# Task Scheduling Strategies for Batched Basic Linear Algebra Subprograms on Many-core CPUs[1)]

Daichi Mukunoki[1] Yusuke Hirota[2] Toshiyuki Imamura[1]

[1]RIKEN Center for Computational Science (R-CCS)
daichi.mukunoki@riken.jp

[2]University of Fukui

Dec. 22, 2021
14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021)

[1)]This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #19H04127 and the FLAGSHIP 2020 project. This research used computational resources of the Wisteria/BDEC-01 system at the University of Tokyo and the supercomputer Fugaku at R-CCS.

**Batched BLAS**[2)]

- Interface for computing multiple independent operations with different parameters in a single routine for a given routine in BLAS
- Exploit many-cores effectively by simultaneously computing multiple small problems for which sufficient parallelism cannot be extracted with classic BLAS
- Standard specification and reference implementation have been released in 2018[3)], and now available in major vendor BLAS implementations
- Applications: block algorithms, divide-and-conquer algorithms, H-matrix computation, deep-learning codes, etc.

---

[2)] J. Dongarra et al., The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems, ICCS 2017, 2017.

[3)] https://github.com/NLAFET/BBLAS

# Introduction (2/2)

**Implementation of batched BLAS**

- Many studies on GPUs – with stream (early days), kernel implementations (current mainstream)
- Few studies on CPUs – vendor implementations are black boxes
  – some implementations have insufficient performance (MKL's batched DGEMV) or no batched routines (OpenBLAS, XBLAS, etc.)

**Our study: efficient implementation of batched BLAS on CPUs**[4)]

- Implementation of batched routines using sequential BLAS routines with OpenMP (with automatic generation)
- Proposal of task scheduling based on apriori cost estimation for load balancing
- Comparison of different task scheduling strategies

---

[4)] A part of it has been already presented at ISC2018 poster: Y. Hirota et al., Automatic Generation of Full-Set Batched BLAS, ISC 2018 research poster session, 2018.

# Batched BLAS Interface (Standard)

```
void blas_dgemm_batch (
    const int group_count, const int *group_size,
    const bblas_enum_t layout,
    const bblas_enum_t *transa,
    const bblas_enum_t *transb,
    const int *m, const int *n, const int *k,
    const double *alpha,
    const double **a, const int *lda,
    const double **b, const int *ldb,
    const double *beta,
    double **c, const int *ldc, int *info);
```

Figure: Batched DGEMM Interface

- All parameters (except *layout*, which determines row/column-major) can be specified independently for each batch
- One can group multiple batches that are executed with the same parameters – but this study ignores this feature: assuming that all batches have the same or different parameters

# Batched Routine using Sequential BLAS Routine with OpenMP

```
#pragma omp parallel for private(i,j) schedule(TYPE, CHUNK)
for(i = 0; i < group_count; i++){
    for(j = 0; j < group_size[i]; j++){
        blas_routine (param1[i], param2[i], addr1[i*group_size+j], ...);
    }
}
```

Figure: Implementation of batched routine using sequential routine with OpenMP

– With OpenMP, each batch is executed by a sequential BLAS routine on each thread. (this study assumes one thread/core)

# Batched Routine using Sequential BLAS Routine with OpenMP

```
#pragma omp parallel for private(i,j) schedule(TYPE, CHUNK)
for(i = 0; i < group_count; i++){
    for(j = 0; j < group_size[i]; j++){
        blas_routine (param1[i], param2[i], addr1[i*group_size+j], ...);
    }
}
```

Figure: Implementation of batched routine using sequential routine with OpenMP

**"Schedule (TYPE, CHUNK)"** clause in parallel for:

- **TYPE** specifies the scheduling method among "*static*", "*dynamic*", "*guided*", "*auto*", and "*runtime*"
- **CHUNK** specifies the *chunk size*, which is a unit of task allocation to a thread.

# Task Scheduling Methods Implemented in OpenMP[7)]

When # of tasks is $n$ and # of threads is $p$,

- **Static**: allocates chunks in units of *chunk size* in a round-robin fashion in the thread number order. If *chunk size* is not specified, the default is approx. $n/p$[5)]
- **Dynamic**: allocates chunks to the thread that has finished executing a chunk one by one. If chunk size is not specified, it becomes 1
- **Guided**: similar to *dynamic*, but the *chunk size* decreases from approx. $n/p$ to the specified *chunk size* (default is 1)[6)]
- (*Auto*: automatically determined by the compiler)
- (*Runtime*: specified through environmental variable)

---

5) The handling of non-divisible cases is left to the implementation.

6) The handling of non-divisible cases and the decreasing step are left to the implementation.

7) For details, see OpenMP Architecture Review Board, OpenMP Application Programming Interface, v5.1, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

# Scheduling Based on Apriori Cost Estimation (1/2)

Load imbalance occurs when the problem size of each batch varies.

- OpenMP's scheduling methods do not reorder the tasks as they assume that the load of tasks are unknown in advance.
- However, in batched BLAS, the load of tasks (batches) can be estimated in advance, and can be reordered accordingly.

– We propose a task scheduling method based on apriori cost estimation with reordering for solving load imbalance in batched BLAS

# Scheduling Based on Apriori Cost Estimation (2/2)

**Apriori-cost scheduling (our proposal)**

1. Compute the computational cost (as Flops) of each batch for all batches
2. Sort them by QuickSort[8)]
3. Starting with the highest cost, allocate them to the thread with the lowest total cost on the allocated batches
4. Repeat 3 until there are no more batches left to allocate

– Not always achieve the optimal load balance – as this is a greedy algorithm, and the estimated cost does not always represent the actual cost (runtime) – but achieve better load balancing than OpenMP's methods

[8)] The sorting cost for $n$ batches is $O(n\mathtt{log}(n))$ on average and $O(n^2)$ in the worst case.

# Comparison of Scheduling Methods

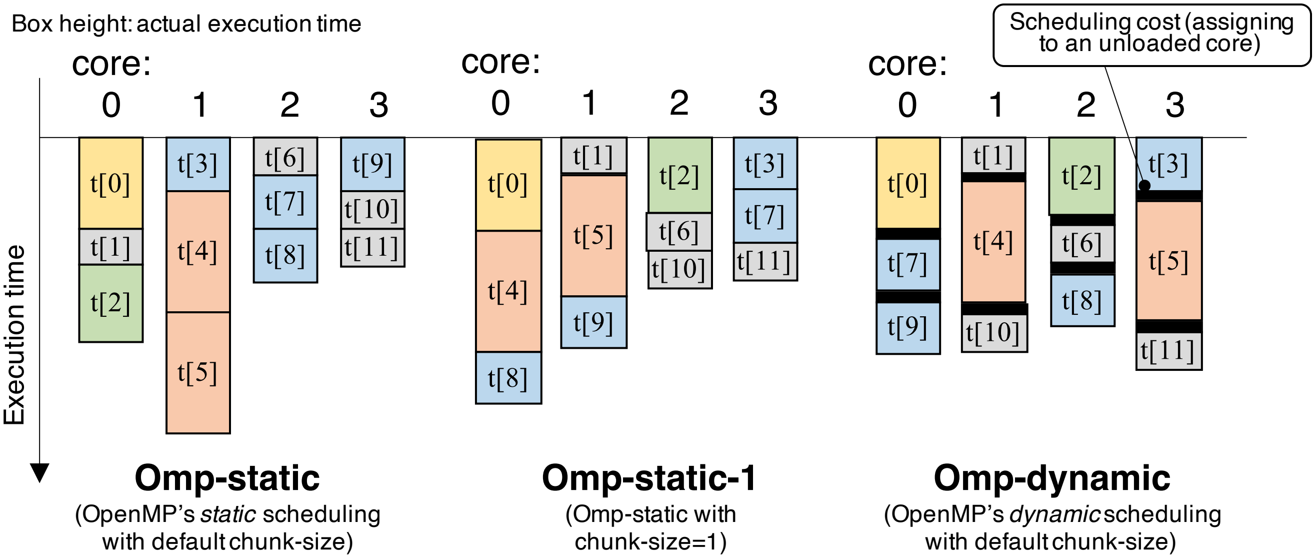



Figure: Scheduling strategies (when # of batches (tasks) = 12, and # of cores = 4)

# Comparison of Scheduling Methods

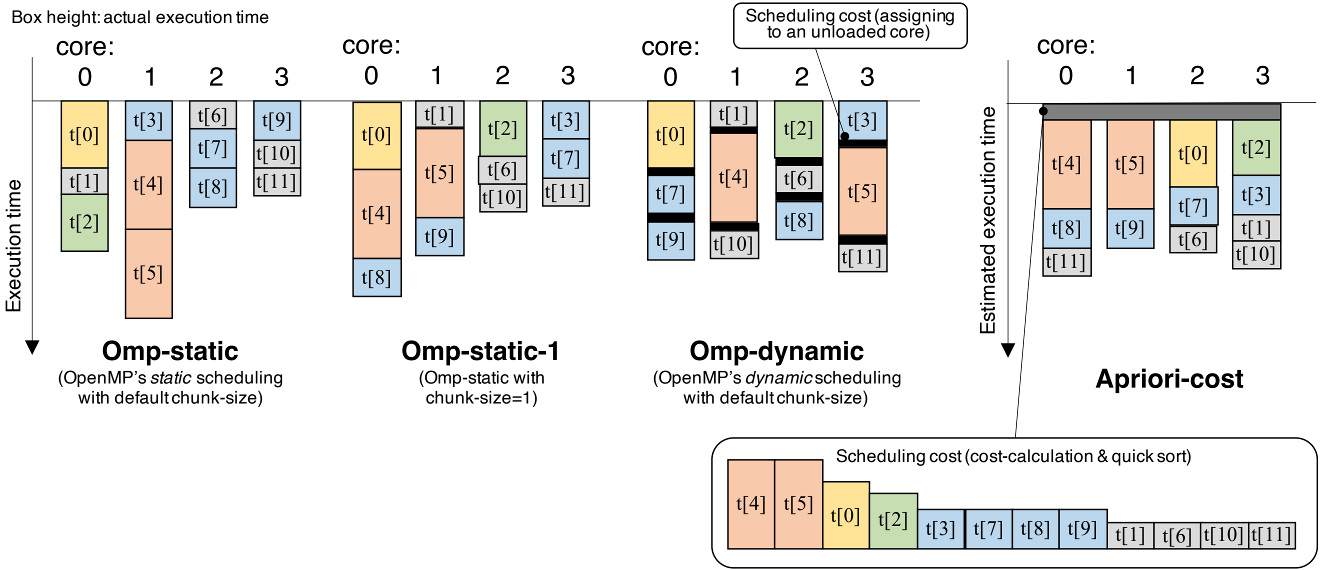



Figure: Scheduling strategies (when # of batches (tasks) = 12, and # of cores = 4)

# Comparison of Scheduling Methods

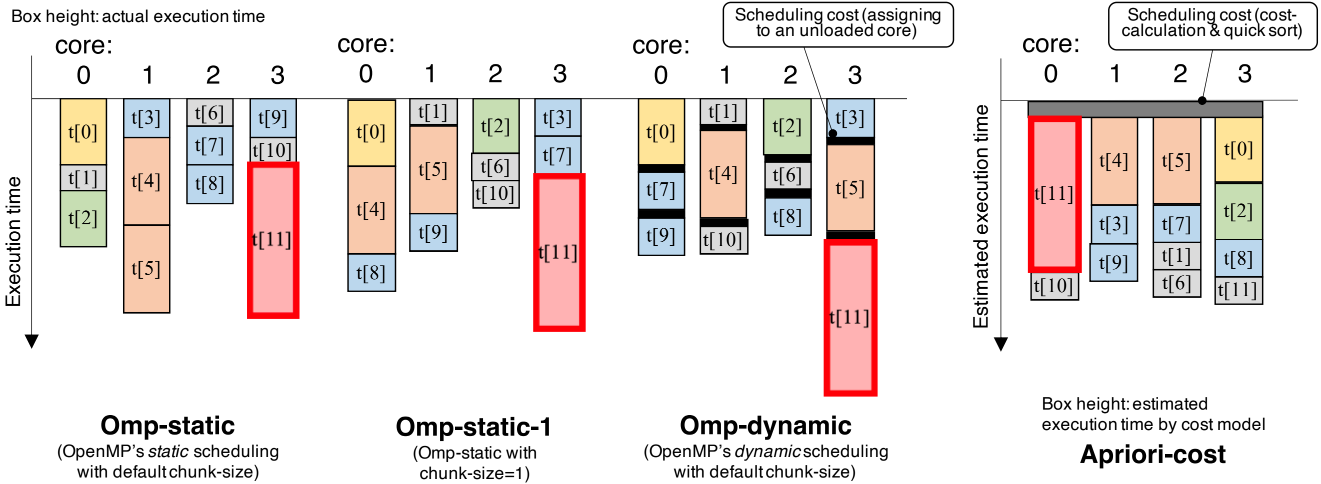



Figure: Scheduling strategies (when # of batches (tasks) = 12, and # of cores = 4)

– When t[11] has an extremely long runtime, OpenMP's scheduling methods cannot solve the load imbalance, but *apriori-cost* could improve it to some extent

# Performance Evaluation (1/3)

**Environments**

- **Armv8.2-A+SSL2**
  - Fugaku (R-CCS), A64FX (Armv8.2-A, only 12 cores in 1 NUMA node are used), fccpx 4.5.0 (-Kfast,ocl,openmp), lang/tcsds-1.2.31, Fujitsu SSL2
- **Zen2+OpenBLAS**
  - AMD Ryzen Threadripper 3990X (Zen2, 64 cores), GCC 8.3.1 (-O3 -fopenmp), OpenBLAS 0.3.15
- **IceLake+MKL**
  - Wisteria-A (U. Tokyo), Intel Xeon Platinum 8360Y (IceLake, 36 cores, numactl -localalloc), ICC 2021.2.0 (-O3 -xHost -qopenmp), MKL 2021.2.0.
- **IceLake+XBLAS**
  - The same hardware as IceLake+MKL, GCC 8.3.1 (-O3 -fopenmp), XBLAS[9] 1.0.248

[9] Extra Precise Basic Linear Algebra Subroutines, https://www.netlib.org/xblas/

# Performance Evaluation (2/3)

**Implementations for comparison**

- **Non-batch**: multi-threaded BLAS routine is executed one-by-one for each batch
- **Apriori-cost**: *apriori-cost* scheduling (proposal)
- **Omp-static**: OpenMP's *static* scheduling
- **Omp-static-1**: OpenMP's *static* scheduling with chunk-size=1
- **Omp-dynamic**: OpenMP's *dynamic* scheduling
- **Omp-guided**: OpenMP's *guided* scheduling
- **MKL-batch**: MKL's batched routines

– Except for *non-batch* and *MKL-batch*, batches are executed using sequential BLAS routine with OpenMP

# Performance Evaluation (3/3)

**Target routines**[10)]

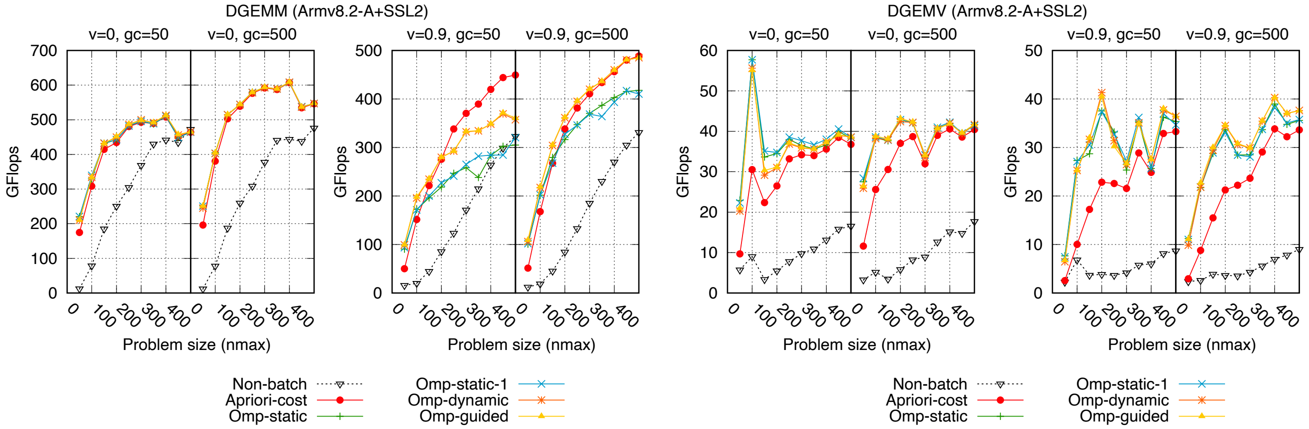
- DGEMM (matrix-matrix multiplication)
- DGEMV (matrix-vector multiplication)

**Conditions**

- # of batches = group count (gc), $gc = 50$ or $gc = 500$
- Problem size $n =$randn(ceil$((1 - v)n_{max})$), where $n_{max}$ is the maximum size, and $v = 0$ (all the same size) or $v = 0.9$ (size varies)
- Parameters: leading dimension= $n_{max}$, non-transposed, incx=incy=1
- Problems (operands) are placed in memory in the order of the batches, no overlapped references
- Average 10 executions after two dummy runs

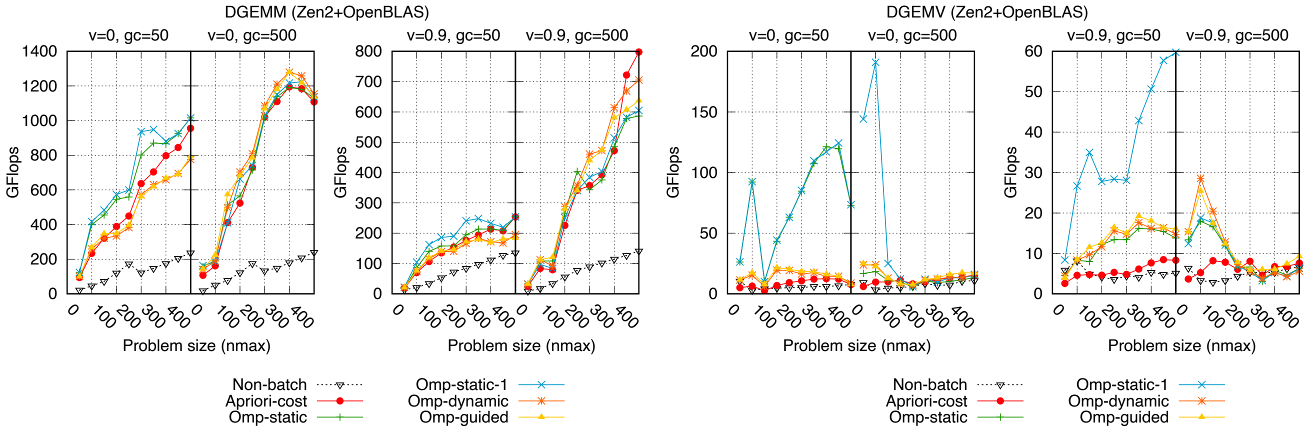
10) Only XBLAS routines compute double-precision inputs with quadruple-precision (106-bit) arithmetic

# Result (Armv8.2-A+SSL2)



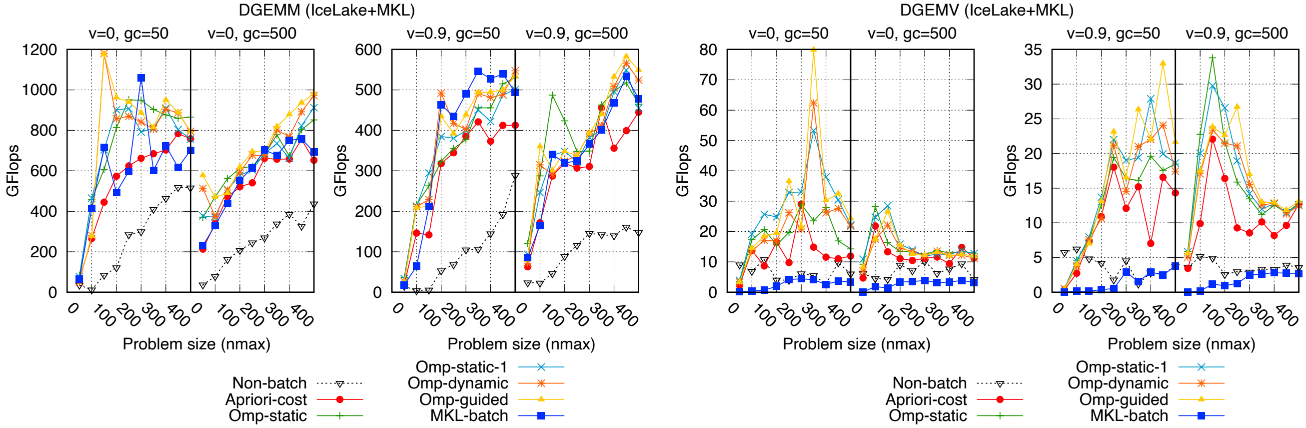

- Dynamic methods are certainly effective for solving load imbalancing, but as # of batches increases, the load imbalance is naturally resolved
- *Apriori-cost* solves load imbalance more powerfully, but the scheduling (sorting) cost degrades the performance when the batch load is light (e.g., GEMV, small size GEMM)

# Result (Zen2+OpenBLAS)



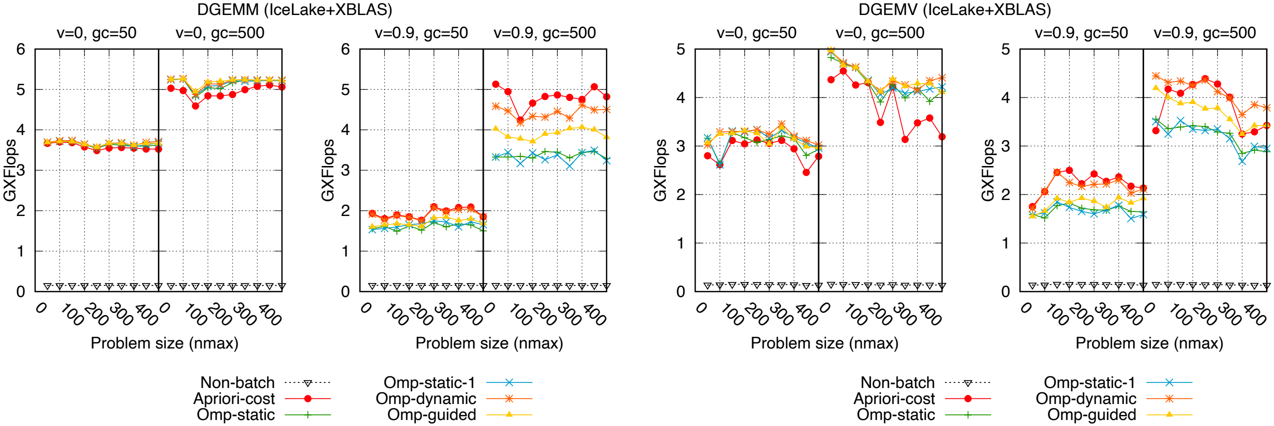

- *Omp-static-1* is often the best when the total load is small – because data locality is preserved and thus cache works effectively

# Result (IceLake+MKL)




- The implementation with existing non-batched routines and OpenMP can achieve comparable performance to MKL's batched routines
- MKL's batched GEMV may not be parallelized

# Result (IceLake+XBLAS)




- XBLAS, which performs extended-precision operation[11)], takes a much longer runtime than other BLAS
- This is on the same *IceLake* environment, but unlike MKL, XBLAS shows clear (and almost expected) differences in the scheduling method

[11)] 106-bit quadruple-precision operation based on double-precision arithmetic, like the double-double arithmetic

# From Experimental Results

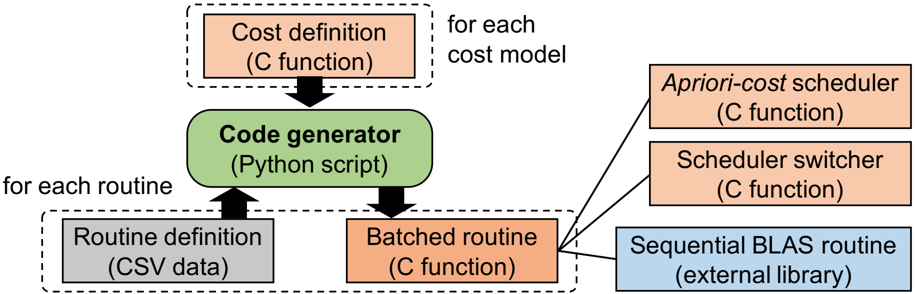
**Observations**

- Batched routine implemented using non-batched sequential BLAS and OpenMP can achieve comparable (or better) performance with MKL's batched routine
- *Apriori-cost* scheduling solves load imbalance better than OpenMP's ones, but its scheduling cost degrades the performance when batches' load is light
- Improving cache hit is important when batches' load is light
- Optimal scheduling strategy is problem and environment dependent

**Automatic selection of scheduling method?**

- Automatic selection is not easy because the cost of scanning input becomes a bottleneck
- Scheduling method should be selectable (as dedicated routine for each scheduling method, or through routine's argument)

# Automatic Generation of Batched BLAS from Non-batched BLAS




**Automatic generator** (Python script) works with

- Routine definition (CSV file) – for each BLAS routine
- Cost definition (C function) – for each cost model
- *Apriori-cost* scheduler (C function)
- Scheduler switcher (C function)
- Sequential BLAS implementation (MKL, OpenBLAS, etc.)

# Conclusion

**Summary**

- Simple and efficient batched BLAS implementation (with automatic generation) on CPUs – using non-batched sequential BLAS with OpenMP
- A task scheduling based on apriori-cost estimation for resolving load imbalance
- Optimal scheduling is highly input and environment dependent – scheduling method should be selectable in batched BLAS

Our automatic batched BLAS generator is available at
`https://www.r-ccs.riken.jp/labs/lpnctrt/projects/batchedblas`
(the current version only adopt *apriori-cost* without scheduler switcher)