Autotuning Dense Batched QR Factorizations on GPU

Tim A. Davis
Wissam M. Sid-Lakhdar

Texas A&M University

February 25, 2017

Overview

Context

Solving sparse least squares problems

Problem :

$$\underset{x}{Min}||Ax - b||^2$$

with:

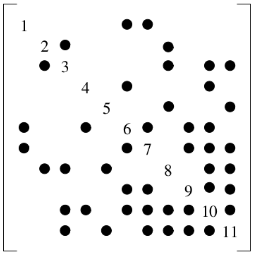
$A \in \mathbb{R}^{mxn}$ (large sparse matrix)
$b \in \mathbb{R}^m$
$x \in \mathbb{R}^m$

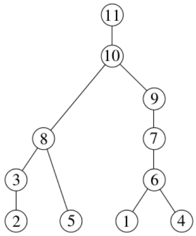
Solution : Factorization of A through the *Multifrontal QR method*

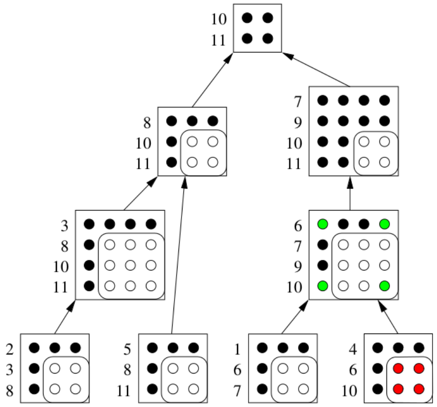
Architecture : Shared-memory computer with a single (or multiple) GPU(s)

Context

Multifrontal QR method (Duff, Reid, others)

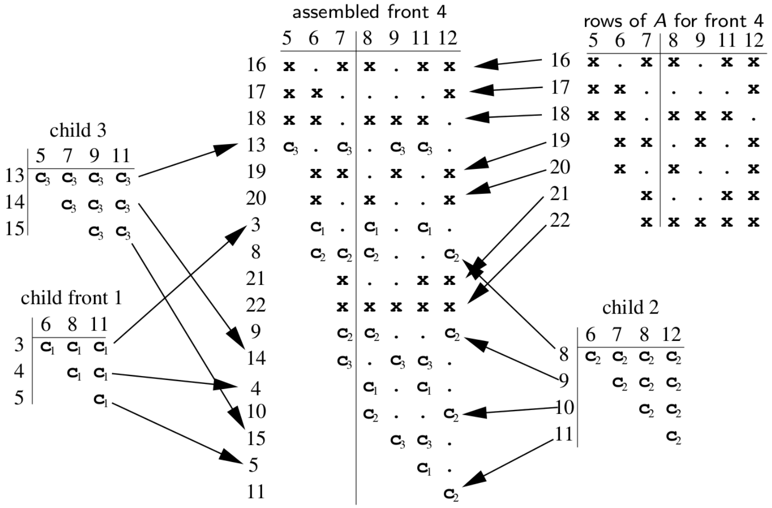






Context

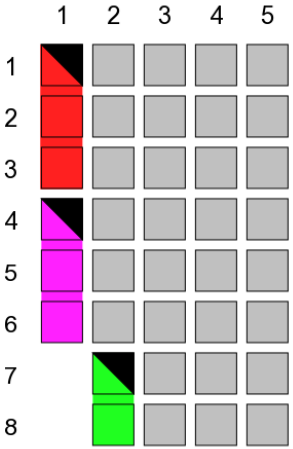
Multifrontal QR method (Duff, Reid, others)



Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

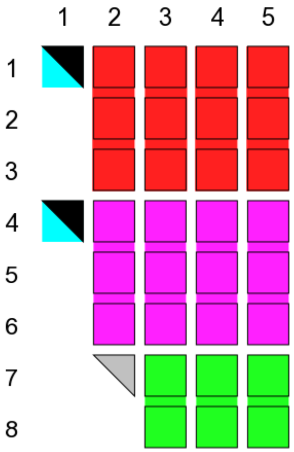
(1,2,3) new

(4,5,6) new

(7,8) new

Context
Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

🟥 (1,2,3) applied

🟪 (4,5,6) applied

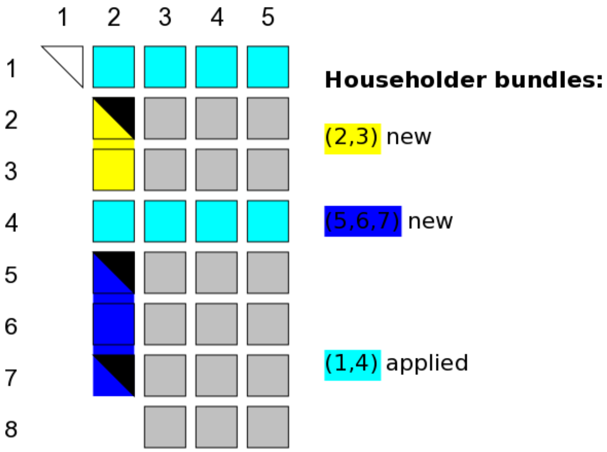
🟩 (7,8) applied

🟦 (1,4) new

Context

Bucket scheduler

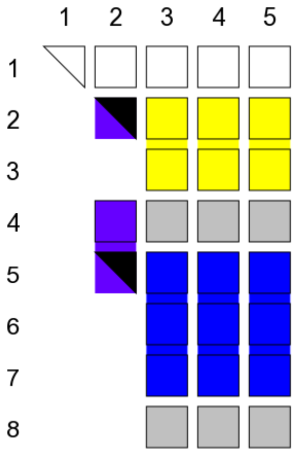
GPU-based QR factorization of a single frontal matrix



Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

(2,3) applied

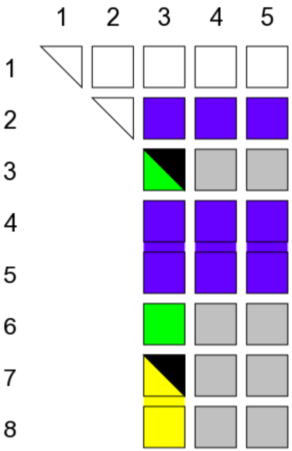
(5,6,7) applied

(2,4,5) new

Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

(3,6) new

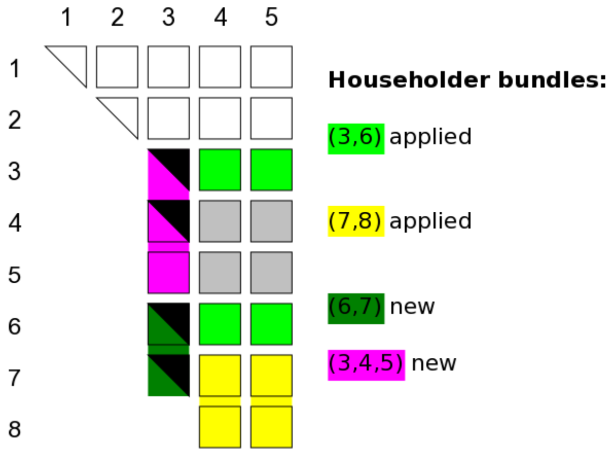
(7,8) new

(2,4,5) applied

Context

Bucket scheduler

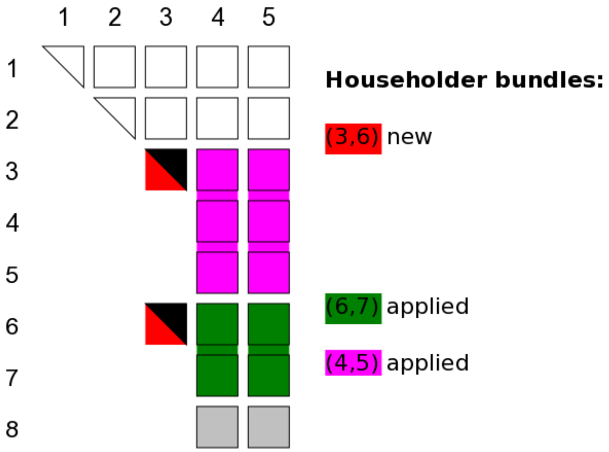
GPU-based QR factorization of a single frontal matrix



Context

Bucket scheduler

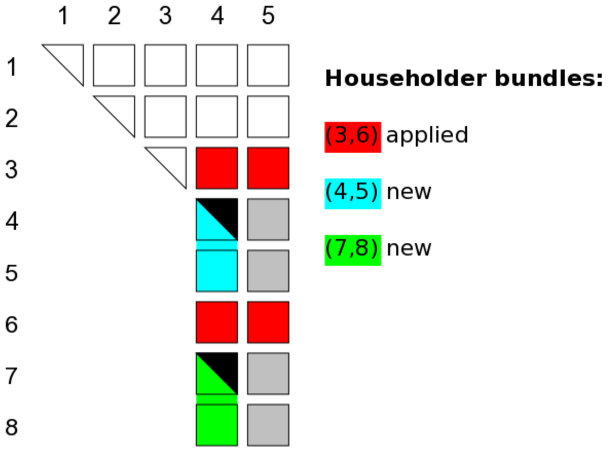
GPU-based QR factorization of a single frontal matrix



Context

Bucket scheduler

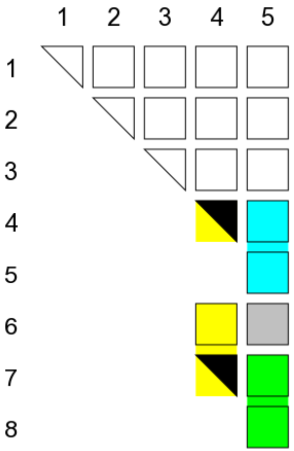
GPU-based QR factorization of a single frontal matrix



Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

(4,6,7) new

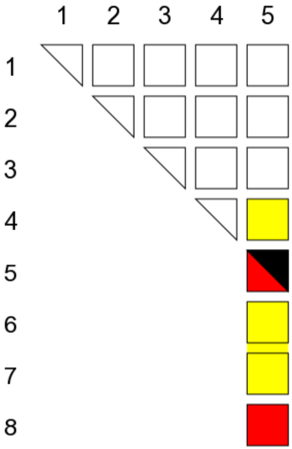
(4,5) applied

(7,8) applied

Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

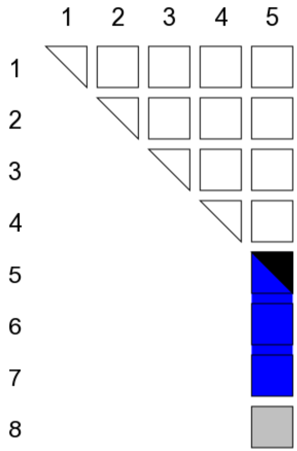
(4,6,7) applied

(5,8) new

Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



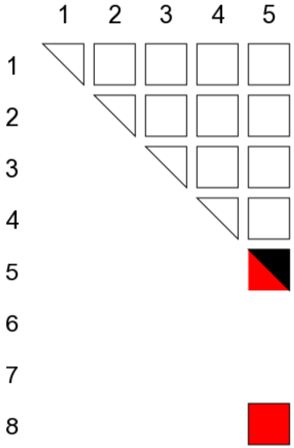
Householder bundles:

(5,6,7) new

Context

Bucket scheduler

GPU-based QR factorization of a single frontal matrix



Householder bundles:

(5,8) new

Need

Batched BLAS on GPU

- Bundles are formed of non-contiguous pebbles
- Low in the tree: **many small** fronts

  i.e. potential for parallel factorizations . . . (1)
- High in the tree: few large fronts, but with staircase shape

  i.e. advantageous to subdivide the factorization of a whole front (gross granularity) into **many smaller** ones (fine granularity) . . . (2)

Need

Batched BLAS on GPU

- Bundles are formed of non-contiguous pebbles
- Low in the tree: **many small** fronts

  i.e. potential for parallel factorizations . . . (1)
- High in the tree: few large fronts, but with staircase shape

  i.e. advantageous to subdivide the factorization of a whole front (gross granularity) into **many smaller** ones (fine granularity) . . . (2)

(1) + (2) $\Rightarrow$ Need for batched GEQRF and batched GEMM!

Old approaches

- First kernels:
  - targets *NVidia C2070* GPU (*Fermi* architecture) specifically
  - fixed pebble (tile) size 32x32
  - maximum bundle size 3x1 (96x32 matrix)
- Second kernels:
  - targets *NVidia K40* GPU (*Kepler* architecture) specifically
  - fixed pebble (tile) size 64x64 (but larger)
  - maximum bundle size 2x1 (128x64 matrix)
  - takes advantage of newer architectural features (*Shuffle* instruction, more registers, more shared memory, . . .)

Old approaches

- First kernels:
  - targets *NVidia C2070* GPU (*Fermi* architecture) specifically
  - fixed pebble (tile) size $32x32$
  - maximum bundle size $3x1$ (96x32 matrix)
- Second kernels:
  - targets *NVidia K40* GPU (*Kepler* architecture) specifically
  - fixed pebble (tile) size $64x64$ (but larger)
  - maximum bundle size $2x1$ (128x64 matrix)
  - takes advantage of newer architectural features (*Shuffle* instruction, more registers, more shared memory, . . .)
- Factorize and Apply kernels faster . . .

Old approaches

- First kernels:
  - targets *NVidia C2070* GPU (*Fermi* architecture) specifically
  - fixed pebble (tile) size 32x32
  - maximum bundle size 3x1 (96x32 matrix)
- Second kernels:
  - targets *NVidia K40* GPU (*Kepler* architecture) specifically
  - fixed pebble (tile) size 64x64 (but larger)
  - maximum bundle size 2x1 (128x64 matrix)
  - takes advantage of newer architectural features (*Shuffle* instruction, more registers, more shared memory, . . .)
- Factorize and Apply kernels faster . . .

  . . . but total sparse factorization time slower!

Old approaches

- First kernels:
  - targets *NVidia C2070* GPU (*Fermi* architecture) specifically
  - fixed pebble (tile) size 32x32
  - maximum bundle size 3x1 (96x32 matrix)
- Second kernels:
  - targets *NVidia K40* GPU (*Kepler* architecture) specifically
  - fixed pebble (tile) size 64x64 (but larger)
  - maximum bundle size 2x1 (128x64 matrix)
  - takes advantage of newer architectural features (*Shuffle* instruction, more registers, more shared memory, . . .)
- Factorize and Apply kernels faster . . .

  . . . but total sparse factorization time slower!
  - More work done within the Apply kernel rather than within the Factorize kernel in the first case
  - The Apply kernel relies on *BLAS*3 routines while the Factorize kernel relies on *BLAS*2 routines

Motivation and Goal

Portability or Efficiency?

Portability	(too general) Write one code that fits all GPU architectures but that is not the fastest / fast enough on any one of them
Efficiency	(too specific) Write the best code for a one GPU architecture but that will be much less efficient / will not work for other architectures
Effort	Writing an efficient code for every architecture is tedious and unsustainable.

Motivation and Goal

Portability or Efficiency?

Portability	(too general) Write one code that fits all GPU architectures but that is not the fastest / fast enough on any one of them
Efficiency	(too specific) Write the best code for a one GPU architecture but that will be much less efficient / will not work for other architectures
Effort	Writing an efficient code for every architecture is tedious and unsustainable.

How to get both Portability and Efficiency with a minimum Effort?

New approach

Within *NSF SparseKaffe project*

Autotuning

- Write a general template code that relies on a **set of parameters**.
- The Autotuner **generates**, **compiles**, **runs** and **checks** a kernel, for every combination of parameters.
- The Autotuner traverses the **parameters search space** in order to find the combination leading to the best (fastest) kernel, for any given GPU architecture.

Overview

Overview

Algorithm

Matlab

```
function [A V1 T] = vthqr_gpu (A)
    [m n] = size(A);
    T = zeros(min(m,n));
    for k = 1:min(m,n)
        [v, tau, s] = house_higham(A(k:m,k)) ;
        V1(k) = v(1);
        A (k+1:m,k) = v(2:end);
        z = -tau * v' * A(k:m,:);
        A(k:m,k+1:n) = A(k:m,k+1:n) + v * z(k+1:n);
        T(1:k-1,k) = T(1:k-1,1:k-1) * z(1:k-1)';
        T(k,k) = tau;
        A(k,k) = s;
    end
```

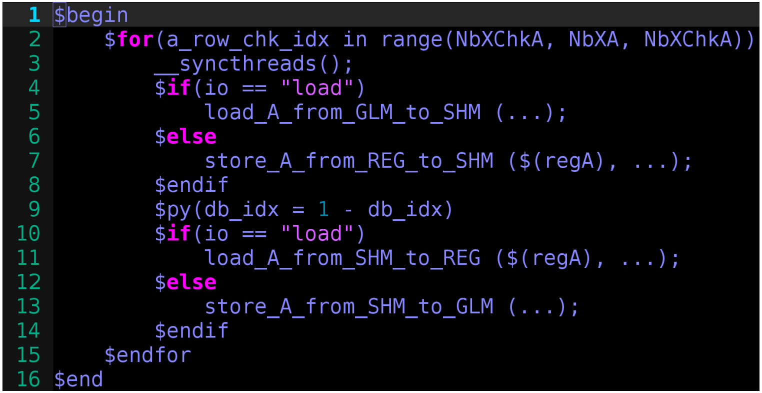
- QR factorization (for GPU)
- Householder *à la* Highim:
  - Numerical stability (when norm of Householder vector is small)
  - Less operations (most Householder vector entries stay unchanged) $\Rightarrow$ GPU friendly
- Computing and using the z vector allows for less branching (warp divergence) and for more parallelism

Template

Python/CUDA

- Given that principle of batched BLAS is to target many **small** matrices, we consider that small is whatever fits into the shared-memory and registers of a GPU. Thus, our strategy for the QR kernels is to load the whole matrix to be factorized into registers, do the whole factorizations using registers as data holders and shared-memory for intra-warp communication, and finally, store the results into global memory
- **PyExpander**: replacing and extending the *C* macros system by leveraging the power of *Python*
  - ability to use loops while very difficult and painful with macros
  - ability to have functions calling other functions or using variables, which is very difficult with C macros
  - nice checking done by the python compiler while hassle with dealing with non understandable errors with the C/CUDA compiler
  - even the Makefile is generated to take into account architecture type and optimization options

Code example



```
$begin
    $for(a_row_chk_idx in range(NbXChkA, NbXA, NbXChkA))
        __syncthreads();
        $if(io == "load")
            load_A_from_GLM_to_SHM (...);
        $else
            store_A_from_REG_to_SHM ($(regA), ...);
        $endif
        $py(db_idx = 1 - db_idx)
        $if(io == "load")
            load_A_from_SHM_to_REG ($(regA), ...);
        $else
            store_A_from_SHM_to_GLM (...);
        $endif
    $endfor
$end
```

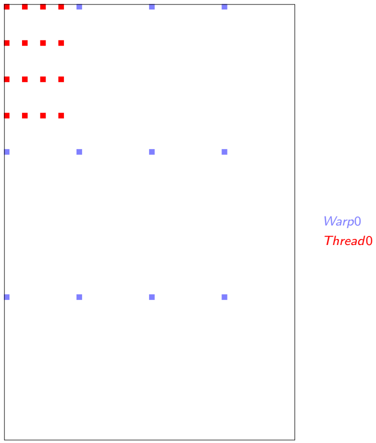
Template Code

PyExpander instructions evaluated by the Python interpreter

- $for ≈ #pragma unroll
- $if ≈ #ifeq ... #endif

Parameters

- **Problem**:
  - $TlSz$, $NbXTl$, $NbYTl$
  - Inputs (fixed for every configuration)
- **Architecture**:
  - $WpSz$, $NbTh$, $NbReg$
- **Mapping**:
  - $\begin{Bmatrix} Nb \\ Dt \end{Bmatrix} \begin{Bmatrix} Th \\ Wp \end{Bmatrix} \begin{Bmatrix} X \\ Y \end{Bmatrix} \begin{Bmatrix} A \\ T \end{Bmatrix}$
- **Load/Store**:
  - $NbXChkA$, $NbXChkT$
- **Code optimization**:
  - $X*, X\_1*, \ldots$
  - Switch between sub-algorithms
  - Replace *pragma* and *inline* of CUDA
- **. . .**: Many more parameters and routines exist, but they [depend on / are deduced from] the above core parameters



Search space

- Some parameters need to be of the form

$$2^i, i \in [0, n]$$

  in order to make the code simpler ($\Rightarrow$ faster)
- The search space for the *Mapping* parameters is bound by the value of the *Problem* parameters
- The search space for the *Architecture* and *Load/Store* parameters depend on the architectural characteristics of the targeted GPU
- The *Optimization* parameters are (most often) Booleans, used to turn On/Off some features

Constraints

- Equalities: enforce a *bijection* between matrices and threads
- Inequalities: prohibit out-of-memory accesses
- Conditional constraints

Examples

0 $NbTh * NbReg \leq NbMaxReg$
- Total # of registers cannot exceed architecture limit

1 $NbThXA * NbThYA * NbTh == TlSz^2 * NbXTl * NbYTl$
- Sum of threads' registers for A equals the surface of A

2 $NbThXA * DtThXA \leq TlSz * NbXTl$
- A thread cannot be mapped on rows outside of A

3 $NbWpXA * NbWpYA == WpSz$
- Layout of a warp respects its size

Positioning

Position of first row of first thread of a warp in matrix A

$$posWpXA = ((\frac{WpIdXA}{cx})*dx+(\frac{WpIdXA\&(cx-1)}{ex})*fx+(WpIdXA\&(ex-1))) \quad (1)$$

Position of thread in warp

$$posThWpXA = \frac{ThWpId}{NbWpYA} * DtWpXA \quad (2)$$

Position of first row of thread

$$posX0A = posWpXA + posThWpXA \quad (3)$$

Relative position of i^{th} row of a thread

$$posThXA(i) = i * DtThXA \quad (4)$$

Position of i^{th} row of a thread

$$posX(i) = posX0A + posThXA(i) \quad (5)$$

- $posThXA(i)$ and $posThYA(j)$ are straightforward to compute
- $posX0A$ and $posY0A$ are expensive to compute. Every thread computes them once only and stores them in dedicated registers

Implementation issues

- Template code is harder to read/write/modify than standard code
- CUDA optimization decisions are not easy to make in template code
- Over-use of the *select* statement

Autotuner

```python
def do(params, matlab_engine):
    try:
        expander(params)
        compiler()
        kernel_time = runner()
        checker(matlab_engine)
    except MyError as error:
        kernel_time = float(-error.code)
    return kernel_time
```

Overview

Optimization problem

- Non-Linear Problem
- Non-Linear Constraints
- Discrete search space instead of Continuous
- *MINLP* (Mixed Integer Non Linear Programming)
- Worst case scenario of most difficult optimization problem

$\Rightarrow$ Most standard optimization tools do not apply!

Optimization solutions

Stochastic

- Strategy 1: **Random sampling** (*Exploration*)
  - Not effective at finding an optimum …
  - … but effective at discovering regions of interest
- Strategy 2: **Mutation** (*Exploitation*)

  Idea Starting from an initial *gene* (valid combination of parameters), slightly modify (with probability distribution) some parameters (given a certain probability)
  - Does not necessarily converge to global optimum …
  - … but does converge to local minimums

Optimization solutions

Deterministic

- Strategy 3: **Exhaustive search**
  - For a 32x32 matrix:
    - Size of the *unconstrained* search space: **2437438960041984**
    - Size of the *constrained* search space: **45536**
- Strategy 3*bis*: **Exhaustive sub-space search**
  - Optimize sets of parameters independently, *i.e.*, fix some parameters in a first phase and optimize them in a second phase
  - Parameters for T similarly than that for A
  - Load/Store parameters
- If we assume that, for any given set of parameters, generating the code, compiling it, running it and checking its correctness takes about **10 seconds** ...

#rows	#cols	Space		SubSpace	
		#Combinations	Time (days)	#Combinations	Time (hours)
32	32	45536	5.2	788	2.2
64	32	93184	10.8	1563	4.3
128	32	57632	6.7	2266	6.3
256	32	11664	1.4	2683	7.5
512	32	0	0	0	0

Table: *Estimated* autotuning times for different matrix sizes

Dealing with hard constraints

- Deterministic case:
  - Too costly to evaluate the validity of all combinations in search space
  - **Backtracking**
    Classical algorithm for finding the solutions of constrained computational problems. It builds candidates incrementally and drops them as soon as it determines that they cannot lead to valid solutions.
- Stochastic case:
  - Too much rejections of potential candidates before finding valid ones
  - Too time consuming to code explicitly rules that ensure the validity of genes
  - **Space reduction**
    For every parameter, start from its whole space and reduce it until only one value remains. Parameters are set one after the other and all the constraints related to them are checked in order to eliminate the impossible values for the other parameters.

Autotuning parallelization

- Autotuning is an embarrassingly parallel process
- If a computer has more than one GPU, it is possible to lunch as many autotuning processes as they are GPUs available
- The load of the host (CPUs) does not disturb performance measure, as only code generation, compilation and checking occur on it ...
- ...However, every program should run on a separate specific GPU.
- We use the **cudaSetDevice($(GPU_ID))** routine to map an autotuner process with a specific GPU
- Our system (*backslash*) contains 24 CPUs and 8 K40 GPUs

Overview

Performance Results

K40 (Kepler)

- NVidia K40 GPU (Kepler architecture)
- Results format: GFlps/s (Time (ms))

#Row	#Col	MAGMA		CuBLAS	AutoTuner
		Release 2.2.0	Experimental		
32	32	3.64 (19.36)	7.98 (8.82)	18.83 (3.74)	19.89 (3.54)
64	32	8.83 (19.45)	18.8 (9.13)	27.64 (6.21)	23.97 (7.16)
128	32	16.9 (22.16)	28.7 (13.1)	27.86 (13.4)	22.97 (16.3)
256	32	21.8 (35.83)	34.0 (22.9)	18.11 (43.1)	19.41 (40.2)

Performance Results

Tegra (Maxwell)

- NVidia Tegra GPU (Maxwell architecture). Jetson TX1 embedding:
  - 4+1 ARM CPUs
  - one SM of a Maxwell GPU (compared to 15 SMs for the K40 GPU)
  - CPUs and GPU share the same memory
  - Lower clock frequency
- More instability of the results. Thus, we repeat every run 10 times (instead of 1) and record the best runtime. Every kernel lunch runs 1000 thread blocks on the SM (instead of 100 per SM)
- Best parameters are different than the ones for the K40 GPU
- Results format: GFlps/s (Time (ms))

#Row	#Col	MAGMA	CuBLAS	AutoTuner
32	32	0.14 (335.44)	1.36 (34.58)	1.59 (29.49)
64	32	0.10 (1095.4)	2.43 (47.12)	1.98 (57.91)
128	32	0.15 (1706.2)	1.95 (127.7)	1.86 (133.8)
256	32	0.17 (3074.7)	1.72 (302.7)	0.97 (534.3)

Counter intuitive results

- The best kernels are not the fastest ones!
- The fastest kernel in terms of GFlops may have a lower occupancy, while the optimal kernel might be slower sequentially, but since more parallel versions can run in parallel, the total computation time of the whole batch is lower.
- The optimal kernels induce register spill and use some local memory . . . but not too much
- There is a trade off between using some local memory and having a higher occupancy. The optimum is somewhere at the limit
- This result is counter intuitive, as when we handcraft a kernel, we always carefully try to avoid register spill and use of local memory.

Overview

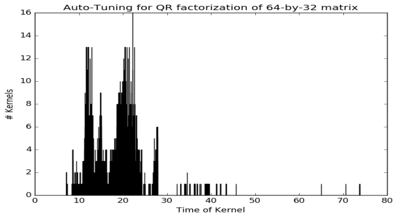
Summary

- Write once, use forever!
- Writing a templatized code is different / more challenging than customizing a code with a specific set of parameters . . .
  . . . But the hassle of tweaking the last bit of performance out of the code is transitioned from the library developper to the computer
- Several days / weeks necessary to find the best kernels for a given architecture, but this tedious work has to be done once only. The optimized kernel can then be packaged in the Batched BLAS library for the end users.
- For batched BLAS, the targeted matrix sizes are usually small . . .
  . . . trying to fit the whole matrices exclusively into GPU registers and shared-memory is beneficial over the traditional approach of using these only as temporary caches

Perspectives

Improving low-level kernels

- Given the time it takes to find the best kernel for a given matrix size exhaustively, improved (stochastic) search methods should be considered ⇒ relying on a *Meta-Model* could be the solution!



- Storing the T matrix in shared-memory instead of registers can be more efficient for tall and skinny matrices
- With an autotuner infrastructure available, it becomes easy to write any kind of Bathed BLAS routine. Indeed, we might be interested in the case of the QR factorization of a set of upper triangular matrices, as this case arises very often in the multifrontal QR method

Perspectives

Designing high-level kernels

- Limited size of matrices fitting in GPUs registers / shared-memory
- For larger matrices, rely on the previous kernels as building blocks
- Hierarchical factorization by row (*à la* CAQR) not applicable as combining the V and T matrices vertically is not feasible
- Hierarchical factorization by column will be considered (as already successfully achieved in the *BEAST* project)

Thank You!