Algorithmic Selection, Autotuning, and Scheduling for Accelerator-Based Codes for Numerical Linear Algebra

Piotr Luszczek

ICL UT INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

Funded by NSF

Motivation

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

$$\forall B_i = \cdot \mathit{POTRF}(A_i)$$

$$\forall B_i = \cdot \mathit{GEQRF}(A_i)$$

$$\forall B_i = \cdot \mathit{GETRF}(A_i)$$

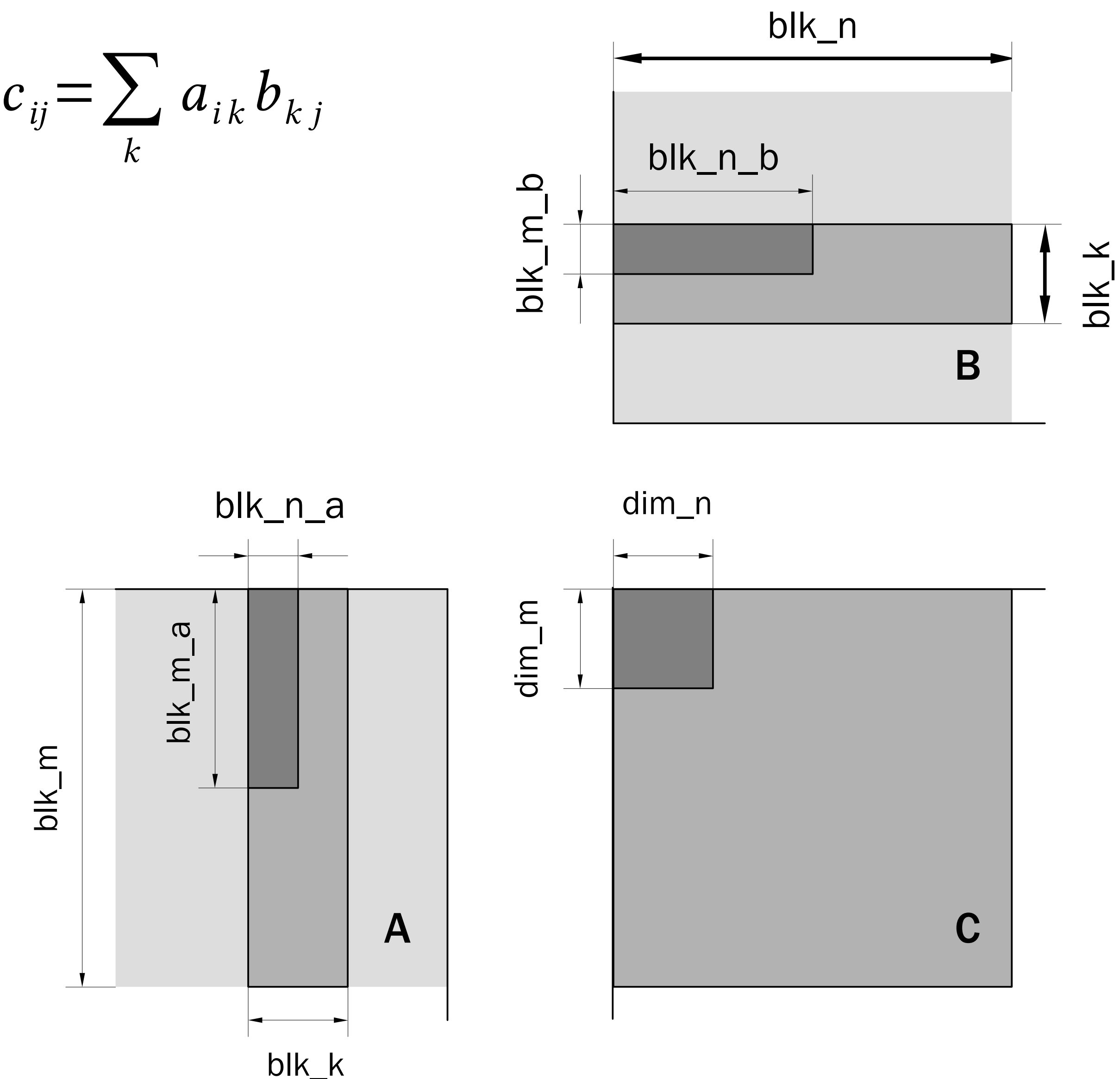
$$O_{n,k,p,q} = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} F_{k,c,r,s} D_{n,c,g(p,u,R,r,h)} \cdots$$

Compilation vs. Autotuning

- Compilation
  - Sometimes uses autotuning
  - Works for all codes
  - Finishes in seconds
  - Obeys the language syntax
  - Optimizes for machine model
  - Performs better for fixed sizes
- Autotuning
  - Often relies on the compiler
  - Works for some codes
  - Finishes when optimized
  - Delivers correct math
  - Optimizes over experimental data
  - Specializes in fixed sizes

Example: C = AB

$$c_{ij}=\sum_k a_{ik} b_{kj}$$



Example: C = AB - Parameters

- dim_m
- dim_n
- blk_m
- blk_n
- blk_k
- blk_m_a
- blk_n_a
- blk_m_b
- blk_n_b
- Vectoriazation
- Use shmem
- ...

Problem with Manual Iteration

- For dim_m = 32:1024
  - For dim_n = 32:1024
    - For blk_m = dim_m:dim_m:maxM
      - For blk_n = dim_n:dim_n:maxN
        - For blk_k = 16:maxK
          - For vectorize = “yes”, “no”
            - For fetch_A = “yes”, “no”
              - For texture = “none”, “1D”, “2D”
                - …
- But make sure that
  - dim_m*dim_n does not exceed the number of thread blocks for the tested card
  - There is enough shared memory
  - …

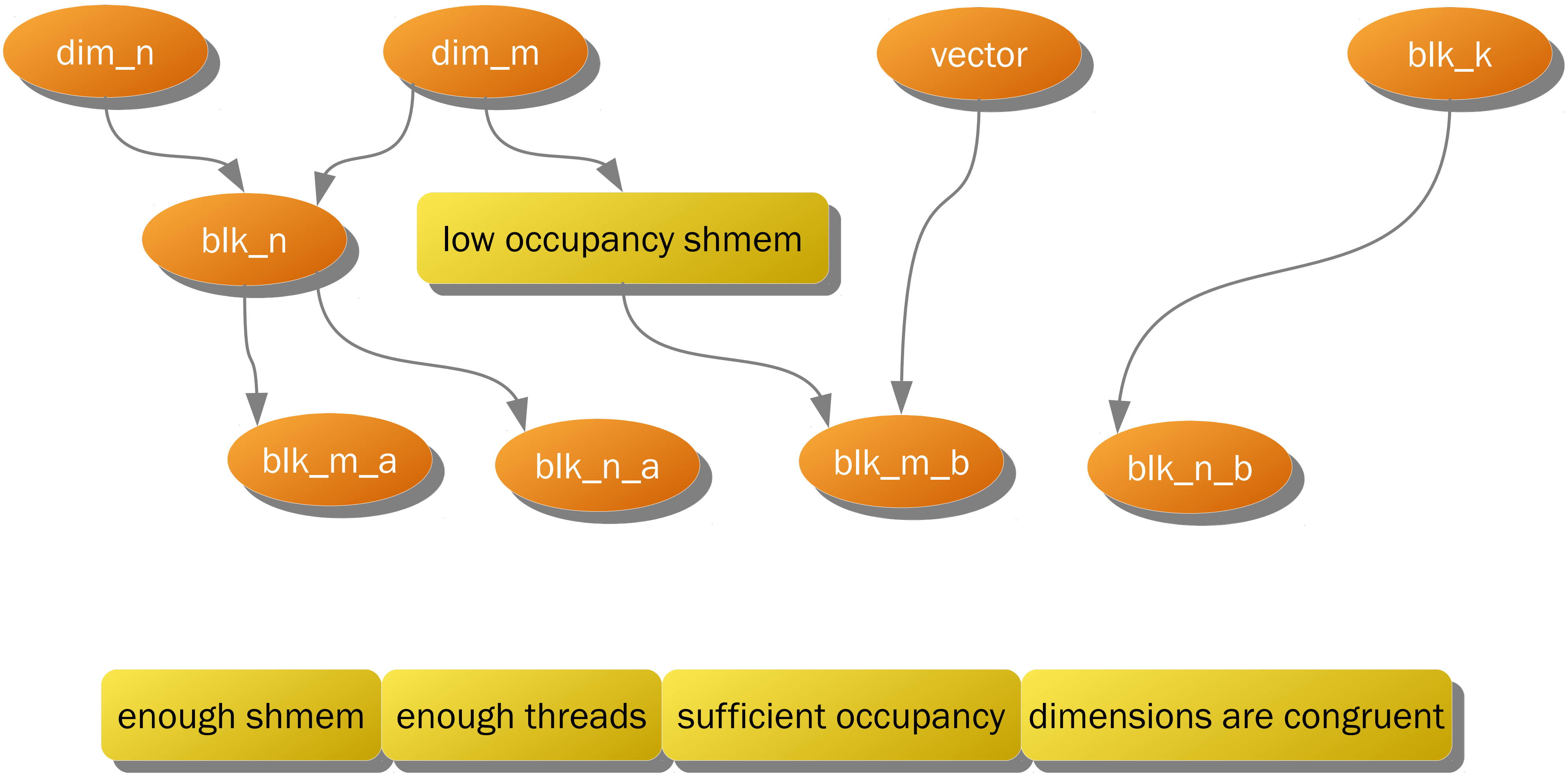
Iterator Basics: Declarative Approach

- Expression iterators
  - ```
    dim_m = range( 32, max_threads_dim_x, 32 )
    blk_m = range( dim_m, maxM, dim_m )
    ```
- Function iterators
  - ```
    @beast.iterator
    def blk_n_a():
        x = blk_k
        if trans_a != 0:
            x = blk_m
        return range(x, 0, -1)
    ```
- Closure iterators
  - ```
    @beast iterator
    def fibonacci():
        prev = next = 1
        while next <= largest_number:
            yield next
            next, prev = next+prev, next
    ```
    ```

Filter Basics

- Expression filters
  - over_max_threads = filter(block_threads > max_threads_per_block)
- Closure filters
  - ```
    @beast.filter
    def over_max_shmem():
        return block_shmem > max_shared_mem_per_block
    ```
    ```

Optimizations Summary

- The code generator figures out the optimal order
- Iterators become loops with proper nesting
- The nesting is determined by the dependence DAG
- Filters have to trigger as early as possible to prune the search space
  - Loop invariant code motion
- Type inference keeps the generated code fast
  - Scripting language iteration may be orders of magnitude slower

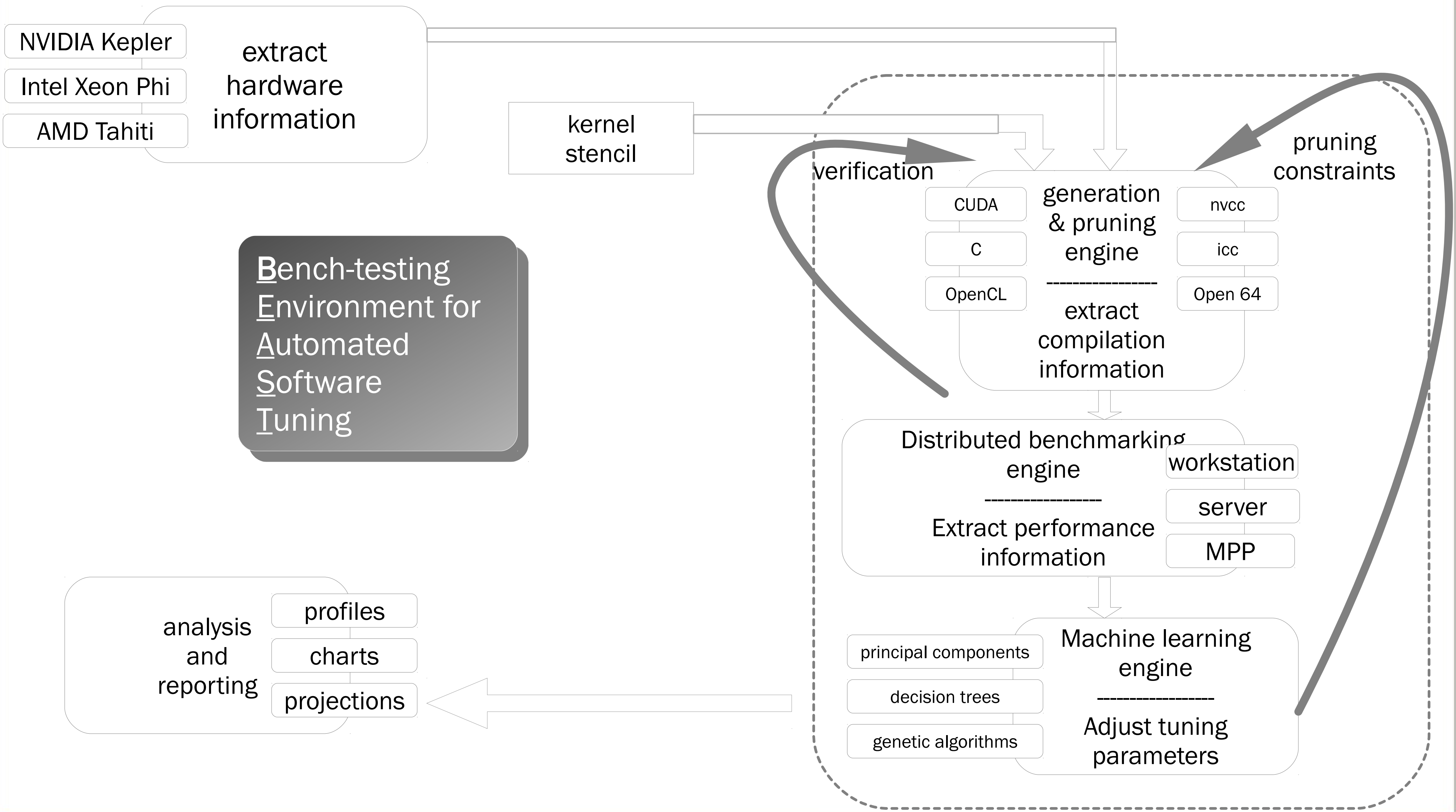
Optimizations: Example



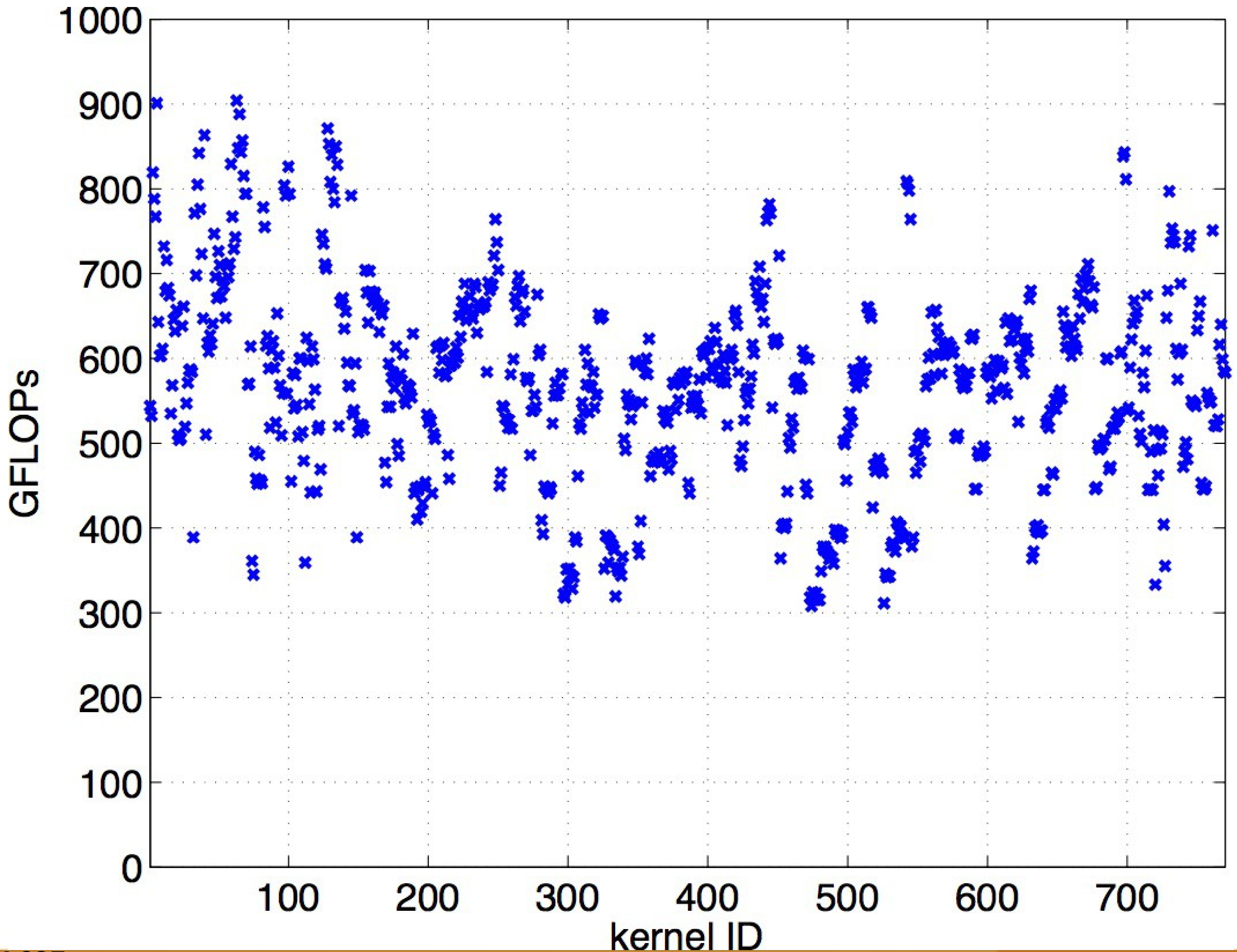
Generated Code: This if for the Compiler

```
for (dim_n = 32; dim_n < 1025; dim_n += 32)
      for (vector = 0; vector < 2; vector += 1)
        for (dim_m = 32; dim_m < 1025; dim_m += 32)
          for (blk_k = 16; blk_k < 64; blk_k += 16)
            for (blk_n = dim_n; blk_n < maxN + 1; blk_n += dim_n)
              for (blk_m = dim_m; blk_m < maxM + 1; blk_m += dim_m) {
                  blk_m_a_type_len = 1;
                  if (vector != 0)
                    blk_m_a_type_len = dim_vec;
                  blk_m_a_x = floor(blk_m / blk_m_a_type_len);
                  if (trans_a != 0)
                    blk_m_a_x = floor(blk_k / blk_m_a_type_len);
                  for (blk_m_a = blk_m_a_x; blk_m_a < 0; blk_m_a += -blk_m_a_type_len) {
                      blk_n_a_x = blk_k;
                      if (trans_a != 0)
                        blk_n_a_x = blk_m;
                      for (blk_n_a = blk_n_a_x; blk_n_a < 0; blk_n_a += -1) {
                          blk_n_b_x = blk_n;
                          if (trans_b != 0)
                            blk_n_b_x = blk_k;
                          for (blk_n_b = blk_n_b_x; blk_n_b < 0; blk_n_b += -1) {
                              blk_m_b_type_len = 1;
                              if (vector != 0)
                                blk_m_b_type_len = dim_vec;
                              blk_m_b_x = floor(blk_k / blk_m_b_type_len);
                              if (trans_b != 0)
                                blk_m_b_x = floor(blk_n / blk_m_b_type_len);
                              for (blk_m_b = blk_m_b_x; blk_m_b < 0; blk_m_b += -blk_m_b_type_len)
```

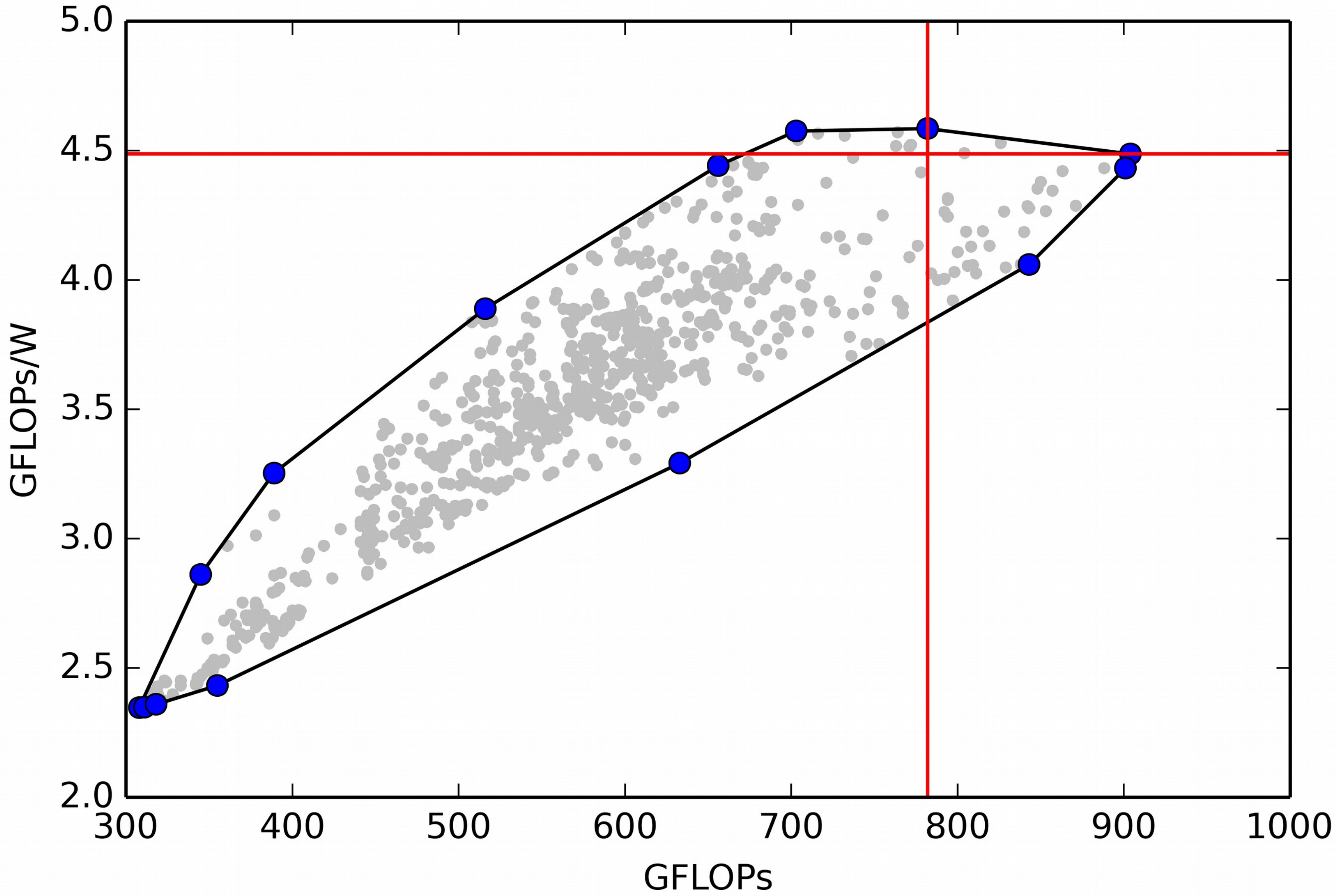
BEAST Design



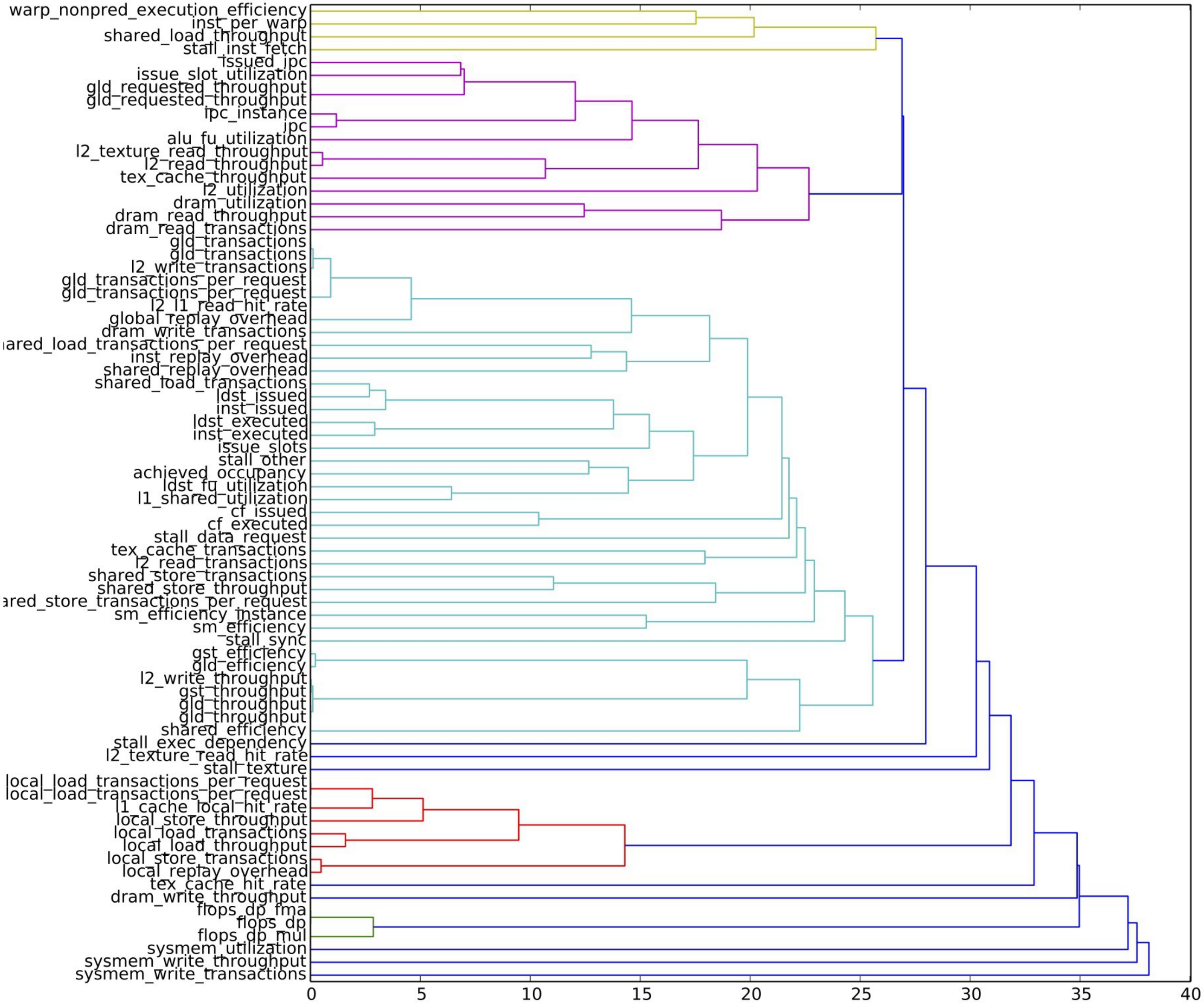
Performance: the Traditional View



Data Analysis: Convex Hull



Hierarchical Clustering of GPU Metrics



Future Work

- Apply autotuning to new kernels
- Continue work on parallel code compilation and autotuning
  - Multilevel parallelism: OpenMP and MPI
- Add new language features to the code generators
- Integration of the generated code with existing libraries