Proceedings of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 671–676
DOI: 10.15439/2024F6469
ISSN 2300-5963 ACSIS, Vol. 39

Automatic Generation of OpenCL Code through Polyhedral Compilation with LLM

Marek Palkowski, Mateusz Gruzewski
West Pomeranian University of Technology in Szczecin
ul. Zolnierska 49, 71-210 Szczecin, Poland
Email: mpalkowski@zut.edu.pl

***Abstract*—In recent years, a multitude of AI solutions has emerged to facilitate code generation, commonly known as Language Model-based Programming (LLM). These tools empower programmers to automate their work. Automatic programming also falls within the domain of optimizing compilers, primarily based on the polyhedral model, which processes loop nests concentrating most computations. This article focuses on harnessing LLM tools to generate OpenCL code for non-serial polyadic dynamic programming kernels.[1] We have chosen the Nussinov RNA folding computational task, previously employed to test polyhedral compilers in optimizing kernels with non-uniform dependences. The code generated in OpenMP by polyhedral optimizers is limited to CPU computations. We automatically convert it into the OpenCL standard using ChatGPT-3.5 through its source-to-source queries to extend the number of possible platforms. The validity and efficiency of the generated code were verified on various CPUs and GPUs from different manufacturers.**

I. Introduction

CODE generation using LLM (Language Model-bas ed Programming) tools has garnered significant interest among programmers and researchers in recent times. This represents a novel form of automatic programming. Generating code effortlessly is also within the domain of polyhedral compilers. Source-to-source techniques enable the generation of parallel and localized code through stages of syntax and loop dependency analysis, transformations, and loop generation. Thus, it is evident that leveraging LLM models for High-Performance Computing (HPC) opens up new possibilities.

Using the capabilities of the ChatGPT tool, we aim to automatically generate OpenCL code [2] based on existing OpenMP code [3] optimized with polyhedral tools. This allows us to execute the code on any graphics card that supports OpenCL. The test code will be based on the Nussinov algorithm [4] for RNA prediction with program loop nets containing many non-uniform loops and posing a challenge for polyhedral tools.

The Nussinov algorithm is a non-serial polyadic dynamic programming (NPDP) kernel, used to assess the efficiency of tiled code generated by advanced optimizing compilers [5], [6], [7], [8]. NPDP dependence patterns, the most complex category of Dynamic Programming (DP), exhibit non-uniform dependences characterized by irregularities and expressed using affine expressions. Dynamic Programming involves finding optimal solutions for simpler instances of a problem and extending them to solve larger instances. Additional difficulty arises from parallelism with synchronization. The goal for GPT is to generate a sequential loop spawning a kernel for the GPU in the OpenCL standard.

The remaining sections of the paper are organized as follows. The subsequent section provides a concise overview of Language Model-based Programming (LLM) tools for High-Performance Computing (HPC) approaches. Section three provides a brief explanation of the Nussinov algorithm, while the fourth section delves into polyhedral optimization for this kernel. The following section introduces the process of generating OpenCL code using ChatGPT, utilizing OpenMP codes from Traco [9], Dapt [10], and Pluto [11] for the Nussinov loop nests. The experimental study assesses the efficiency of the generated codes for CPUs and GPUs. Finally, the last section concludes the paper and outlines potential avenues for future work.

II. Related work

Polyhedral frameworks for loop transformation comprise three essential stages: dependency tests, loop transformation via affine operations on polytopes, and code generation from modified loop polytopes. The central process of loop transformation varies across established compilers like Traco, Pluto, and Dapt, yet it plays a pivotal role in optimizing loop structures within program code.

Loop tiling, also referred to as loop blocking or loop partitioning, stands as a compiler optimization technique aimed at enhancing cache utilization and bolstering the performance of loop-based computations [12]. This method involves breaking down a loop into smaller blocks, or tiles, which can be efficiently accommodated within the cache memory. By organizing data access in close proximity to memory, loop tiling diminishes cache misses and optimizes memory access patterns, thereby contributing to overall performance improvements. Additionally, these tiles facilitate parallel processing, further amplifying computational efficiency.

However, achieving parallelization of NPDP kernels often involves employing established strategies such as loop-skewing. The form of multi-threaded code within polyhedral compilers hinges on the tiling algorithm adopted.

The PLUTO compiler [11] leverages the affine transformation framework (ATF) to produce parallel tiled code, utilizing loop transformations to bolster multi-threading capabilities and enhance data locality. It optimizes tiling hyperplanes by

employing an embedded Integer Linear Programming (ILP) cost function, thereby achieving efficient parallelism while minimizing communication overhead in the processor space. TRACO [9], on the other hand, utilizes the transitive closure of dependence relation graphs to generate valid target tiles, rectifying them by eliminating invalid dependence destinations. DAPT [10] addresses non-uniform dependencies by approximating them to uniform counterparts, thereby simplifying complexities associated with nonlinear time-tiling constraints. Unlike PLUTO, DAPT and TRACO support three-dimensional tiling and benchmarks like nussinov, nw, and sw. It's worth noting that Pluto faces limitations in parallelizing mcc code [13].

A drawback of these solutions is the inability to generate code for graphics cards, significantly limiting their applicability on platforms other than CPUs. There have been compilers and converters designed for heterogeneous computing, adhering to standards such as OpenMP and CUDA, for many years, such as Par4All [14] or Cetus [15], but they lack an optimizing engine at the level of the polyhedral model or rely on basic transformations.

The PPCG compiler [16], developed a decade ago, excels in producing optimized GPU code through polyhedral optimization and is actively maintained. Leveraging the ATF framework (specifically, the Pluto algorithm) and the ISL library, this tool generates CUDA [17] and OpenCL codes, showcasing successful performance in stencils and Polybench kernels. However, when applied to NPDP codes, several challenges emerged. PPCG 0.9.1 lacks the capability to generate parameterized code, necessitating constant parameter values during compilation. Consequently, the resulting code contains numerous constants tied to specific parameter values, demanding code regeneration for each parameter set. Hence, we opted to exclude the PPCG compiler from our experimental study.

The evaluation in papers [18], [19], [20] indicates that OpenMP and OpenACC compilers demonstrate proficiency in generating efficient parallel code for simpler cases with GPUs. However, as the complexity of the code increases, a notable performance gap emerges between CUDA or OpenCL and OpenACC, OpenMP. Specifically, when examining memory access patterns such as sum reduction, OpenMP exhibits significantly slower performance compared to the same optimized reduction pattern implemented in CUDA or OpenCL.

Alternatively, developers have the option to utilize solutions like LLM for creating a prototype of the target code, evaluating its performance, and subsequently integrating it into tool implementations. Nichols et al. [1] have further refined the application of LLMs to improve the generation of OpenMP pragmas, with extensions to MPI cases. In a related context, Chen et al. [21] introduced LM4HPC, a framework specifically tailored for HPC tasks using LLMs. They tackled the challenge of limited training and evaluation datasets in HPC by proposing an approach to identify parallelism in code through machine learning techniques.

In the two recent studies, the topic of generating efficient code for classical benchmarks using artificial intelligence (AI) has garnered attention in the scientific literature. Godoy et al. [22] delved into AI-driven generative capabilities, focusing on fundamental numerical kernels in high-performance computing (HPC). Their evaluation encompassed various programming models like OpenMP and CUDA, across languages such as C++ or Python, utilizing both CPU and GPU processing. GitHub Copilot [23], powered by OpenAI Codex [24], was employed to generate multiple implementations based on prompt variations. Subsequently, Pedro et al. [25] revisited the experimental investigation using the Llama-2 engine [26], aiming to generate high-quality HPC codes for the same benchmarks and programming language models. Despite Llama-2's focus on providing optimized code solutions, the study noted a trade-off in terms of reliability when compared to Copilot.

Hence, however, while OpenCL demands significantly more programming effort, applying LLM techniques may render the code generation process easier and improve performance.

III. Nussinov RNA folding

Nussinov pioneered one of the earliest attempts at computationally efficient RNA folding using the base pair maximization approach in 1978 [4]. An RNA sequence comprises a chain of nucleotides from the alphabet G (guanine), A (adenine), U (uracil), C (cytosine). The Nussinov algorithm addresses the challenge of predicting RNA non-crossing secondary structures by calculating the maximum number of base pairs for sub-sequences. It initiates the process with sub-sequences of length 1 and incrementally builds upwards, storing the result of each sub-sequence in a DP array.

Let N be a $n \times n$ Nussinov matrix and $\sigma(i,j)$ be a function which returns 1 if RNA[i], RNA[j] are a pair in the set (*AU*, *UG*, *GC*) and $i < j - 1$, or 0 otherwise. Then the following recursion $N(i,j)$ is defined over the region $1 \leq i \leq j \leq n$ as

$$N_{i,j} = max(N_{i+1,j-1} + \sigma(i,j), \max_{1 \leq j \leq n}(N_{i,k} + N_{k+1,j})) \quad (1)$$

and zero elsewhere [27].

The equation leads directly to the C/C++ code with triple-nested loops presented in Listing 1 [5].

Listing 1. Nussinov loop nest.

```
for (i = N-1; i >= 0; i--) {
 for (j = i+1; j < N; j++) {
  for (k = 0; k < j-i; k++) {
   S[i][j] = MAX(S[i][k+i] + S[k+i+1][j], S[i][j]);
  }
  S[i][j] = MAX(S[i][j], S[i+1][j-1] + signa(i,j));
 }
}
```

IV. Polyhedral optimization for Nussinov loop nests

The polyhedral model represents loop nests as polyhedra with affine loop bounds and schedules. This model provides a foundation for advanced loop transformations and the analysis of data dependences. By harnessing the power of the polyhedral model, compilers can automatically optimize loops,

enhance performance (especially in terms of locality with loop tiling), and exploit parallelism (particularly with loop skewing for NPDP codes) [28].

The polyhedral model is implemented in compilers such as Pluto [29], Dapt [10], and Traco [30], each based on ATF, space-time tiling, and tile correction, respectively. Pluto excels in generating well-balanced affine schedules, but for NPDP codes, the framework can address a set of affine equations to tile all loop nests [30]. Traco compiler generates 3D tiles using the transitive closure of the dependence graph of the union of loop dependences. Further transformations for 3D-tiling of NPDP codes were implemented in the Dapt compiler by dividing the iteration space into timed parallel spaces. Dapt addresses irregularities in code obtained in Traco, providing a comprehensive solution to optimize and refine the generated code [31].

Listing 2. The OpenMP Pluto code for Nussinov RNA folding code.

```
for (t2=1;t2<=N-1;t2++) {
    lbp=t2;  ubp=N-1;
#pragma omp parallel for private(lbv,ubv,t4,t6)
    for (t4=lbp;t4<=ubp;t4++) {
      for (t6=0;t6<=t2-1;t6++) {
        S[-t2+t4][t4] = MAX(S[(-t2+t4)][t6+(-t2+t4)]
        + S[t6+(-t2+t4)+1][t4], S[(-t2+t4)][t4]);
      }
      S[(-t2+t4)][t4] = MAX(S[(-t2+t4)][t4],
          S[(-t2+t4)+1][t4-1] + pair((-t2+t4), t4));
}  }
```

To parallelize the Nussinov RNA folding code, compilers apply the well-known loop skewing transformation [30]. This transformation yields code in which the outermost loop becomes serial, enabling the parallelization of the remaining loops. Listing 2 showcases the parallel code generated using Pluto for the Nussinov code presented in Listing 1. This code, along with the equivalents from Traco and Dapt, serves as input for the OpenCL code generator in ChatGPT.

V. OpenCL code generation with GPT-3.5

To generate codes, we utilize Traco's output code adhering to OpenMP standards. GPT serializes the code; however, if we do not specify which program loop nest is parallel by placing atomic functions, we provide hints about parallel loop nests, and then GPT correctly prepares kernel spawning. The remainder of the code is generated automatically, including context and kernel source loading, as well as memory transfer. The main skeleton of code is presented on Listing 3. Finally, we instruct GPT to generate separate kernels for Pluto and Dapt in a similar manner as for TRACO. The kernel of TRACO code is presented on Listing 4.

GPT adeptly prepares source kernels and seamlessly integrates them into the main code. It facilitates memory allocation, platform management, context selection, and command queue building. This tool streamlines program and kernel argument construction. It effectively generates a serial outermost loop that spawns parallel kernels on the GPU. GPT also autogenerates code for time measurements and compares host and device output arrays. The inclusion of OpenCL object releases is automated at the end of the program.

In kernels, only 1-dimensional arrays are accepted. ChatGPT-3.5 linearizes 2-dimensional arrays and appropriately sets loop bounds in the target OpenCL kernels for each polyhedral optimizer—Traco, Pluto, and Dapt — within the generated OpenMP codes.

The full documentation of the GPT session is available on the GitHub repository page: https://markpal.github.io/fedcsis24/, accessed on 1 March 2024. We employed straightforward, communicative English, which proved sufficient for GPT.

VI. Experimental study

In the experimental study, our primary objective was to validate the correctness of the codes generated by the language model. Unlike polyhedral compilers, which often rely on mathematical proofs, our approach necessitated empirical testing due to the absence of such proofs. We began by scrutinizing the structure of the generated codes, focusing on aspects like loop spawning in GPU kernels, kernel arguments, and linearized array addresses. Furthermore, we compared the values of output arrays with those obtained from computations performed on the host CPU using OpenMP, generated with Pluto, Traco and Dapt. Remarkably, the results exhibited consistency between the outputs. While GPT occasionally produced trivial errors, these were easily rectified, yielding code that met the required standards.

To evaluate the performance of the generated OpenCL codes for the studied Nussinov kernel, we conducted experiments on two modern Intel processors and three graphic cards from NVIDIA, AMD and Intel. Our assessment utilized an Intel Xeon Gold 6326 machine, featuring 36 hardware threads running at 3.5 GHz in turbo mode, alongside a substantial 48 MB L3 cache and 128 GB of RAM. Complementing the CPU, the system incorporated an NVIDIA A100 Tensor Core GPU equipped with 6912 CUDA cores and 80 GB of memory. We tested also the second card, AMD Radeon RX 6700S. It uses the Navi 22 chip based on the new RDNA 2 architecture. The 128 Bit memory system connects 8 GB GDDR6 with 2 GHz memory clock. Furthermore, the RX6800S includes 32 MB Infinity Cache. We analyse also the Intel Core i5 12th 1235-U CPU with 12MB Cache L3 Cache and 16MB RAM with Intel Iris Xe GPU 1.4GHz with 80 execution units.

The experimental setup operated on the Ubuntu 22.04 operating system, and we compiled the programs using the Intel C Compiler (icc 2021), gcc 11.4.0, and clang 14.0 for OpenCL with the -O3 optimization flag. The codes utilized in our study are accessible via the repository link: https://github.com/markpal/fedcsis24/, accessed on 8 April 2024.

The Table 1 contains measurements of the execution time for the Xeon Gold processor and the A100 and Radeon graphics cards for the original code and OpenMP tiled code for Traco, Dapt, Pluto compiled with the Intel C++ compiler, and OpenCL compiled with Clang. OpenCL enables the utilization of polyhedral codes for graphics cards. The execution times for the Tesla A100 are significantly faster than the tiled code on the CPU, and while AMD codes are slower, they still show

Listing 3. A wide listing float, single column

```
#include <CL/cl.h>
...
int main() {
... // declarations, source of kernel load
    int n = 30000;    int* h_S, *cpu_S;     char* kernelSource = (char*)malloc(fileSize + 1);
    FILE* file = fopen("computeS.cl", "r"); fread(kernelSource, 1, fileSize, file);
... // memory allocation
    h_S = (int*)malloc(n * n * sizeof(int));
    cpu_S = (int*)malloc(n * n * sizeof(int));
... // OpenCL classes for target platform
    cl_int err;    cl_platform_id cpPlatform[2];        cl_uint platf_num;
    cl_device_id device;    err = clGetPlatformIDs(1, cpPlatform, &platf_num);
...// Context, Queue and Program Build
    cl_context context = clCreateContext(0, 1, &device, NULL, NULL, &err);
    cl_command_queue queue = clCreateCommandQueue(context, device, 0, &err);
    cl_program program = clCreateProgramWithSource(context, 1, sources, NULL, &err);
    clBuildProgram(program, 1, &device, NULL, NULL, NULL);

    cl_int build_status;
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_STATUS, sizeof(cl_int), &build_status, NULL);

    if (build_status != CL_SUCCESS) {
        // Print compilation errors
...     return 1; // or some other error code
    }
    // Kernel & buffer init, S array passing
    cl_kernel kernel = clCreateKernel(program, "computeS_pluto", &err);
    cl_mem d_S = clCreateBuffer(context, CL_MEM_READ_WRITE, n * n * sizeof(int), NULL, &err);
    clEnqueueWriteBuffer(queue, d_S, CL_TRUE, 0, n * n * sizeof(int), h_S, 0, NULL, NULL);
... // Arguments to kernel
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_S);
    clSetKernelArg(kernel, 1, sizeof(int), &n);
    clSetKernelArg(kernel, 2, sizeof(int), &chunk);
... // Nussionov calculation on device
    auto gpu_start = std::chrono::high_resolution_clock::now();
    for (int c1 = 1; c1 < 2 * n - 2; c1 += 1) {
        clSetKernelArg(kernel, 3, sizeof(int), &c1);
        clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
        clFinish(queue);
    }
    auto gpu_end = std::chrono::high_resolution_clock::now();
    // Array S receive
    clEnqueueReadBuffer(queue, d_S, CL_TRUE, 0, n * n * sizeof(int), h_S, 0, NULL, NULL);
    // Host computation for OpenCL code validation
    auto cpu_start = std::chrono::high_resolution_clock::now();
    // host computation ...
    auto cpu_end = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n * n; i++) // host validation
       assert(h_S[i] == cpu_S[i]);
... // OpenCL object releases
    clReleaseMemObject(d_S);    clReleaseKernel(kernel);    clReleaseProgram(program);
    clReleaseCommandQueue(queue);    clReleaseContext(context);
...
}
```

acceleration compared to the original code using the same code as the NVIDIA card. The measurements were conducted for RNA sequences with nucleotide counts ranging from 1000 to 30000.

In the subsequent Table 2, the timing results for Intel hardware; i5 processor with dedicated Iris Xe graphics are compared. Tiled code was compiled with the gcc compiler, and GPU code was compiled with Clang. Although for Dapt codes, which feature an aggressive tiling algorithm for NPDP codes, the Iris Xe sometimes lags behind, for TRACO tile-corrected codes, the graphics card exhibits comparable performance, and even faster for Pluto codes. However, Pluto codes do not tile the innermost loop [30].

In summary, correct and efficient parallel code was generated on graphics cards. Although the code is the same for graphics cards, three different kernels constructed using LLM based on the output OpenMP codes from Pluto, Traco, and Dapt were placed in the OpenCL kernel file. GPU of each manufacturer showed acceleration compared to sequential code executed on the host.

VII. Conclusion

In this study, we successfully generated efficient and correct OpenCL code, which we verified using cards from three

Listing 4. A wide listing float, single column

```
__kernel void computeS_pluto(__global int* d_S, int n, int CHUNK_SIZE, int t2) {
    int globalThreadIdx = get_global_id(0);
    int t4_base = globalThreadIdx * CHUNK_SIZE + t2;
    for (int offset = 0; offset < CHUNK_SIZE && (t4_base + offset) <= n - 1; offset++) {
        int t4 = t4_base + offset;
        for (int t6 = 0; t6 <= t2 - 1; t6++) {
            d_S[(-t2 + t4) * n + t4] = max(d_S[(-t2 + t4) * n + t6 + (-t2 + t4)]
                        + d_S[(t6 + (-t2 + t4) + 1) * n + t4], d_S[(-t2 + t4) * n + t4]);
        }
        d_S[(-t2 + t4) * n + t4] = max(d_S[(-t2 + t4) * n + t4], d_S[(-t2 + t4 + 1) * n + t4 - 1]
           + pair(-t2+t4, t4));
    }
}
```

TABLE I
TIME EXECUTION IN SECONDS FOR XEON GOLD, NVIDIA A100 AND AMD RADEON, AND OPENMP/OPENCL LIBRARIES.

size	XEON GOLD				NVIDIA A100			AMD Radeon		
	original	traco	dapt	pluto	dapt	traco	pluto	dapt	traco	pluto
	icc	icc + openmp			clang + opencl			clang + opencl		
1000	0,35	0,23	0,18	0,05	0,24	0,23	0,13	0,28	0,18	0,28
2500	7,94	1,68	0,46	1,25	1,48	1,47	0,76	1,68	1,07	1,68
5000	165.32	7.84	6.71	4.33	8,18	8,15	4,24	7,23	5,14	7,23
7500	754.11	21.29	21.09	16.46	44,59	43,16	16,98	23,12	21,76	22,18
10000	1696.16	44.83	48.49	115.71	55,47	56,62	28,37	67,19	64,11	67,06
15000	5183.39	133.99	127.81	398.54	105,76	106,2	56,6	179,80	193,37	181,90
20000	13924.45	295.48	283.14	1100.78	243,53	239,98	119,8	582,49	602,83	580,06
30000	60565.98	972.52	917.96	4056,29	494,24	496,68	307,1	2 142,12	2 267,41	2 197,76

TABLE II
TIME EXECUTION IN SECONDS FOR CORE I5 AND IRIS XE, AND OPENMP/OPENCL LIBRARIES.

size	Intel Core i5				Intel Iris Xe		
	original	dapt	traco	pluto	dapt	traco	pluto
	gcc	gcc + openmp			clang + opencl		
1000	0.46	0,28	0,38	0,12	0,64	0,64	0,42
2500	14.15	2,35	3,22	2,08	3,57	3,84	2,67
5000	114.06	12,24	19,89	21,5	19,3	19,24	16,9
7500	404.86	40,2	61,86	101,9	65,76	66,06	57,27
10000	2261.75	114,27	181,33	717,31	190,89	191,59	161,85
15000	>3000	368,08	551,48	2438	437,53	442,03	435,28
20000	>3000	851,1	1478,94	>3000	1591,09	1589,77	1286,72

different manufacturers. This expanded the possibilities of utilizing results from polyhedral compilers and the concept of automated programming itself. The OpenCL code was created without writing a single line of code.

In future studies, we intend to explore further possibilities of LLM tools, including GPT-4, Github Copilot, or Llama-2, for various NPDP benchmarks. We are interested in the potential of utilizing once constructed code on other similar codes with non-uniform dependencies. Additionally, we plan to investigate optimization techniques, such as the use of shared memory, to enhance the performance of the generated code. LLM tools expand the capabilities of automated programming and appear to be a promising solution in heterogeneous programming.

REFERENCES

[1] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "Modeling parallel programs using large language models," 2023, accessed on: 2024-01-11.

[2] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.1*, 2011. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[3] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," https://www.openmp.org/specifications, 2021, accessed on: 2023-10-22.

[4] R. Nussinov *et al.*, "Algorithms for loop matchings," *SIAM Journal on Applied mathematics*, vol. 35, no. 1, pp. 68–82, 1978.

[5] R. T. Mullapudi and U. Bondhugula, "Tiling for dynamic scheduling," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, S. Rajopadhye and S. Verdoolaege, Eds., Vienna, Austria, Jan. 2014.

[6] D. Wonnacott, T. Jin, and A. Lake, "Automatic tiling of "mostly-tileable" loop nests," in *5th International Workshop on Polyhedral Compilation Techniques*, Amsterdam, 2015.

[7] R. Chowdhury, , and et. al., "Autogen: Automatic discovery of efficient recursive divide-8-conquer algorithms for solving dynamic programming problems," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 1–30, oct 2017. doi: 10.1145/3125632

[8] W. Bielecki, P. Blaszynski, and M. Poliwoda, "3d parallel tiled code implementing a modified Knuth's optimal binary search tree algorithm," *Journal of Computational Science*, vol. 48, p. 101246, jan 2021. doi: 10.1016/j.jocs.2020.101246

[9] W. Bielecki and M. Palkowski, "A parallelizing and optimizing compiler - traco," http://traco.sourceforge.net, 2013, accessed on: 2024-01-11.

[10] W. Bielecki and M. Poliwoda, "Automatic parallel tiled code generation based on dependence approximation," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Cham: Springer International Publishing, 2021, pp. 260–275.

[11] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: http://pluto-compiler.sourceforge.net

[12] J. Xue, *Loop Tiling for Parallelism*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-7933-0

[13] M. Palkowski and W. Bielecki, "NPDP benchmark suite for the evaluation of the effectiveness of automatic optimizing compilers," *Parallel Computing*, vol. 116, p. 103016, Jul. 2023. doi: 10.1016/j.parco.2023.103016. [Online]. Available: https://doi.org/10.1016/j.parco.2023.103016

[14] A. Mehdi, *Par4All User Guide*, 2012. [Online]. Available: http://www.par4all.org

[15] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, pp. 36–42, 2009.

[16] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, p. 1–23, Jan. 2013. doi: 10.1145/2400682.2400713. [Online]. Available: http://dx.doi.org/10.1145/2400682.2400713

[17] "Nvidia corporation, cuda programming guide 12.3," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2023, accessed on: 2023-10-22.

[18] K. Thouti and S. R. Sathe, "Comparison of openmp & opencl parallel processing technologies," 2012. doi: 10.48550/ARXIV.1211.2038. [Online]. Available: https://arxiv.org/abs/1211.2038

[19] M. Khalilov and A. Timoveev, "Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu," *Journal of Physics: Conference Series*, vol. 1740, no. 1, p. 012056, Jan. 2021. doi: 10.1088/1742-6596/1740/1/012056. [Online]. Available: http://dx.doi.org/10.1088/1742-6596/1740/1/012056

[20] G. Kan, X. He, L. Ding, J. Li, K. Liang, and Y. Hong, "A heterogeneous computing accelerated sce-ua global optimization method using openmp, opencl, cuda, and openacc," *Water Science and Technology*, vol. 76, no. 7, p. 1640–1651, Jun. 2017. doi: 10.2166/wst.2017.322. [Online]. Available: http://dx.doi.org/10.2166/wst.2017.322

[21] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. de Supinski, *LM4HPC: Towards Effective Language Model Application in High-Performance Computing*. Springer Nature Switzerland, 2023, p. 18–33. ISBN 9783031407444. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-40744-4_2

[22] W. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, and J. Vetter, "Evaluation of openai codex for hpc parallel programming models kernel generation," in *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ser. ICPP-W 2023. ACM, Aug. 2023. doi: 10.1145/3605731.3605886. [Online]. Available: http://dx.doi.org/10.1145/3605731.3605886

[23] G. C. Team, "Github copilot," https://copilot.github.com/, 2022, an AI pair programmer for GitHub, Accessed on: 2023-10-22.

[24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," https://arxiv.org/abs/2107.03374, 2021, accessed on: 2023-10-22.

[25] P. Valero-Lara, A. Huante, M. A. Lail, W. F. Godoy, K. Teranishi, P. Balaprakash, and J. S. Vetter, "Comparing llama-2 and gpt-3 llms for hpc kernels generation," 2023. [Online]. Available: https://arxiv.org/abs/2309.07103

[26] "Introducing llama 2, the next generation of our open source large language model," https://ai.meta.com/llama/, 2023, accessed on: 2023-10-22.

[27] D. Wonnacott, T. Jin, and A. Lake, "Automatic tiling of "mostly-tileable" loop nests," in *IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques, At Amsterdam, The Netherlands*, 2015.

[28] S. Verdoolaege, "Integer set library - manual," www.kotnet.org/~skimo//isl/manual.pdf, 2011, accessed on: 2024-01-11.

[29] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. doi: 10.1145/1379022.1375595

[30] M. Palkowski and W. Bielecki, "Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing," *BMC Bioinformatics*, vol. 18, no. 1, p. 290, 2017. doi: 10.1186/s12859-017-1707-8

[31] M. Palkowski and M. Gruzewski, "Time and energy benefits of using automatic optimization compilers for NPDP tasks," *Electronics*, vol. 12, no. 17, p. 3579, Aug. 2023. doi: 10.3390/electronics12173579. [Online]. Available: http://dx.doi.org/10.3390/electronics12173579