includes linear interpolation $\Rightarrow$ can leverage texture sampling (V6)
- Texture sampling is reduced precision, but data can be upsampled ($O(N^2 \log N)$) prior to backprojection ($O(N^3)$) to increase accuracy

	K20c GBP/s	C2050 GBP/s	SER
V5	11.0	5.4	62.9 dB
V6	14.7	7.5	59.0 dB

Constant and Shared Memory

- V7: Constant memory
  - Platform positions (24B/pulse) in constant memory
- V8: Shared memory
  - Incremental phase calculation LUT in shared memory
  - LUT can be large, so first calculate the portion needed for the image chip being processed by a given block and load only the relevant entries

	K20c GBP/s	C2050 GBP/s	SER
V6	14.7	7.5	59.0 dB
V7	18.9	8.2	59.0 dB
V8	19.9	8.5	59.0 dB

Source-level optimizations

Workflow: Inspect PTX for "missed opportunities", check SASS to confirm issues, modify code; lather, rinse, repeat

Example: Newton-Raphson update. $x_1 = x_0 - (x_0 * x_0 - \alpha) * (0.5/x_0)$

```
// Outside of loop -- common subexpression elimination
mul.f64 %fd5, %fd3, %fd3;          [ x0*x0 ]

// Inner loop
sub.f64 %fd34, %fd5, %fd33;        [x0 * x0-α]
mul.f64 %fd35, %fd34, %fd4;        [(x0 * x0 − α)*(0.5/x0)]
sub.f64 %fd36, %fd3, %fd35;        [x0−(x0 * x0 − α) * (0.5/x0)]
```

Missed opportunity: We are not using fused multiply-add (FMA) instructions for this calculation.

Source-level optimizations (FMA)

Rewrite $a - b * c$ as either $a + (-b) * c$ or $a + b * (-c)$.

Revised Newton-Raphson update. $x_1 = x_0 + (x_0 * x_0 - \alpha) * (-0.5/x_0)$

```
// Outside of loop -- common subexpression elimination
mul.f64 %fd6, %fd4, %fd4;                [ x0*x0 ]

// Inner loop
sub.f64 %fd33, %fd6, %fd32;              [x0 * x0-α]
fma.rn.f64 %fd34, %fd33, %fd5, %fd4;     [x0+(x0 * x0 - α)*(0.5/x0)]
```

Applied this to several cases. For example,
$(a - b_{\text{const}}) * c_{\text{const}} \Rightarrow a * c_{\text{const}} + (-b_{\text{const}} * c_{\text{const}})$

Source-level optimizations (type conversions)

Examining PTX also revealed some avoidable type conversions

```
for (int pulse = 0; pulse < N; ++pulse) {
    ...
    tex2D(..., pulse + 0.5f); // pulse converted to float
    ...
}
```

Which can be eliminated:

```
float pulse_f = 0.5f;
for (int pulse = 0; pulse < N; ++pulse) {
    ...
    tex2D(..., pulse_f);
    pulse_f += 1.0f;
    ...
}
```

	K20c GBP/s	C2050 GBP/s	SER
V8	19.9	8.5	59.0 dB
V9	21.9	9.4	59.0 dB

Multiple Pixels Per Thread

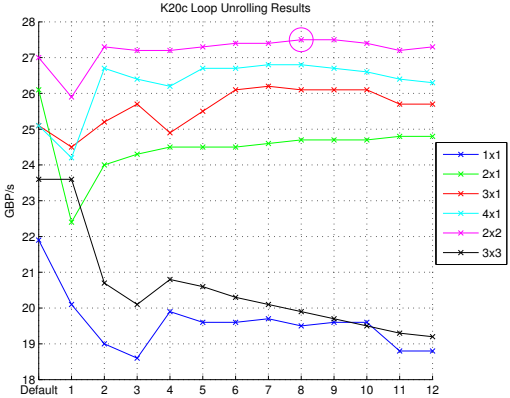
- Can amortize some redundant costs by processing multiple pixels per thread
- Compact groups of pixels have locality benefits

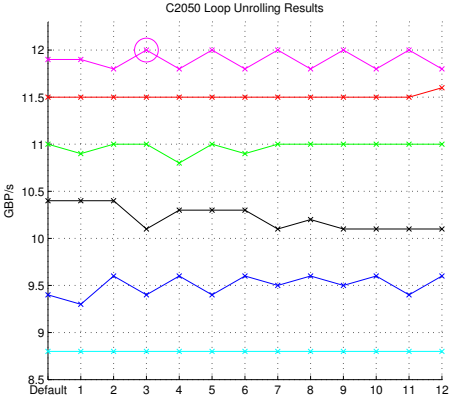
Group	K20c GBP/s	Reg	C2050 GBP/s	Reg	SER
1x1	21.9	47	9.4	56	59.0 dB
2x1	26.1	51	11.0	56	57.3 dB
3x1	25.1	54	11.5	62	56.9 dB
4x1	25.1	61	8.8	63	53.7 dB
2x2	27.0	59	11.9	63	57.3 dB

Reg column indicates the kernel register usage. SER_{dB} varies due to differing initial estimates for Newton-Raphson square root solves.

Autotuning: Loop Unrolling

- Previous results did not use `#pragma unroll` to specify an unrolling factor (but the compiler still unrolls)
- Autotune by sweeping from `#pragma unroll 1` to `#pragma unroll 12`





Autotuning slightly improves results on both Fermi and Kepler.

Autotuning: Max Register Count Motivation

- Can specify the max register count (`-maxrregcount`)
- Assume 128 threads/SM[X] (more autotuning!)
- Kepler: 64K registers/SMX, 255 max registers/thread
  - 46 registers/thread $\Rightarrow$ 11 max blocks/SMX
  - 51 registers/thread $\Rightarrow$ 10 max blocks/SMX
  - 56 registers/thread $\Rightarrow$ 9 max blocks/SMX
  - 64 registers/thread $\Rightarrow$ 8 max blocks/SMX
  - 73 registers/thread $\Rightarrow$ 7 max blocks/SMX
- Fermi: 32K registers/SM, 63 max registers/thread
  - 42 registers/thread $\Rightarrow$ 6 max blocks/SM
  - 51 registers/thread $\Rightarrow$ 5 max blocks/SM

Autotuning: Max Register Count Results

K20c autotuning results:

Max Reg	Used Reg	Max Perf (GBP/s)	Unroll factor	Group
46	46	27.2	2	2x2
51	51	27.6	1	2x2
56	56	27.9	unspecified	2x2
64	56	27.8	2	2x2
73	58	27.1	2	2x2
128	56	27.9	2	2x2

C2050 autotuning results:

Max Reg	Used Reg	Max Perf (GBP/s)	Unroll factor	Group
36	36	12.0	3	2x2
42	42	11.6	1	2x2
51	50	11.9	4	2x2
63	61	12.0	3	2x2

In cases where multiple configurations achieved the highest performance, the minimum register utilization is reported.

“These go to 11”

The max supported clock frequency exceeds the default clock

```
nvidia-smi -q
...
Applications Clocks
    Graphics                        : 705 MHz
    Memory                          : 2600 MHz
Max Clocks
    Graphics                        : 758 MHz
    SM                              : 758 MHz
    Memory                          : 2600 MHz
...
shell> nvidia-smi --application-clocks=2600,758
...
Applications Clocks
    Graphics                        : 758 MHz
    Memory                          : 2600 MHz
...
```

Summary of Results

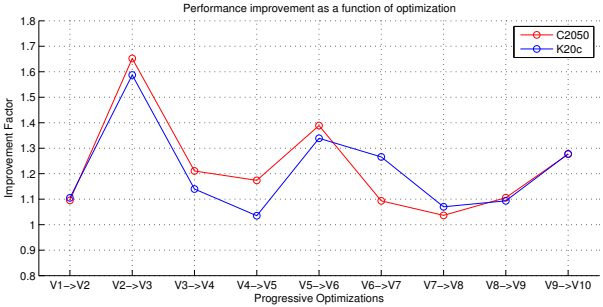
	K20c[2]	K20c[3]	C2050	SER
V1	5.2	5.7	2.1	–
V2	5.9	6.3	2.3	118.7 dB
V3	9.2	10.0	3.8	112.1 dB
V4	10.7	11.4	4.6	77.7 dB
V5	11.0	11.8	5.4	62.9 dB
V6	14.7	15.8	7.5	59.0 dB
V7	18.9	20.0	8.2	59.0 dB
V8	19.9	21.4	8.5	59.0 dB
V9	21.9	23.4	9.4	59.0 dB
V10	27.9	29.9	12.0	57.3 dB

Summary of results for all implementations in GBP/s. V10 corresponds to best achieved results for autotuning pixels/thread, loop unrolling factors, and maximum registers/thread.

[2] 705 MHz

[3] 758 MHz

Optimization Effectiveness for Fermi and Kepler



- V2 : Replaced FP64 with FP32 ⇒ Kepler (barely)
- V3,V4,V5 : Reduced total arithmetic operations ⇒ Fermi
- V6 : Reduced arithmetic operations and exploited texture cache – improved L1 cache hit rate on Fermi ⇒ Fermi
- V7,V8 : Exploit constant and shared memory ⇒ Kepler
- V9 : Reduced instruction count ⇒ Fermi
- V10 : More work per thread (high register utilization) ⇒ Equal win

Comparison to Previously Published Results

- Context for achieved performance: prior published results
  - Only showing optimized implementations on modern hardware
- Caveat: Run-time performance is not directly comparable without also considering achieved accuracy – working on an apples-to-apples comparison

Hardware	GBP/s	Peak (FP32/FP64)	Reference
Dual Intel Xeon E5-2670	7.4	664 / 332	[1]
Tesla C2050	11.7	1030 / 515	this
Intel Xeon Phi[4]	14.0	1920 / 960	[1]
Tesla K20c	29.9	3783 / 1261	this

[1] J. Park, P. Tang, M. Smelyanskiy, T. Benson, "Efficient backprojection-based synthetic aperture radar computation with many-core processors", Supercomputing 2012.

[4] Evaluation card. Included 60 cores at 1.0 GHz (vs 1.053 GHz for a 5110P)

Summary and Conclusions

- Determining required accuracy for floating point algorithms is critical for effective optimization
  - Metrics play a vital role, but they rarely exist a priori
  - Evaluating correctness requires domain expertise
  - Who determines required accuracy? Algorithm designer or HPC programmer?
- Optimization is an iterative process
- Over 5x performance improvement using many optimizations
  - Improved previously published C2050 results by over 2x
- Autotuning is your friend – optimal parameters are not obvious

Acknowledgements

We would like to thank NVIDIA for supplying early access to K20 hardware in order to carry out this performance evaluation.

This work supported in part by DARPA under contracts HR0011-10-C-0145 and HR0011-10-9-0008. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Thank You

- Thank you!
- Questions / Comments?