Strategy Selection in the Arm Performance Libraries

BLIS Retreat 2024

Joseph Dobson
27th September 2024

© 2024 Arm

arm PERFORMANCE LIBRARIES

Optimized BLAS, LAPACK and FFT for HPC applications

Best-in-class performance

Validated and maintained by Arm

Freely available

Arm provided 64-bit A-profile math libraries

- Support for industry-standard BLAS, LAPACK, RNG and FFTW interfaces
- Sparse linear algebra and batched BLAS support
- Optimized scalar and vector math.h routines

Best-in-class performance

- Tuned for latest Arm Architecture features and CPU designs
- Serial and parallel implementations

Broad compatibility with existing software

- Compatible with a wide range of commercial and open-source toolchains
- Support for C and Fortran users
- Versions for Linux, Windows and macOS

 © 2024 Arm

arm

CLAG Framework

Central **L**inear **A**lgebra **G**ateway

- A framework used to implement Arm PL's dense linear algebra routines
  - BLAS
  - LAPACK
- High level Modern C++
- Focus on high levels of code reuse and modularity
- The Framework has a model within which we can write solutions

Why not BLIS?

- At the inception BLIS was not stable on Arm
- Arm values having multiple solutions serving each market

© 2024 Arm

CLAG Model and Strategy Selection

- BLAS routines are Interfaces to computing **ProblemFamilies**
- Problems are generalized into **ProblemFamilies** and encoded into a **ProblemContext**

© 2024 Arm

Generalization of the problem space

matmul3 is the **ProblemContextBase** for

C=alphaAB + betaC

```
template<
    typename AMatrixType,
    typename BMatrixType,
    typename CMatrixType,
    typename ScalarType
>
struct matmul3 {
    using a_matrix_type = AMatrixType;
    using b_matrix_type = BMatrixType;
    using c_matrix_type = CMatrixType;
    using scalar_type   = ScalarType;


    a_matrix_type a;
    b_matrix_type b;
    c_matrix_type c;


    scalar_type   alpha;
    scalar_type   beta;


    matmul3(AMatrixType a_, BMatrixType b_, CMatrixType c_,
                ScalarType alpha_, ScalarType beta_)
    :   a     { std::move(a_) }
    ,   b     { std::move(b_) }
    ,   c     { std::move(c_) }
    ,   alpha { std::move(alpha_) }
    ,   beta  { std::move(beta_) }
    {   }
}; //struct matmul3
```

matmul3 generalization table

Routine Name	M	N	K	Alpha	Beta	A-Type	B-Type	C-Type
GEMM	M	N	K	Alpha	Beta	Gen	Gen	Gen
GEMV	M	1	N	Alpha	Beta	Gen	Gen	Gen
GER(B)	M	N	1	Alpha	Beta	Gen	Gen	Gen
AXP(B)Y	N	1	1	Alpha	Beta	Gen	Gen	Gen
DOT	1	1	N	1.0	1.0	Gen	Gen	Gen
SCAL	N	1	0	0.0	Alpha	Gen	Gen	Gen
COPY	N	1	1	1.0	0.0	Gen	Gen	Gen
SYMM	M	N	M	Alpha	Beta	Symm	Gen	Gen
SYMV	N	1	N	Alpha	Beta	Symm	Gen	Gen
HEMM	M	N	M	Alpha	Beta	Herm	Gen	Gen
HEMV	M	1	N	Alpha	Beta	Herm	Gen	Gen
SYRK	N	N	K	Alpha	Beta	Gen	Gen	Symm
SYR	N	N	1	Alpha	Beta	Gen	Gen	Symm
HERK	N	N	K	Alpha	Beta	Gen	Gen	Herm
HER	N	N	1	Alpha	Beta	Gen	Gen	Herm

© 2024 Arm

CLAG Model and Strategy Selection

- BLAS routines are Interfaces to computing **ProblemFamilies**
- Problems are generalized into **ProblemFamilies** and encoded into a **ProblemContext**
- **Strategies** are algorithm implementations used to compute problems
- **Strategies** can be constrained to solve a subset of a **ProblemFamily**

© 2024 Arm

Strategies

```
class outer_product {
public:
template<typename ProblemContext>
requires spec::has_get_spec<outer_product, ProblemContext>
bool operator() (const ProblemContext& pctx) const {
    using scalar_type = typename ProblemContext::scalar_type;

    if( !this->can_compute(pctx) ) return false;

    const auto spec = get_spec(spec::strategy_tag<outer_product>{}, pctx);

    const auto buf_size = min(pctx.a.strd(), spec.a_strd_block_size);

    scalar_type *buffer = is_strd_contig(pctx.a)
                        ? nullptr
                        : get_memory<scalar_type, memory_bank::level2>(buf_size * spec.max_threads);

    buffer_pool buffer_pool { buffer, spec.max_threads, buf_size };

    auto driver =
        parallelize            { general_parallel_strat, b_strd, spec.max_threads,
        resident               { a_matrix, 1, spec.a_strd_block_size, false,
        copy_matrix            { a_matrix, buffer_pool, general_strd_contig_generator{},
        outer_product_terminal { spec.kernel_axpby } } } };

    driver(pctx.a, pctx.b, pctx.c, compute_position{}, pctx.alpha, pctx.beta);

    if (buffer != nullptr) {
        return_memory<scalar_type, memory_bank::level2>(buffer);
    }

    return true;
}
```

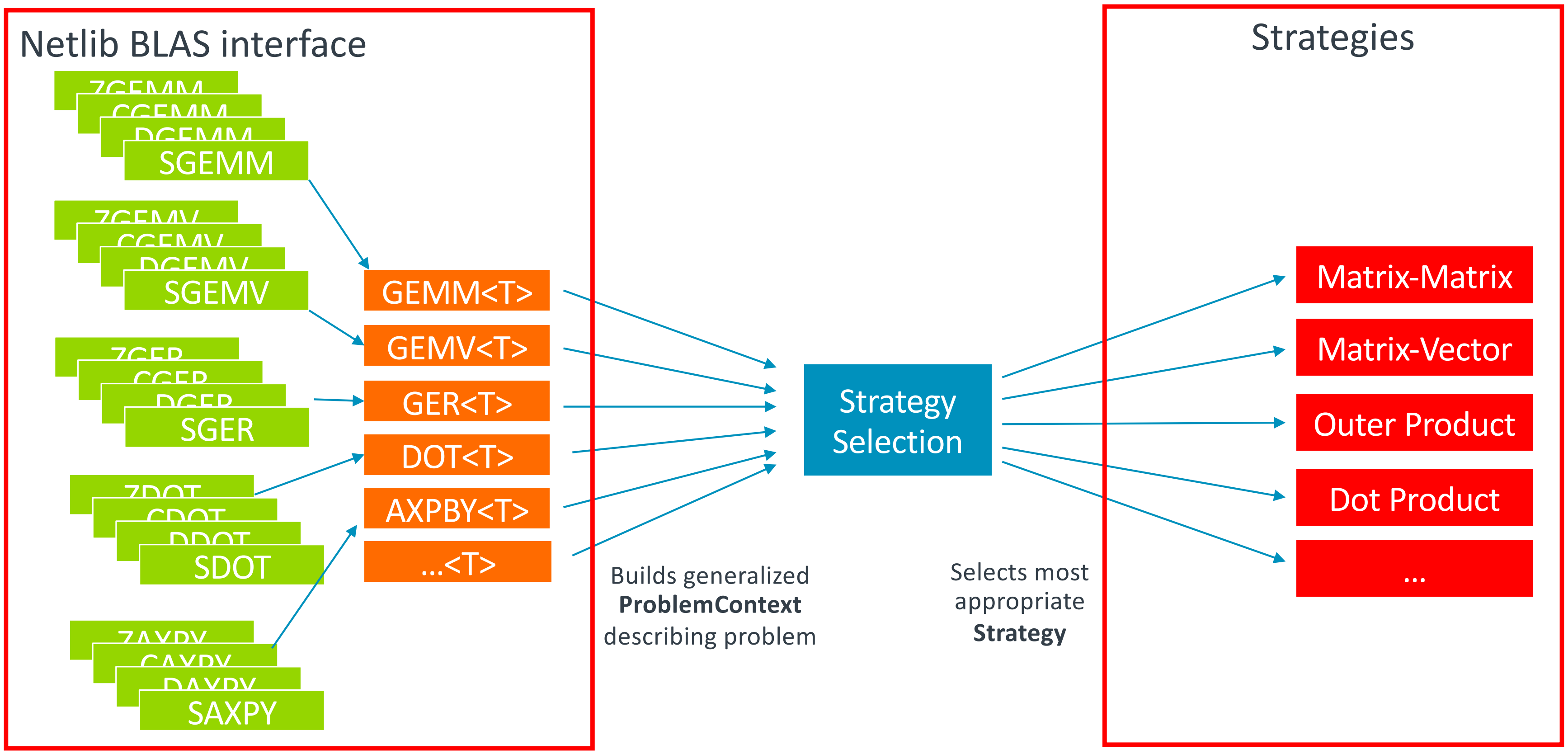
Implementation

```
    template<typename ProblemContext>
    ARMPL_CLAG_INLINE
    constexpr bool operator() (const ProblemContext&) const { return false; }

    template<typename ProblemContext>
    requires spec::has_get_spec<outer_product, ProblemContext>
    ARMPL_CLAG_INLINE
    constexpr bool can_compute(const ProblemContext& pctx) const {
        return pctx.alpha != zero<typename ProblemContext::scalar_type>
            && pctx.a.cntg() == 1 && pctx.b.cntg() == 1
            && pctx.c.cntg_step() == 1 && !pctx.a.is_conj();
    }

    template<typename ProblemContext>
    ARMPL_CLAG_INLINE
    constexpr bool can_compute(const ProblemContext&) const { return false; }
}; //class outer_product
```

Constraints

© 2024 Arm

Strategies



© 2024 Arm

CLAG Model and Strategy Selection

- BLAS routines are Interfaces to computing **ProblemFamilies**
- Problems are generalized into **ProblemFamilies** and encoded into a **ProblemContext**
- **Strategies** are algorithm implementations used to compute problems
- **Strategies** can be constrained to solve a subset of a **ProblemFamily**
- **Strategies** are tried in a preference order until one signals it has performed the computation, i.e. **StrategySelection**
- The strategy preference list is tuned for problem cases and on a per micro-architecture basis
- A strategy may not perform the computation if the **ProblemContext** does not meet its constraints
  - Ie. a matrix-matrix strategy may require the output matrix is col major, a GEMV interface may encode strides into the “M” dimension

© 2024 Arm

Strategy Selection

```
template<typename... T, typename ArchitectureSpec>
constexpr auto strategies<spec::problem_context<matmul::matmul3<T...>, ArchitectureSpec>> = std::tuple {
    matmul::set_or_scale { },
    matmul::compressed_general_matrix_vector { },
    matmul::symmetric_matrix_vector { },
    matmul::compressed_symmetric_matrix_vector { },
    matmul::compressed_rank_one_update { },
    matmul::out_of_place_matmul_left { },
    matmul::out_of_place_matmul_right { },
    matmul::atomic { },
    matmul::dot { },
    matmul::axpby { },
    matmul::gemv { },
    matmul::outer_product { },
    matmul::small { },
    matmul::basic { },
    matmul::sequential { },
    matmul::large { },
    matmul::large_no_sync { },
    matmul::rank_k_update_large { },
    matmul::rank_k_update_basic { },
    matmul::rank_one_update { },
    matmul::gemm_reference { },
    matmul::symm_hemm_l_reference { },
    matmul::symm_hemm_r_reference { },
    matmul::syrk_herk_reference { },
    matmul::backstop { }
};
```

StrategyList registers all of the strategies available

```
template<typename ProblemContext>
void compute(const ProblemContext& pctx) {

    const auto spec = get_spec(spec::strategy_selection_tag{}, pctx);


    for(const auto i : spec.strategy_preferences) {
        if( compute_index(strategies<ProblemContext>, pctx, i)) {
            return;
        }
    }
}
```

Strategies are evaluated in accordance with tuned **StrategyPreferences**

 © 2024 Arm

Generalizing Strategies

- In the opposite manner we Generalize Routines into **ProblemFamilies**
  - Matrix-Matrix -> Matrix-Vector (*n=1*)-> Vector-Vector (*n=1 & k=1*)
- We can also generalize our constrained **Strategies** to solve a wider range of problems.
- The generalized **Strategies** may now be considered for more cases.
  - They may never be used in this general cases, but the benefits of auto tuning is we needn't care
- In effect, you end up with non-packing-matmuls with different loop orderings

Example

- **AXPBY** is constrained by *n=1 & k=1*
  - loops over *n* -> outer-product Matrix-Vector
  - loops over *k* -> Matrix-Matrix
- **DOT** is constrained by *m=1 & n=1*
  - loops over *m* -> Matrix-Vector
    - i.e. **GEMV** *transa=T*
  - loops over *n* produces a Matrix-Matrix

© 2024 Arm

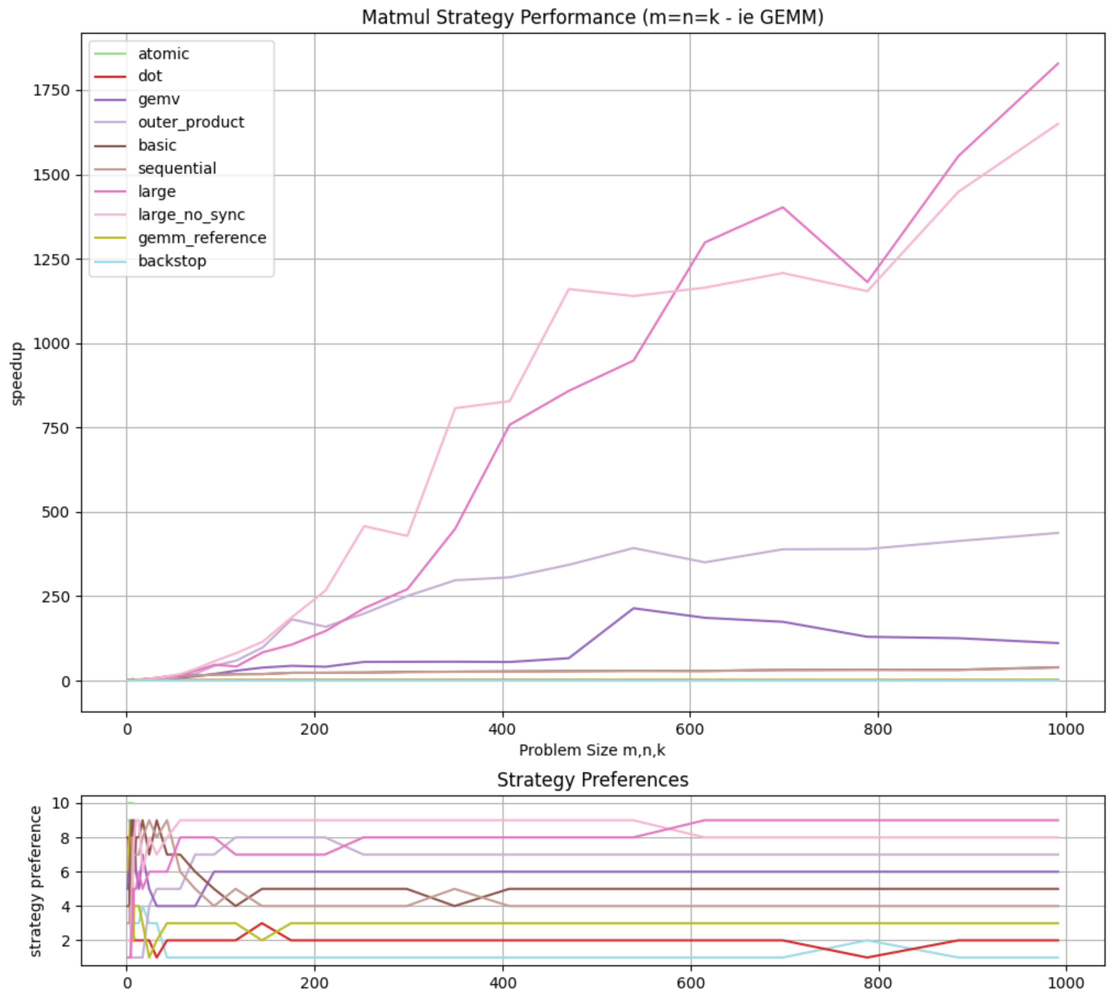
Tuning

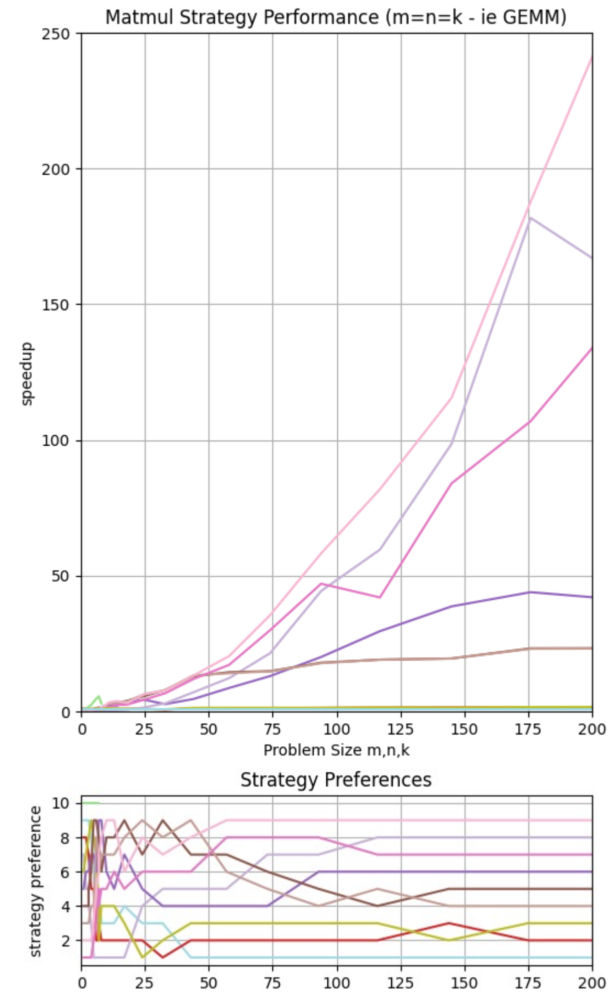
- Arm PL already uses a CI based auto-tuning system i.e.
  - Thread Throttling
  - L1 and L2 kernel selection
  - Not L3 kernels or block sizes
- This system is used to produce strategy preference list
- Good, not perfect!
  - Improving on static tuning
- Considerations:
  - Micro-arch
  - Problem family
  - Problem cases
  - Problem sizes

Example

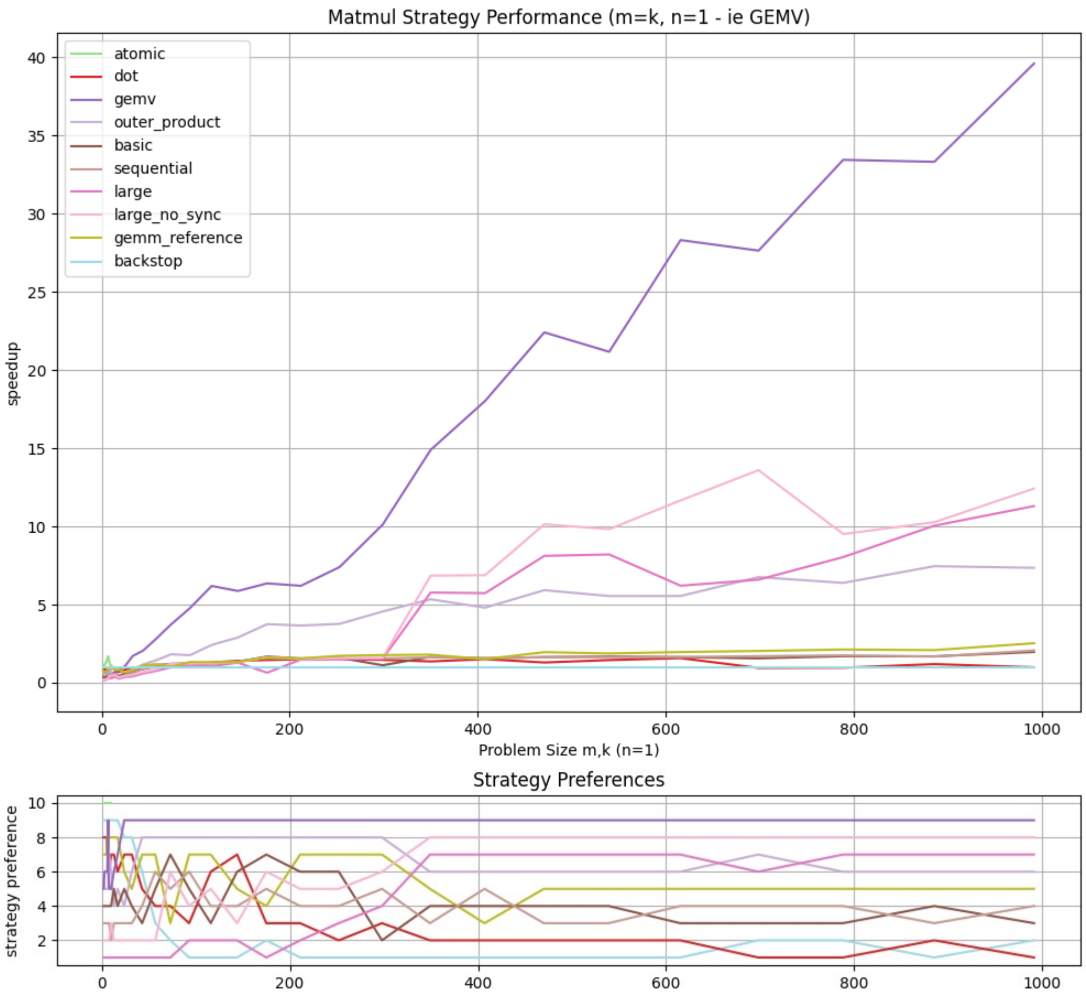
- **For each micro-architecture**
  - **For each problem family**
    - **For each problem case (constraints)**
      - **For each problem size**
        - **Benchmark every strategy**
        - **Rank order by best performing**
    - **Generate C++ strategy preference lists**

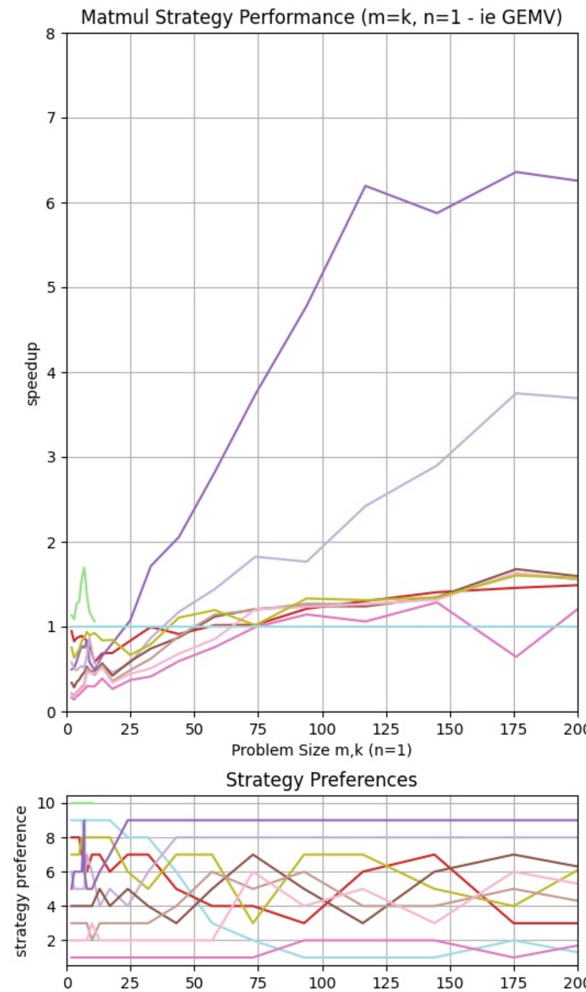
 © 2024 Arm


Matmul Strategy Performance (m=n=k - ie GEMM)
atomic
dot
gemv
outer_product
basic
sequential
large
large_no_sync
gemm_reference
backstop
speedup
Problem Size m,n,k
Strategy Preferences
strategy preference

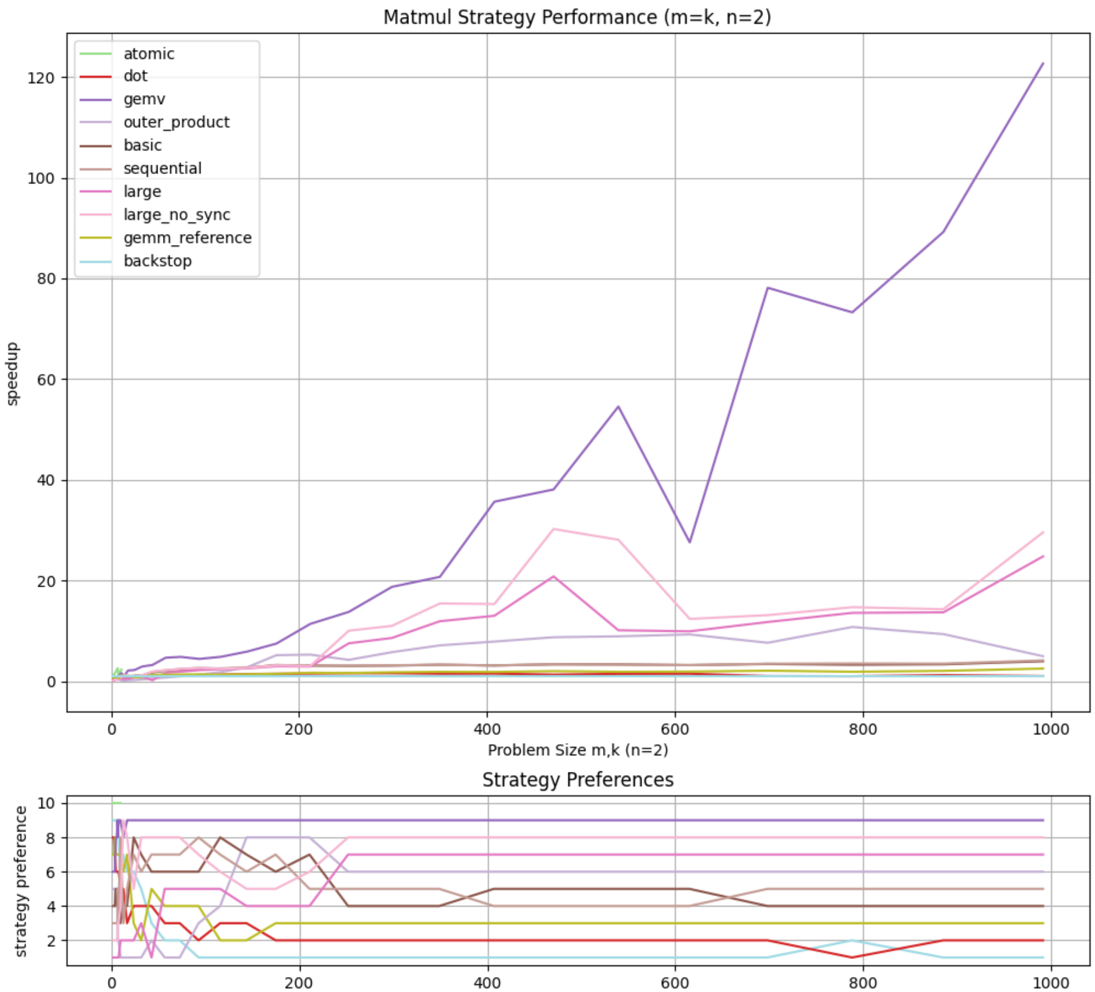

Matmul Strategy Performance (m=n=k - ie GEMM)
speedup
Problem Size m,n,k
Strategy Preferences
strategy preference

 © 2024 Arm

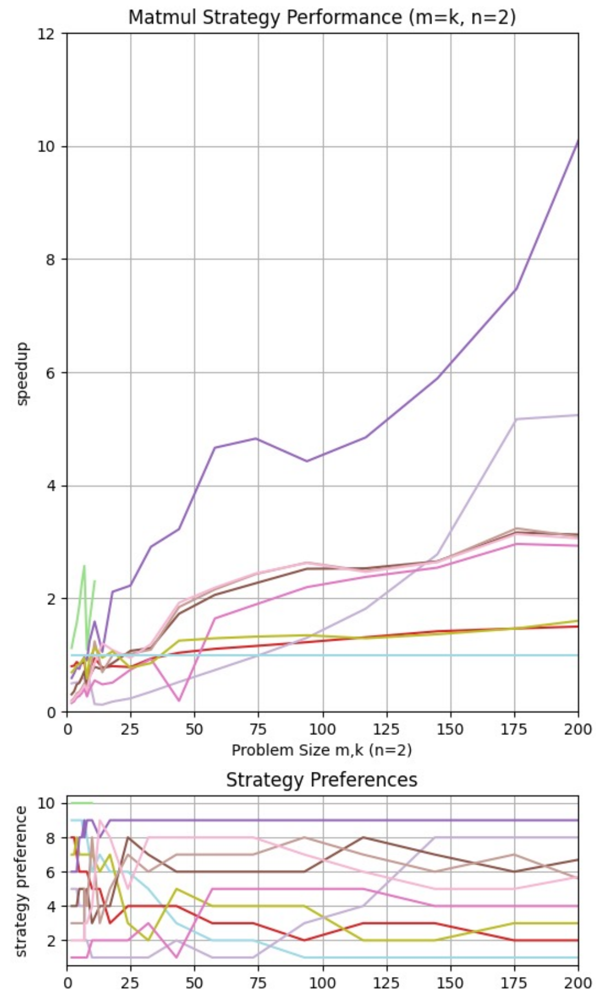

Matmul Strategy Performance (m=k, n=1 - ie GEMV)
atomic
dot
gemv
outer_product
basic
sequential
large
large_no_sync
gemm_reference
backstop
speedup
Problem Size m,k (n=1)
Strategy Preferences
strategy preference


Matmul Strategy Performance (m=k, n=1 - ie GEMV)
speedup
Problem Size m,k (n=1)
Strategy Preferences
strategy preference

 © 2024 Arm



Matmul Strategy Performance (m=k, n=2)
atomic
dot
gemv
outer_product
basic
sequential
large
large_no_sync
gemm_reference
backstop
speedup
Problem Size m,k (n=2)
Strategy Preferences
strategy preference



Matmul Strategy Performance (m=k, n=2)
speedup
Problem Size m,k (n=2)
Strategy Preferences
strategy preference

© 2024 Arm

Complications

- Strategies were originally written for Netlib interfaces
- Input matrices are assumed to be Column major matrices
  - When vector routine maps onto a matrix strategy you may end up with unexpected strides
    - Negative strides
    - C has column strides
- NaN propagation with application of Beta
  - SCAL propagates NaNs if beta is zero
  - GEMM does not
- Conjugate Transpose options vary between vector and matrix routines
- All these problems are surmountable
  - Further generalization of the strategies
  - More concise constraints
  - Increasing the **ProblemContext** problem space

© 2024 Arm

Conclusion

- **StrategySelection** is the final part of the CLAG model
- In short, it maps generalised Strategies onto generalised problems
- We can delegate that mapping to auto tuning
  - The mapping can take problem space parameters into consideration
  - This can cover more cases than the original interfaces specified
    - This is where the real performance gains really lie
- There are complications but fixing them improve the model

© 2024 Arm

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు

© 2024 Arm

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks

© 2024 Arm