Dynamic Thread Coarsening for CPU and GPU OpenMP Code

Ivan R. Ivanov
Institute of Science Tokyo
Tokyo, Japan
RIKEN Center for Computational Science
Kobe, Japan
ivanov.i.e641@m.isct.ac.jp

Jens Domke
RIKEN Center for Computational Science
Kobe, Japan
jens.domke@riken.jp

Toshio Endo
Institute of Science Tokyo
Tokyo, Japan
endo@scrc.iir.isct.ac.jp

Johannes Doerfert
Lawrence Livermore National Laboratory
Livermore, California, USA
doerfert1@llnl.gov

Abstract

Thread coarsening is a well known optimization technique for GPUs. It enables instruction-level parallelism, reduces redundant computation, and can provide better memory access patterns. However, the presence of divergent control flow - cases where uniformity of branch conditions among threads cannot be proven at compile time - diminishes its effectiveness. In this work, we implement multi-level thread coarsening for CPU and GPU OpenMP code, by implementing a generic thread coarsening transformation on LLVM IR. We introduce dynamic convergence - a new technique that generates both coarsened and non-coarsened versions of divergent regions in the code and allows for the uniformity check to happen at runtime instead of compile time. We performed evalution on HecBench for GPU and LULESH for CPU. We found that best case speedup without dynamic convergence was 4.6% for GPUs and 2.9% for CPUs, while our approach achieved 7.5% for GPUs and 4.3% for CPUs.

ACM Reference Format:
Ivan R. Ivanov, Jens Domke, Toshio Endo, and Johannes Doerfert. 2025. Dynamic Thread Coarsening for CPU and GPU OpenMP Code. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767482

1 Introduction

Modern high-performance computing increasingly relies on heterogeneous systems, however, there are still large CPU-only systems in use such as Fugaku [1].

OpenMP has evolved from being a programming model purely for CPU multithreading, to also offer unified support for GPU offloading. It is the choice of programming model for many applications being run on supercomputing systems.

OpenMP provides many directives to help programmers optimize their code. For example, automatic vectorization, loop unrolling, tiling, fusion, interchange, etc. Some of these types of transformations are extremely tedious to implement manually and severely decrease the maintainability of the code if they are applied manually.

One such example, and one which OpenMP does not provide, is thread coarsening. Thread coarsening merges the work of multiple threads into one while interleaving their instructions. This allows interleaving of inherently independent sequences of instructions, which can better exploit instruction level parallelism and can reduce redundant computation.

It is a well established optimization for GPUs, and similar optimizations also exist for CPUs, such as unroll-and-jam and chunking in OpenMP. While existing automated approaches for thread coarsening exist, to the best of our knowledge, none is available for OpenMP. In addition all existing implementations suffer from the inability to handle divergent code. When the branching behavior of the parallel region does not match between different threads, code cannot be safely interleaved as original threads must be able to branch independently. This prevents profitable coarsening when compiler thread uniformity analysis was not strong enough, or some behavior of the code or input data was not available to the compiler.

In this work, we enable thread coarsening for OpenMP for both CPU and GPU C/C++ code. We apply it to both the `omp teams distribute` and `omp parallel for` constructs via a transformation on LLVM IR in the clang compiler. To address earlier work's limitations with potential thread divergence, we introduce *dynamic convergence* - a strategy to allow for generating a coarsened version of code behind potentially divergent branches that can be used if the runtime uniformity conditions allow for it. This approach allows us to coarsen codes that traditional approaches would have conservatively disabled coarsening for.

This work makes the following contributions.

- A generic thread coarsening transformation on LLVM IR for both CPUs and GPUs that supports multi-level OpenMP parallelism (`teams` and `parallel`).
- The introduction of *dynamic convergence* - a technique to enable thread coarsening for potentially divergent code using runtime uniformity checks.

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767482

```
parallel (i = 0 to 16) {
  A(i);
  B(i);
}
```
Original

```
parallel (i = 0 to 8) {
  A(i * 2);
  A(i * 2 + 1);
  B(i * 2);
  B(i * 2 + 1);
}
```
Coarsened

Figure 1: Thread coarsening of a parallel loop with a factor of 2. It is achieved by partially unrolling the loop and interleaving operations from different original "iterations" (shown in different colors).

- OpenMP extension clauses in clang to enable easy use of the transformation in source code.

The paper is structured as follows. We will first introduce the reader to the thread coarsening transformation in Section 2 along with its drawbacks. We will describe our improvements to it in Section 3, before detailing how we apply it to OpenMP in Section 4. We will finally evaluate our improvements in Section 5 on both CPU and GPU.

2 Thread Coarsening Background

Thread coarsening is an optimization that was first introduced for GPUs [12]. The first publications explored the optimization using manual source code rewriting. Since then, many compiler approaches have appeared to perform the transformation automatically or to decide on the factor automatically. It has been applied to OpenACC[7, 8], OpenCL[2], and CUDA[4].

Thread coarsening combines the work of multiple parallel iterations of a loop in a single iteration while interleaving instructions to exploit instruction level parallelism and reduce redundant computation.

We will use C-like pseudocode to illustrate the transformation. Fig. 1 shows a very simple example. On the left is the original code before the transformation. `parallel` indicates a *parallel loop* which indicates that the *iterations* of the loop can be executed in any order and in parallel. In this example, we coarsen with a factor of 2. This means that we consolidate two original iterations in one new one. As we can see on the right of Fig. 1, the number of iterations changed from 16 to 8 to account for that, and each new iteration does the work of two original ones (two `A` and two `B` calls). Also, we are allowed to freely interleave the code from different iterations (as long as `A`, `B` from the same original iteration still appear in the same order).

The number of original iterations we consolidate is called the coarsening *factor*.

2.1 Divergent Control Flow Limitations of Thread Coarsening

One shortcoming of the thread coarsening optimization is that it cannot handle cases where the code contains divergent control flow.

We will use Fig. 2 to illustrate the issue. This example contains a parallel loop which we want to coarsen, and a conditional branch `if (test(i))`. Suppose that `test(i)` evaluates differently for iterations we try to interleave. A branch whose condition may evaluate differently is called a *divergent* branch, and one which evaluates the same is called *uniform*. Then, if we try to interleave `B` and `C` in one basic block as in Fig. 2 (b) [1], we would either execute `B` and `C` for an original iteration when they should not have been if we take the `true` branch, or miss executing `B` and `C` for the other iteration if we take the `false` branch. Thus, in this case, we cannot interleave the contents of the `if` statement and must evaluate the condition and branch off differently for each original iteration. This is achieved by considering the entire `if` statement as a single unit for the purpose of interleaving. In this case, the code after the transformation looks like Fig. 2 (c).

```
for (int i = 0 to 16) {
  A(i);
  if (test(i)) {
    B(i);
    C(i);
  }
}
```
(a) Original code

```
parallel (i = 0 to 8) {
  A(i * 2);
  A(i * 2 + 1);
  if (test(i * 2)) {
    B(i * 2);
    B(i * 2 + 1);
    C(i * 2);
    C(i * 2 + 1);
  }
}
```
(b) If we can prove test(i) is uniform

```
parallel (i = 0 to 8) {
  A(i * 2);
  A(i * 2 + 1);
  if (test(i * 2)) {
    B(i * 2);
    C(i * 2);
  }
  if (test(i * 2 + 1)) {
    B(i * 2 + 1);
    C(i * 2 + 1);
  }
}
```
(c) If we must assume test(i) is divergent

Figure 2: We may not be able to interleave everything as the `if (test(i))` branch may evaluate differently depending on the original iteration number `i`. The inability to interleave `B` and `C` is a problem shared by all existing automatic coarsening techniques to the best of our knowledge.

In the case we cannot prove that `test(i)` will evaluate the same for interleaved iterations, we must conservatively generate the same code for the divergent case (Fig. 2 (c)) as that results in correct behaviour for both divergent and uniform branches.

It is often the case that the important computation that would benefit from vectorization or coarsening is nested in control flow, and any potential divergence there would prevent coarsening.

The inability to interleave `B` and `C` is a problem shared by all existing automatic coarsening techniques to the best of our knowledge.

3 Our Approach

We implement coarsening as a loop transformation in LLVM IR and use it for CPU and GPU OpenMP code. We introduce a *dynamic convergence* notion to alleviate the limitations of thread coarsening for code with potentially divergent control flow.

3.1 Coarsening with *Dynamic Convergence*

To alleviate the issue with the inability to handle divergent code (see Section 2.1), we introduce a new approach. In short, we generate both an optimistic interleaved and a pessimistic non-interleaved

[1] A basic block is a sequence of instructions which does not have any branches and will thus execute all instrutions contained.

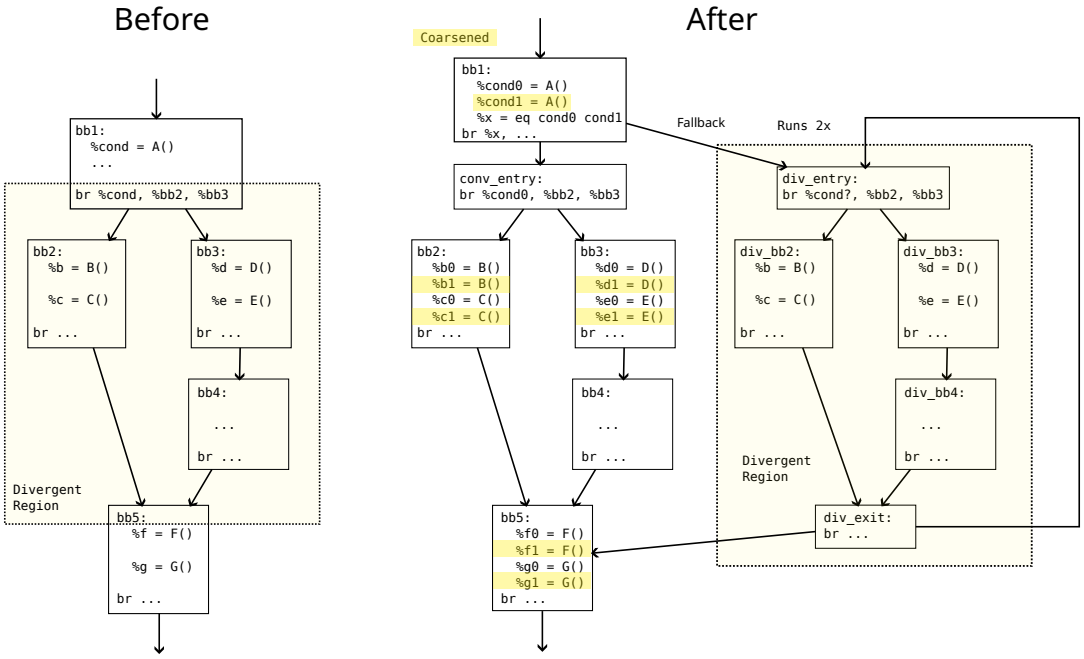

Figure 3: Coarsening code in a loop body with control flow with statically unknown uniformity. Both coarsened and fallback non-coarsened version are generated and which to execute is chosen at runtime.

version for code after potentially divergent control flow. Then, at runtime we can choose to use the interleaved version if the all combined iterations agree on the condition. Otherwise, we fall back to the non-interleaved version. We call this *dynamic convergence*. A similar technique has previously been explored in the context of vectorization [9]. The overview of our LLVM IR transformation is shown in Fig. 3.

We are now going to detail how our transformation algorithm works.

3.1.1 Iteration Space Partitioning. First, we find the canonical induction variable of the loop. This is the induction variable that is incremented by a loop invariant increment until it the loop exit condition defined by a comparison of the upper bound to it is met. This allows us to compute the total number of iterations the loop will go through. This, in general, is only known at runtime. Let us denote the number of iterations with N and the coarsening factor with f.

We then partition the original loop iteration space in groups of f. The number of iterations of the new coarsened loop will be $\lfloor N/f \rfloor$, and it will handle the original iterations i where $0 \leq i < f \times \lfloor N/f \rfloor$ For the remaining iterations $N \mod f$, we generate an epilogue loop which is a clone of the original one.

Now, we need to generate the contents of the coarsened loop, where we need to do the work of f original iterations per new iteration.

3.1.2 Coarsened Loop Execution State. There are two states which the execution of an iteration in the coarsened loop can be in - we are either in a converged or diverged state. The converged state is when the original iterations *agree* on which basic block they are executing, and can thus execute a coarsened basic block (see the left *Coarsened* side of the *After* state in Fig. 3). The diverged state is when a previous branch condition differed between the set of original iterations, and they must now execute non-coarsened basic blocks independently. (see the right *Fallback* side of the *After* state in Fig. 3).

When we start executing an iteration in the coarsened loop, because of the way we partitioned the iteration space to go in groups of f, all original iterations start executing the same loop entry block, which means we start in a converged state. This means that we can always use a coarsened block for the entry block to the coarsened loop (see bb1 in Fig. 3) .

After a uniform branch we can continue executing coarsened code, however, after encountering a divergent branch we need to let the original iterations branch independently and start executing a non-coarsened version of the loop. It would not be incorrect to continue executing the original iterations independently until they finish their own body loop, however, there is a question of whether it is possible to make them re-converge and start executing the coarsened version again.

To address this issue, we introduce the concept of *divergent regions*.

3.1.3 Divergent Regions. We first generate the optimistic loop structure which contains only coarsened basic blocks (the *Coarsened* side of the *After* state in Fig. 3) - this assumes all branches are uniform. Now we need to insert the runtime checks for divergence.

Our algorithm first analyzes all branches $b \in B$ in the body of the loop and groups them in two. The first group is the provably uniform branches B_U, which will evaluate the same for all

```
#pragma omp teams distribute
for (int i = 0; i < n; i++) {
  #pragma omp parallel for
  for (int j = 0; j < m; j++) {
    CODE(i, j);
  }
}
```

Figure 4: OpenMP provides two levels of parallelism which can be used to execute loops in parallel. Our transformation is able to coarsen both of these levels.

iterations of the loop. The second group is the (potentially) divergent branches B_D which we may or may not evaluate the same for different original iterations.

The uniform branches in B_U do not need any special handling as when in a converged state, the original iterations agree on the next block and can continue executing the coarsened loop in a converged state. For each branch in B_D, we insert a runtime check which tells us whether the original iterations agreed on the branch condition (`%x = eq cond0 cond1` in Fig. 3). If they do, they remain in the converged state and continue executing the coarsened code. If they do not, they need to diverge and execute independently. We call the region of the code they must execute independently the *divergent region* of the branch. Each branch $b \in B_D$ has its own divergent region which we will note as $DR(b)$.

To determine where the independently executing original iterations will converge again we use post-dominator tree analysis in LLVM. This gives us information about what is the "earliest" basic block *guaranteed* to be executed after a given basic block. I.e. this analysis gives us the earliest point at which the original iterations (which started executing independently after encountering a runtime divergent branch branch in B_D) will meet again and can start executing in a converged state. Thus, the the divergent region of branch b - $DR(b)$ is the set of blocks reachable from the basic block that b is in, without passing through the post-dominator of b - $PD(b)$.[2]

This gives us a guarantee when we execute the original iterations independently, they will all arrive at $PD(b)$ once they exit $DR(b)$. This means, that at that point, we can re-converge them and start executing in a convergent state again. This allows us to recover in the middle of the iteration and not give up after the first divergence.

This is illustrated in Fig. 3. The identified divergent region is highlighted in yellow on the left side. This then gets transformed into a fallback region which gets executed sequentially by the original iterations, entering through `div_entry`, and re-converging at `bb5`, which is the post-dominator of `bb1`. While the execution state can change into divergent mode at any branch b in B_D, after that it needs to execute the entirety of the associated $DR(b)$ until it re-converges.

[2]The existence of a post-dominator *in* the loop body itself is given to us by the fact that we work on loops in a canonical form which are guaranteed to have a single loop latch block which will either exit the loop, or start another iteration. Thus, all loop iterations finish execution at the latch block, which means that the latch block always post-dominates all other blocks in the loop.

3.2 Characteristics of Coarsening with Dynamic Convergence

An important characteristic of dynamic convergence is that the performance can be very input-dependent.

While generally coarsening with a higher factor (for example to fully utilize the vector-width of a CPU) can be more beneficial in the optimal case where all divergent branches turn out to be runtime-uniform, it can have adverse effects due to the code size and complexity increase. In addition, under an assumption of random distributions of the branch condition results, it is increasingly unlikely to find a sequence of f matching conditions (making the branch runtime-uniform) as the factor f grows.

An important advantage to traditional coarsening is that it can enable vectorization in parallel loops with long bodies that are guarded by a potentially divergent conditional which prevents coarsening.

Drawbacks. Dynamic convergence can also have some drawbacks.

One of the biggest drawbacks is the explosion in code size it can result in. Traditional thread coarsening results in roughly a $(f+1)\times$ code size increase, where f is the factor. This comes from the fact that the coarsened version of the loop will contain roughly $f\times$ more instructions, however, generally, we also need to generate an epilogue loop to iterate through the remaining iterations which were not divisible by the factor. When dynamic convergence is enabled, fallback code for each divergent region needs to be introduced. These divergent regions can be nested in each other which results in more code duplication.

Another drawback is the increased computational overhead for the uniformity checks and the additional frequent branching.

Profitable cases. We expect that there are some cases where the profitability of dynamic convergence is apparent and heuristics can be implemented to detect them and selectively enable them. For example, if there is a large amount of uniform code (straight-line or with uniform branch conditions) that is hidden behind a single potentially divergent branch, the benefit we can get from coarsening that region can greatly outweigh the overhead of introducing some additional comparisons and branching. Further, for CPUs, if interleaving of code hidden behind a divergent branch would enable profitable vectorization, it is even more likely to be profitable. We leave exploration in this space to future work.

4 OpenMP Thread Coarsening

First, we will go through how the parallel OpenMP constructs are implemented in clang, before detailing how we apply our transformation.

4.1 Background to Parallel Constructs in OpenMP

OpenMP contains constructs for both task and structured loop parallelism. In this work we are specifically going to look into the structured loop parallelism constructs since that is where thread coarsening can be easily applied.

For loop parallelism, OpenMP provides two levels of parallelism - teams and threads, where teams are composed of threads. Teams are

```
parallel (teamId = 0 to teamNum) {
  parallel (threadId = 0 to threadNum) {
    // The code at this level becomes the outlined device kernel code for GPUs.
    for (int i : range(teamId * itemsPerTeam, (teamId + 1) * itemsPerTeam)) {
      for (int j : range(threadId * itemsPerThread, (threadId + 1) * itemsPerThread)) {
        // Optional chunk loop can get introduced here
        CODE(i, j);
      }
    }
  }
}
```

Figure 5: By default, OpenMP distributes the work of the loops to be processed by the parallel workers of each level in equal portions. In our work, we leave the thread launch configuration (number of teams and threads) the same, and coarsen the two inner distribution for loops.

```
#pragma omp teams distribute ompx_coarsen_distribute(3)
for (int i = 0; i < n; i++) {
  #pragma omp parallel for ompx_coarsen_for(2)
  for (int i = 0; i < n; i++) {
    ...
  }
}
```

Figure 6: The user can easily annotate the OpenMP loop they want to coarsen. Because we support dynamic coarsening we can coarsen both the `teams distribute` and `parallel for` loops. This roughly corresponds to "block" and "thread" coarsening explored in earlier work. [4, 10, 11]

launched by the `omp teams` directive, while threads are launched by the `omp parallel` directive. These two levels of parallelism can be used to execute the iterations of for loops in parallel using the `omp distribute` and `omp for` respectively. The launch of the parallelism and the parallel execution of the loops using the just launched parallel resources can be combined in one directive - `omp teams distribute` and `omp parallel for` respectively as shown in Fig. 4.

In the default static scheduling scheme, the compiler then decomposes these loops so that each team and thread gets an equal amount of work (unless the number of work items is not divisible by the number of workers, when some workers get fewer). This scheduling is shown in Fig. 5. On GPUs these two levels of parallelism get mapped to the blocks (workgroups) and threads, while on CPUs the behavior usually depends on compiler and runtime options, however, OpenMP code written for CPUs usually only uses the `omp parallel` level.

These outer two parallel loops are implicit in the IR. They do not materialized, and are instead handled by the OpenMP runtime. The region nested in the two parallel loops get materialized in the IR and can be optimized as normal.

4.2 Applying our Transformation to OpenMP in Clang

We apply our coarsening transformation to the two inner `for` loops in the decomposed loop nest in Fig. 5. We assume that the programmer does not use any knowledge about how the scheduling is done internally and thus parallel execution of the individual `CODE(i, j)` in the inner two loops is allowed. This way, we do not need to alter the thread launch configuration (number of teams and threads).

OpenMP also provides a chunked schedule, which introduces one more loop at the innermost level. In that case, since the OpenMP standard guarantees execution of the items in the chunk in order, we do not coarsen the chunk loop.

Multi-level coarsening. As stated above, we enable coarsening of multiple levels of parallelism, which is not a very well explored domain. Block level coarsening (coarsening the *outer* level of parallelism) was first proposed by Unkule et al.[11] and later independently evaluated by Stawinoga and Field [10]. Earlier work explored combining both levels of coarsening [4] and showed that there are cases where coarsening both parallel dimensions achieved the best performance. We also bring the benefits of this to OpenMP.

4.3 Multi-level OpenMP Coarsening Extension in Clang

Since we do not have an automated approach to deciding the appropriate coarsening factor, we provide the programmer with easy to use OpenMP clauses which can be specified at the `omp teams distribute` and `omp parallel for` directives - `ompx_coarsen_distribute(f)` and `ompx_coarsen_for(f)` respectively, where `f` is the coarsening factor. Their usage is illustrated in Fig. 6. Currently dynamic convergence can only be toggled using a compiler command line option or an environmental variable, however, in the future, it can be added as an additional parameter to the clauses.

5 Evaluation

We evaluate our approach on both CPU and GPU OpenMP offloading C/C++ codes. Since our aim is to provide programmers with easy to use transformations which can be specified in source code, we gather measurements for a suite of different configurations of our transformation.

In general we have two parameters which can vary for coarsening a given parallel loop - the coarsening factor (see Section 2) and whether we use dynamic convergence (see Section 3.1).

5.1 GPU

Benchmarks. For our evaluation on GPUs, we used the HeCBench [5] benchmarking suite. It contains 314 benchmarks for OpenMP offloading, we obtained measurements for roughly 350 kernels.

Systems. We use two systems to evaluate our approach on GPUs. Our first GPU system has an AMD MI60 GPU, and the second one an AMD MI210. These GPUs were released in 2018 and 2022 and

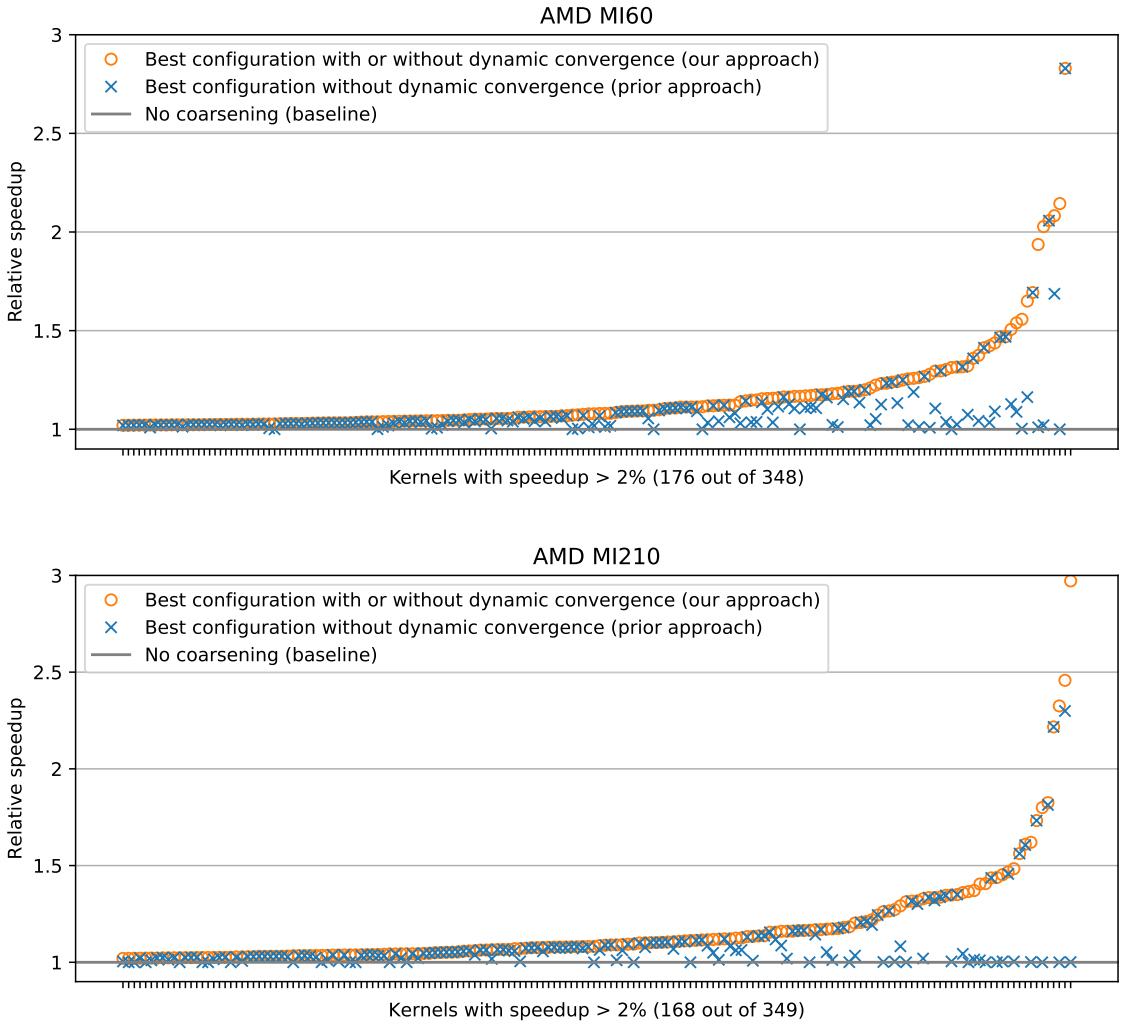

Figure 7: Autotuned best performing coarsening configuration of HeCBench OpenMP kernels on AMD MI60 and AMD MI210 GPUs. In many cases dynamic convergence was required to reach the best performance.

we hope this can show how our approach offers improvements for a wide range of GPUs.

Coarsening Configurations. Typical OpenMP GPU offloading code makes use of both the `teams` and `parallel` levels of parallelism (see Section 4.1). Therefore, we need to evaluate coarsening both levels and how the coarsening factors of both levels may interact. We have two levels of loops which can both independently have different factors and have dynamic convergence disabled or enabled which makes the possible configuration space very large. For this reason, we limit dynamic convergence to be on or off for both loops at the same time, and we experimented with factors in $\{2, 4, 8\}$. Our entire configuration matrix is $\{\text{off}, 2, 4, 8\} \times \{\text{off}, 2, 4, 8\} \times \{\text{on}, \text{off}\}$, which are the `teams` factors, `parallel` factors, and whether dynamic convergence is on or off. The configurations (off, off, $*$) are duplicates,[3] so we use only one of them. This gives us a total of 31 configurations per kernel.

Collecting Runtime Data. The runtime responsible for launching OpenMP GPU kernels in LLVM has support for generating profiling data for each kernel launch.[4] We use this feature to obtain the wall-clock runtime of individual kernel launches and aggregate the runtimes of the same kernel across three executions of each benchmark.

Best Configuration Performance. We compared the best-performing configuration with and without dynamic convergence to the baseline performance where coarsening is turned off. Then, we took either the transformed version if the best one outperformed the

[3]Whether dynamic convergence is on or off has no effect when coarsening is off.
[4]Enabled by the environmental variable `LIBOMPTARGET_PROFILE`.

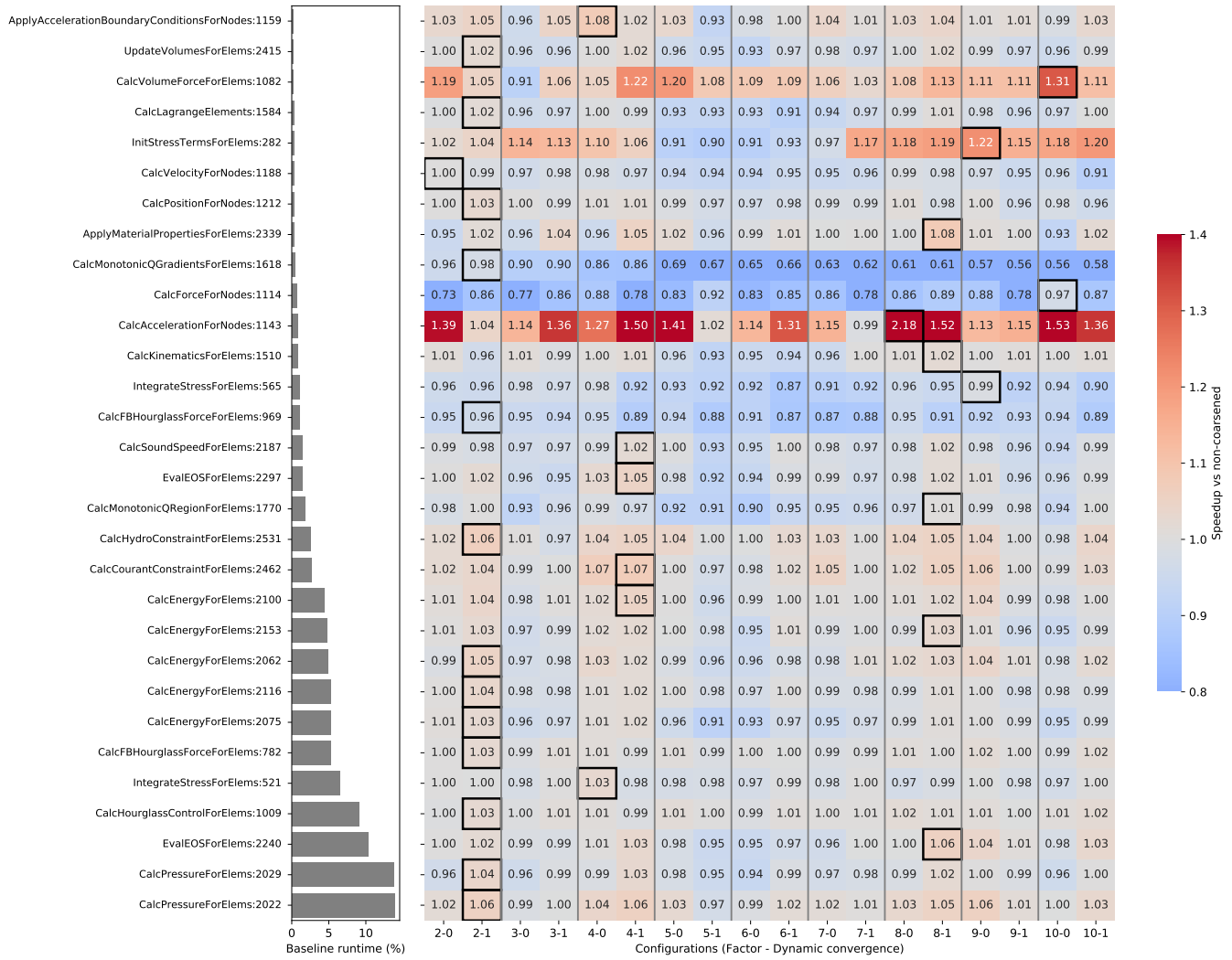

Figure 8: Evaluation of our thread coarsening transformation for different configurations (factor - dynamic convergence) on the kernels in LULESH. We show the speedup of each configuration compared to the baseline (no coarsening) configuration. The best performing configuration for each kernel (row) is outlined in black.

baseline, or the baseline version. This way, we attempt to summarize what the effect of this optimization would be if the user picked the best possible configuration (including the baseline non-transformed one).

We computed the speedup relative to the baseline, and plotted the results for each kernel in Fig. 7. Since many benchmarks did not show a substantial improvement, we limited the plots to only kernels where we achieved a $> 2\%$ speedup. We can observe how in many cases dynamic convergence was required to obtain the peak performance, and the gap in performance for many kernels is very large.

We also calculated the overall geomean effect of the user having the ability the use dynamic convergence. Without it, when picking the best possible configuration, we achieved a speedup 4.8% for MI60 and 4.8% for MI210. When we enable the dynamic coarsening configurations, the overall speedup grows to 7.5% and 7.8% respectively.

5.2 CPU

Benchmarks. For our evaluation on CPUs, we used LULESH [6] with its default invocation, which let us measure 30 OpenMP parallel regions (which we will call kernels).

System. Our CPU system has an EPYC 7713 CPU, with 256GB of RAM.

Coarsening Configurations. LULESH (as is typical for CPU-only OpenMP codes) only uses the `parallel` level in OpenMP (see Section 4.1). This means that we only have one choice to make for the factor we coarsen with for each kernel. We also have a choice

to turn dynamic convergence on or off. Since the transformation configuration space is much smaller than the GPU case, we can experiment with more coarsening factors in this case.

We choose to coarsen with factors in $\{i : 2 \leq i \leq 10\}$ and with dynamic convergence on and off for each factor. Together with the case where coarsening is off, we run the benchmark in 19 total configurations.

Collecting Runtime Data. To collect runtime data, we measured the wall-clock time it took for each of the parallel regions to execute. For this purpose, we patched LLVM's OpenMP runtime to measure CPU parallel regions' runtimes and dump them to a file for later analysis. In this case, we ran the benchmark 20 times in each configuration, and summed up the runtimes of each kernel across each execution, and took the median across the 20 benchmark runs.

Results. We compute the speedup of each configuration with regard to the baseline with coarsening turned off and we plot the results in Fig. 8.

We also computed the overall speedup that can be obtained for the parallel kernel portions of the entire LULESH application if we picked the best performing configuration for each kernel separately. This is achieved by summing up the runtimes for the per-kernel best-performance configurations and comparing that against the sum of the runtimes in the baseline configuration. Doing this we achieved a 2.9% speedup if we do not use any dynamic convergence configurations and 4.3% with dynamic convergence enabled.

The majority of the best performing configurations used dynamic convergence, with most of them at coarsening factors 2, 4, and 8.

6 Future Outlook

In the future, we would like to explore strategies for automatic coarsening factor decision. Input generation [3] can be used to train models to predict optimal factors. However, as previously discussed in Section 3.2 the effects of dynamic convergence can be very input-dependent, thus integrating a solution for recording and replaying kernels for fast autotuning can make the use of this transformation easier and more effective.

We believe that thread coarsening with dynamic convergence has the potential to greatly benefit from profile-guided optimizations. Profiling can be used to check in advance how often divergent branches are actually uniform for a given factor. There are two possible optimizations that can make use of this information.

Firstly, in cases where it is rare that a branch is uniform, we can skip generating the optimistic coarsened version and only generate the fallback loop where each original iteration's divergent region executes sequentially. This can alleviate the code size issue of dynamic convergence (see Section 3.2)

Secondly, when the runtime divergence of a branch is exceedingly rare, while we cannot skip generating the fallback version, as it would be illegal (see Section 2), we can make use of code layout optimizations which make sure the very rarely executed code will not take up instruction cache space which can be used for hot code.

7 Conclusion

We implemented a general thread coarsening transformation on LLVM IR. We give programmers the ability to apply thread coarsening on OpenMP `teams distribute` and `parallel for` constructs using new extension clauses in clang. To overcome a limitation existing thread coarsening implementations share - inability to handle divergent control flow - we introduced *dynamic convergence*, which can at runtime choose a coarsened version of the code subject to a uniformity check.

Our evaluation on AMD MI60 and MI210 GPUs and on an AMD EPYC 7713 CPU, showed how dynamic convergence allowed us to achieve aggregated speedups of 4.3% on CPU and 7.5% on GPU compared to 2.9% and 4.6% without, respectively.

Acknowledgments

This work was supported by JST SPRING, Japan Grant Number JPMJSP2180 and the RIKEN Junior Research Associate Program.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2010264). This manuscript has been partially co-authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. The United States Government retains, and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

ChatGPT was utilized to help draft the abstract, introduction, and conclusion sections of the paper.

References

[1] [n. d.]. *TOP500.* https://www.top500.org/

[2] Prithayan Barua, Jun Shirako, and Vivek Sarkar. 2018. Cost-Driven Thread Coarsening for GPU Kernels. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. doi:10.1145/3243176.3243196

[3] Ivan R. Ivanov, Joachim Meyer, Aiden Grossman, William S. Moses, and Johannes Doerfert. 2024. Input-Gen: Guided Generation of Stateful Inputs for Testing, Tuning, and Training. arXiv:2406.08843 [cs.SE] https://arxiv.org/abs/2406.08843

[4] Ivan R. Ivanov, Oleksandr Zinenko, Jens Domke, Toshio Endo, and William S. Moses. 2024. Retargeting and Respecializing GPU Workloads for Performance Portability. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 119–132. doi:10.1109/CGO57630.2024.10444828

[5] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. doi:10.1109/ISPASS57527.2023.00041

[6] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[7] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic Optimization of Thread-Coarsening for Graphics Processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. doi:10.1145/2628071.2628087

[8] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. doi:10.1145/2503210.2503268

[9] Jaewook Shin. 2007. Introducing Control Flow into Vectorized Code. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 280–291. doi:10.1109/PACT.2007.4336219

[10] Nicolai Stawinoga and Tony Field. 2018. Predictable Thread Coarsening. *ACM Trans. Archit. Code Optim.* 15, 2, Article 23 (6 2018), 26 pages. doi:10.1145/3194242

[11] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. 2012. Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality. 21–40. doi:10.1007/978-3-642-28652-0_2

[12] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html