electronics

MDPI

Article

GPT-Driven Source-to-Source Transformation for Generating Compilable Parallel CUDA Code for Nussinov's Algorithm

Marek Palkowski *,† and Mateusz Gruzewski †

Faculty of Computer Science, West Pomeranian University of Technology, Zolnierska 49, 72210 Szczecin, Poland; gruzewski94@gmail.com

* Correspondence: mpalkowski@zut.edu.pl

† These authors contributed equally to this work.

Abstract: Designing automatic optimizing compilers is an advanced engineering process requiring a great deal of expertise, programming, testing, and experimentation. Maintaining the approach and adapting it to evolving libraries and environments is a time-consuming effort. In recent years, OpenAI has presented the GPT model, which is designed for many fields like computer science, image processing, linguistics, and medicine. It also supports automatic programming and translation between programming languages, as well as human languages. This article will verify the usability of the commonly known LLM model, GPT, for the non-trivial NPDP Nussinov's parallel algorithm code within the OpenMP standard to create a parallel equivalent of CUDA for NVIDIA graphics cards. The goal of this approach is to avoid creating any post-processing scripts and writing any lines of target code. To validate the output code, we compare the resulting arrays with the ones calculated by the optimized code for the CPU generated employing the polyhedral compilers. Finally, the code will be checked for scalability and performance. We will concentrate on assessing the capabilities of GPT, highlighting common challenges that can be refined during future learning processes. This will enhance code generation for various platforms by leveraging the outcomes from polyhedral optimizers.

Keywords: NPDP; LLM, artificial intelligence; CUDA; parallel computing; automatic programming; RNA folding; locality, scalability; GPU computing

Citation: Palkowski, M.; Gruzewski, M. GPT-Driven Source-to-Source Transformation for Generating Compilable Parallel CUDA Code for Nussinov's Algorithm. *Electronics* **2024**, *13*, 488. https://doi.org/10.3390/electronics13030488

Academic Editor: Ping-Feng Pai

Received: 19 December 2023
Revised: 21 January 2024
Accepted: 22 January 2024
Published: 24 January 2024

1. Introduction

Source-to-source compilers optimize program loops because they contain the majority of computations. Most of these tools utilize the polyhedral model, a mathematical framework for analyzing and optimizing nested loop structures in programs [1]. It allows for advanced loop transformations, which are particularly beneficial for regular computation patterns in scientific and numerical applications. Solutions like polyhedral compilers, such as TRACO [2], DAPT [3], and Pluto [4], produce parallel and cache-efficient code annotated with OpenMP pragmas [5]. The challenge lies in porting this code to target platforms like graphics cards, given libraries like CUDA [6], SYCL [7], and Kokkos [8]. Compiler development teams focus on defining code at a high level of abstraction, independent of the target programming architecture. Adapting the resulting code to a specific architecture is tedious, error-prone, time-consuming, and hard to maintain. Many solutions are poorly documented or are overshadowed by other libraries. The limited popularity or constraints of solutions like polyhedral compilers (i.e., Pluto, PPCG [9]) indicate a need for alternative approaches.

Introduced in 2022, the Generative Pre-trained Transformer 3 (GPT-3.5) [10] marked a transformative advancement in the realm of human–computer interactions. Crafted by OpenAI, GPT-3.5 (also named ChatGPT) represents the third iteration of prediction-based large language models (LLMs) and serves as the backbone for numerous AI-driven

applications that produce human-like textual content. ChatGPT employs sophisticated deep learning techniques to process and respond to textual inputs. By leveraging vast amounts of textual data from sources like books, articles, and online content, it can comprehend intricate linguistic structures and nuances. This extensive training enables ChatGPT to generate insightful and coherent responses consistent with human-like language understanding.

In recent years, other chatbot solutions based on artificial intelligence have been also developed, facilitating automated programming with the assistance of large language models [11–21]. For example, ChatSonic [13], which was trained on a similar dataset as ChatGPT and naturally became an alternative to it. Compared to ChatGPT, it also allows users to generate images and voice support and has a code editor. It enables automatic code completion, suggestions for optimization, and advice on good programming practices.

Along with the possibility of generating code using artificial intelligence, the Github CoPilot [14] tool appeared, which was created in cooperation between Github and OpenAI. It allows code generation in the development environment as an extension. It is based on the GPT-4 language model. Through the extension, it supports developers in creating code using context-based code generation and analysis of existing code.

The widespread utilization of LLMs in high-performance computing (HPC) code generation has garnered increasing interest and has been extensively documented in recent publications. Nichols et al. [22] refined the application of LLMs to enhance the generation of OpenMP pragmas in parallel algorithm implementations, encompassing MPI cases. In a related context, Chen et al. [23] introduced LM4HPC, a framework designed for HPC-specific tasks within the realm of LLMs. They emphasized the scarcity of training and evaluation datasets in the field of high-performance computing (HPC). It is noteworthy that the paper presented an attempt to detect parallelism in codes from the dataset using machine learning.

In [24], the authors assessed AI-driven generative capabilities on fundamental numerical kernels in high-performance computing (HPC), covering AXPY, GEMV, GEMM, SpMV, Jacobi Stencil, and CG. They examined the generated kernel codes across various programming models, including OpenMP, CUDA, and others, for the C++, Fortran, Python, and Julia languages, employing CPU and GPU processing. They leveraged GitHub Copilot [14], powered by OpenAI Codex [25], in Visual Studio Code to generate numerous implementations based on simple prompt variations such as <kernel> + <programming model> + <optional hints>. To assess and compare the results, they introduced a proficiency metric based on the initial 10 suggestions for each prompt. Their findings indicated that OpenAI Codex outputs for C++ correlate with the popularity and maturity of programming models, with high scores for OpenMP and CUDA but lower scores for HIP. Prompts in targeted languages like Fortran or the more general-purpose Python benefit from code keywords, whereas Julia prompts perform well for its mature programming models (e.g., Threads and CUDA.jl). The considered benchmarks aim to serve as a reference point for each programming model's community. The authors claimed that understanding the synergy of large language models, AI, and HPC is crucial due to its rapidly evolving nature and its impact on redefining human–computer interactions.

In a subsequent study [26] conducted by the same research team a few months later, the authors revisited the experimental investigation using the Llama-2 engine [12]. Llama-2 is available for free for both research and commercial use. This follow-up research aimed to generate high-quality HPC codes for the same set of benchmarks and programming language models. In their comparison of Llama-2 with GitHub Copilot, the authors observed that although Llama-2 strives to provide more optimized code solutions, it exhibits a trade-off in terms of reliability when compared to Copilot.

In the realm of high-performance computing (HPC), where hardware and programming models exhibit extreme diversity, ChatGPT-assisted code generation has the potential to revolutionize the way we create, deploy, and test software for optimizing compilers and target HPC systems.

In this article, we pose the following question: Using AI-based solutions, is it possible to generate code for graphics cards based on parallel OpenMP code without significant effort? Is this code valid, and most importantly, can a trained model consistently regenerate code for computational tasks that are similar but vary significantly in structure? For this purpose, we have chosen Nussinov's RNA folding algorithm [27], which is not trivial to optimize using polyhedral compilers [28]. To train our model, we will rely on the outcomes of the TRACO compiler, which can produce parallel code but solely for CPUs. We anticipate that GPT will generate CUDA code that is both compilable and efficient. The code is expected to facilitate valid memory transfers between the CPU and GPU, accurately perform kernel calculations, and yield results consistent with the host version.

The Nussinov algorithm is classified as a non-serial polyadic dynamic programming (NPDP) kernel. These NPDP kernels are employed to evaluate the efficiency of tiled code generated through state-of-the-art optimizing compilers [28–31]. The NPDP dependence pattern represents the most complex category of dynamic programming (DP) due to its non-uniform dependences, characterized by irregularities and represented using affine expressions. The idea of DP is to start from the simplest instance of a problem, find an optimal solution for it, and extend the optimal solution to bigger instances.

In this paper, we utilize the TRACO output to compare two other automatic compilers, namely Pluto [4] and DAPT [3], for the generation of optimized codes in the context of the Nussinov algorithm [32]. The base algorithms, like Nussinov's approach, encompass intricate NPDP dependence patterns that pose limitations in achieving high-performance and energy-efficient code through automatic optimizers. These tools not only facilitate multi-threading of the code but also enhance its locality by employing techniques like loop tiling. Pluto utilizes the affine transformation framework (ATF), TRACO implements the transitive closure of the dependence relation graph, and DAPT applies space-time tiling with the dependence uniformization approach. Unfortunately, compilers cannot automatically translate their generated OpenMP code into other programming languages like CUDA or adapt it to run on specific hardware such as GPUs. Therefore, we conduct experiments using the GPT-3.5 [10] approach to replicate multi-threaded code for GPU platforms, assessing the feasibility of AI-assisted code generation. To evaluate the outcomes, we employ modern machines equipped with Intel Xeon Gold and NVIDIA A100 GPUs.

We justify the selection of the CUDA programming interface based on the high performance guaranteed by the library creators and NVIDIA, the manufacturer of the target graphics card. Although there are other standards, such as OpenMP target offloading [5], the effectiveness of these can vary depending on the specific code and the GPU hardware. Toward the end of our research, we conduct a comparison between the generated CUDA code and OpenMP target, which could easily be derived from the polyhedral compiler's OpenMP code designed for the CPU.

The remaining sections of this paper are structured as follows. Section 2 provides a detailed description of Nussinov's RNA folding algorithm, GPT-driven CUDA code generation based on the OpenMP code, and guidelines for the AI code generation provided by the compiler and expert. Section 3 presents the experimental methodology and results. Section 4 discusses the role of automatic programming with GPT for HPC and analyzes the generated codes. Finally, the paper concludes in Section 5, highlighting future directions for research.

2. NPDP Code Optimization

Our motivating code is the Nussinov algorithm [27]. The kernel is a representative NPDP problem that is challenging to optimize using traditional transformations and compilers based on ATF (affine transformation framework) [28,31,33]. Many publications are dedicated to its manual optimization methods (e.g., the "Four Russians" method [34] and transposition method [35]), as well as automatic approaches (like ATF [36], tile correction [32], and time spacing [3]) using CPU and GPU platforms.

2.1. Nussinov's RNA Folding Algorithm

Nussinov made one of the first attempts at folding RNA in a computationally efficient way with the base pair maximization approach in 1978 [27]. An RNA sequence is a chain of nucleotides from the alphabet: G (guanine), A (adenine), U (uracil), and C (cytosine). Given an RNA sequence, the Nussinov algorithm addresses the problem of RNA non-crossing secondary structure prediction by computing the maximum number of base pairs for sub-sequences, starting with sub-sequences of length 1 and building upwards, storing the result of each sub-sequence in a dynamic programming array.

Let N be an $n \times n$ 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 [31].

This equation leads directly to the C/C++ code with triple-nested loops presented in Listing 1 [28].

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]); //s0
  }
  S[i][j] = MAX(S[i][j], S[i+1][j-1] + signa(i,j));//s1
 }
}
```

2.2. GPT-Driven CUDA Code Generation

Using the TRACO compiler, we generated a valid parallel code in OpenMP (Listing 2). This code can also be calculated by applying other polyhedral optimizers such as DAPT or Pluto. To parallelize the input code, we applied the well-known loop skewing transformation [32]. Loop skewing is a transformation employed to reconfigure an iteration space by introducing a new loop, where the index is a linear combination of two or more existing loop indices. This transformation yields code in which the outermost loop becomes serial, enabling the parallelization of the remaining loops.

Listing 2. The OpenMP code of the Nussinov loop nest generated using the TRACO compiler.

```
for( c1 = 1; c1 < 2 * n - 2; c1 += 1)
  #pragma omp parallel for
  for( c3 = max(0, -n + c1 + 1); c3 < (c1 + 1) / 2; c3 += 1) {
        for( c5 = 0; c5 <= c3; c5 += 1)
          S[(n-c1+c3-1)][(n-c1+2*c3)] = MAX(S[(n-c1+c3-1)][(n-c1+c3+
              c5-1)] + S[(n-c1+c3+c5-1)+1][(n-c1+2*c3)], S[(n-c1+c3-1)
              ][(n-c1+2*c3)]);
       S[(n-c1+c3-1)][(n-c1+2*c3)] = MAX(S[(n-c1+c3-1)][(n-c1+2*c3)],
            S[(n-c1+c3-1)+1][(n-c1+2*c3)-1] + sigma((n-c1+c3-1), (n-c1
           +2*c3)));

    }
```

Then, we instructed GPT-3.5 to generate CUDA code using the provided OpenMP code as input. Listing 3 presents the CUDA kernel of Nussinov's calculations generated by GPT. The full documentation of the GPT session is available on the GitHub repository

page: https://markpal.github.io/electronics_jan_2024/, accessed on 22 January 2024. We employed straightforward, communicative English, which proved sufficient for GPT.

The first answer (Listing 3) is somewhat distant from the target code we desire. First, we needed to inform the GTP model that the outermost loop, labeled as $c1$, should be executed serially, whereas the second loop should be parallel. The remaining loops were executed sequentially. GPT could not identify data dependencies within the loop, but providing the aforementioned hints enabled it to generate a valid code.

Listing 3. The CUDA kernel of the Nussinov loop nest with unnecessary atomic statements (generated with GPT).

```
__global__ void dynamicProgrammingKernel(int *d_S, int n) {
    int c1 = blockIdx.x + 1;
    int c3 = threadIdx.x + max(0, -n + c1 + 1);

    if(c3 < (c1 + 1) / 2) {
        for(int c5 = 0; c5 <= c3; c5++) {
            atomicMax(&d_S[(n-c1+c3-1)*n + (n-c1+2*c3)], d_S[(n-c1+c3
                -1)*n + (n-c1+c3+c5-1)] + d_S[(n-c1+c3+c5-1+1)*n + (n-
                c1+2*c3)]);
        }
        atomicMax(&d_S[(n-c1+c3-1)*n + (n-c1+2*c3)], d_S[(n-c1+c3
            -1+1)*n + (n-c1+2*c3)-1] + sigma((n-c1+c3-1), (n-c1+2*c3))
            ); // Assuming you have the sigma function implemented
    }
}
```

The GTP chatbot explained, 'Use the *atomicMax* function to ensure correct concurrent updates, if necessary'. It is known that the parallel code generated by the tool is valid and respects all data dependencies. GPT rectified this by restoring the original MAX functions and recorded this modification for future code transformations.

In the main code, GPT introduced memory transfers (Listing 4). However, we clarified that memory transfers are only required at the beginning and end of GPU calculations. Transferring the scoring array, S, between iterations is unnecessary. This information can be derived from the TRACO optimizer, which analyzes the structure of the original loop. The S array is transferred only before and after parallel calculations. GPT corrected the code according to our hints.

Listing 4. The unnecessary memory transfer between executions of CUDA kernels in sequential program loop (generated with GPT).

```
    for (int c1 = 1; c1 < 2 * n - 2; c1++) {
        // Copy current state of S to device at the beginning of each
            c1 iteration
        cudaMemcpy(d_S, S, n * n * sizeof(int),
            cudaMemcpyHostToDevice);

        // Launch the CUDA kernel
        dynamicProgrammingKernel<<<1, (n + 1) / 2>>>(d_S, n, c1);

        // Copy the results back to the host after each kernel
            execution
        cudaMemcpy(S, d_S, n * n * sizeof(int),
            cudaMemcpyDeviceToHost);
    }
```

We requested GPT to include the device prototype of the sigma function (representing a base pair in Nussinov). It is worth noting that this function uses two read-only arrays containing RNA strands, which also need to be copied to the device memory.

We also asked the GPT model to use 2D arrays in both the kernel and host code (this is possible in CUDA). This resulted in the creation of 2D references within the code body and the addition of flattened arrays for data allocation and transfer. The kernel is more readable for developers and similar to the TRACO OpenMP output code (Listing 5).

Listing 5. The kernel with 2D pointers and parts of the main code (generated with GPT).

```
__global__ void dynamicProgrammingKernel(int** d_S, int n, int c1) {
    int c3 = threadIdx.x + max(0, -n + c1 + 1);

    if(c3 < (c1 + 1) / 2) {
        for(int c5 = 0; c5 <= c3; c5++) {
            d_S[n-c1+c3-1][n-c1+2*c3] = max(d_S[n-c1+c3-1][n-c1+c3+c5
                -1] + d_S[n-c1+c3+c5-1+1][n-c1+2*c3], d_S[n-c1+c3-1][n
                -c1+2*c3]);
        }
        d_S[n-c1+c3-1][n-c1+2*c3] = max(d_S[n-c1+c3-1][n-c1+2*c3],
            d_S[n-c1+c3-1+1][n-c1+2*c3-1] + sigma((n-c1+c3-1), (n-c1
            +2*c3)));
    }
}

// Allocate and initialize 2D host array
    int** S = new int*[n];
    for(int i = 0; i < n; i++) {
        S[i] = new int[n];
        // Optionally, initialize the array elements here.
    }

    // Allocate 2D device array (interpreted from a flattened layout)
    int* flat_d_S;
    int** d_S;

    cudaMalloc(&flat_d_S, n * n * sizeof(int));
    cudaMalloc(&d_S, n * sizeof(int*));

    int* h_S[n];
    for(int i = 0; i < n; i++) {
        h_S[i] = flat_d_S + i * n;
    }
    cudaMemcpy(d_S, h_S, n * sizeof(int*), cudaMemcpyHostToDevice);
```

It is worth noting that programming with pointers can be challenging and prone to manual errors.

Subsequently, we requested GPT to generate host calculations and compared the outputs to ensure correctness. GPT introduced the *S_CPU* array and added validation code at the end. Initially, when we tried to compile the code with the *nvcc* compiler, we encountered an error, indicating that the sigma function had no implementation for the host. GPT resolved this issue.

Following this, our generated code compiled without any syntax errors, but we discovered a logic error when running the code. The CPU and GPU arrays differed. GPT assisted us in identifying the problem.

We resolved the issue of array differences by adding synchronization with the following message: '*Synchronization: After the kernel launch, use cudaDeviceSynchronize(); to ensure all threads have finished before copying the data back to the host*'. In the OpenMP code,

synchronization is implicit at the end of the *pragma parallel* and is executed automatically. In the CUDA code, explicit barriers are required.

However, the arrays still exhibited differences. GPT provided guidance with the message 'Ensure Coherent Memory Accesses'. We identified the issue: the memory allocation for 2D data was fragmented over time and needed defragmentation. In memory transfer, we declared that the arrays were aligned. We informed GPT about this, and the model corrected the code accordingly. The changes are illustrated in Listing 6.

Listing 6. The defragmented dynamic 2D array allocation (generated with GPT).

```
    // Allocation
    int* flatArray_S = new int[n * n];
    int** S = new int*[n];
    int* flatArray_S_CPU = new int[n * n];
    int** S_CPU = new int*[n];

    for(int i = 0; i < n; i++) {
        S[i] = &flatArray_S[i * n];
        S_CPU[i] = &flatArray_S_CPU[i * n];
    }
```

The code was functional for short RNA strands, but when their length exceeded 2048, the CPU and GPU arrays exhibited discrepancies. GPT pointed out that we did not have enough threads in the block and recommended the addition of chunking (Listing 7). As a result, the kernel executed a group of $c3$ values of the index variable in a parallel loop. We employed this chunking model to generate the subsequent codes.

Listing 7. The chunking calculations into blocks (generated with GPT).

```
...
int threadsNeeded = (n + 1) / 2;
    int numBlocks = (threadsNeeded + BLOCK_SIZE - 1) / BLOCK_SIZE;
    int chunk_size = (threadsNeeded + numBlocks * BLOCK_SIZE - 1) / (
        numBlocks * BLOCK_SIZE);

    for (int c1 = 1; c1 < 2 * n - 2; c1++) {
        dynamicProgrammingKernel<<<numBlocks, BLOCK_SIZE>>>(d_S, n,
            c1, chunk_size);
        cudaDeviceSynchronize();  // Ensure all threads finish before
             proceeding to the next iteration
    }
...
```

In summary, for code generation, the GPT model needs to be informed about the following tasks:

- Determine which loop is serial or parallel;
- Specify arrays to copy between the CPU and GPU memory (in, out, or in/out);
- Avoid unnecessary communication between the host and device memory during calculations;
- Provide the internal function prototypes;
- Synchronize parallel calculations;
- Declare dynamic memory arrays as flattened;
- Utilize 2D array floating pointers in the kernel;
- Provide chunking;
- Verify with the input OpenMP code on the CPU.

The first four points can be handled by the compiler, whereas the remaining steps require expert control to produce readable and CUDA-valid code. Summing up, the output was generated without any manual coding effort.

3. Results

In the subsequent step, we attempted to employ the procedure outlined in the previous section to re-generate the CUDA code corresponding to the optimized program loops calculated with DAPT and Pluto. However, GPT failed to reproduce the results when we provided instructions as a single group or sequentially. The model seemed to forget the prior hints about the non-identical nature of code generation with the target, resulting in codes that deviated significantly from the desired pattern. Consequently, each session was stateless; hence, we explored an alternative solution.

We provided the input code produced by TRACO and the achieved output code as examples. We then requested GPT to replicate these steps for other OpenMP codes generated with DAPT and Pluto. The GitHub repository page https://markpal.github.io/electronics_jan_2024/CreateDAPTandPlutofromTracoCUDA.html (accessed on 21 January 2024) presents a short session on creating these codes. This approach proved highly effective, as it enabled us to obtain compilable code with all the desired features. In conclusion, we established a framework outlining the entire procedure for generating the results, as illustrated in Figure 1.

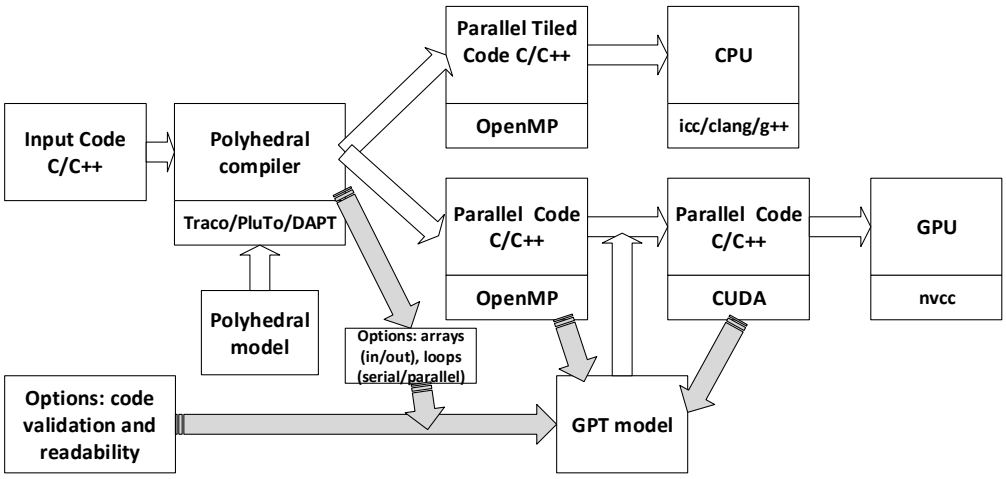

Figure 1. GPT-driven CUDA code generation using polyhedral parallel codes.

According to the schema, the main idea is the usage of the polyhedral compiler and the LLM model to produce target-optimized code for new platforms. First, input C++ code is optimized using polyhedral frameworks such as ATF. The compiler provides technical details regarding parallelism and the input/output arrays of data. The OpenMP code is transformed into the target code, such as CUDA, using additional information for code validation and readability. The GPT model is also trained with its own generated codes to produce similar ones in the future. The output code is compiled using the *nvcc* compiler and executed on the target GPU card. The GPT model can be replaced with other LLM models, and changes can be made to the target hardware, language, or framework in the schema.

To evaluate the quality of the generated CUDA codes for Nussinov's approach, we executed them on a modern machine equipped with a parallel processor and a graphics card. To measure the performance of both the CPU and GPU codes, we utilized an Intel Xeon Gold 632 machine, boasting 32 hardware threads, operating at 3.5 GHz in turbo mode, with a 48 MB L3 cache and 128 GB of RAM. Additionally, the system featured an NVIDIA A100 Tensor Core GPU, equipped with 6912 CUDA cores and 80 GB of memory. The machine was operating under Ubuntu 22.04, and we compiled the programs using the Intel C Compiler (icc 2021) and nvcc 12.3 (2023 release) with the -O3 optimization flag.

To validate the results, we compared the outputs of the CPU and GPU codes after each experiment. The code generated by GPT consistently returned correct results each time, indicating the advanced language model proficiency of the OpenAI product.

The NVIDIA A100 graphics processor represents an advanced solution within the NVIDIA Ampere family of graphics processors, designed for deep computing workloads. Significant improvements in performance and energy efficiency are key aspects of this processor, contributing significantly to the optimization of computational results and the sustainable use of energy resources [37].

The third-generation Tensor core in the NVIDIA Ampere architecture allows for data sharing among all 32 threads in a wrap, representing a significant improvement compared to Volta's Tensor core, which allows for data sharing among 8 threads. Enhanced data sharing reduces the register file bandwidth required to deliver data to Tensor cores. Additionally, the amount of redundant data loaded into shared memory (SMEM) registers is also reduced, leading to savings in both bandwidth and register file storage. In pursuit of increased performance, the A100 Tensor core's instructions increase the k-matrix dimension per instruction by up to 4 times compared to the V100. When performing matrix multiplication operations, the A100 issues 8× times fewer instructions and performs 2.9× times fewer register accesses than the V100 [37].

The codes for the experimental study are available on the GitHub repository: https://github.com/markpal/electronics_jan_2024, accessed on 22 January 2024.

Our primary objective in this study is time comparison. However, on the 32-core Intel Xeon processor, we have access to parallel and tiled versions of the Nussinov algorithm through well-known compilers—Pluto, TRACO, and DAPT—from previous publications [38,39]. On the A100 graphics card, boasting 6192 CUDA cores, the code must be scalable and fine-grained to fully utilize its resources. To address this, we investigated various sizes of input sequences for the Nussinov algorithm, ranging from 5000 to 30,000 nucleotides.

Table 1 presents the time results in seconds for the original code executed on a single thread on the Xeon Gold CPU, parallelized code on all 32 Xeon Gold cores, and parallel and tiled code generated using the TRACO, DAPT, and Pluto compilers for the CPU using OpenMP pragmas. For 30,000 RNA strands, the best time was achieved using the DAPT compiler, with a time of 917.96 s. When using the A100 card, the time results were significantly improved. The best time was observed using the Pluto compiler as input. However, although Pluto did not exhibit the best performance for the CPU codes, its parallel version of the Nussinov algorithm demonstrated outstanding results on the GPU (318.94 s). The parallel code generated using ATF was exceptionally well balanced and scalable, which is crucial for fine-grained calculations. The remaining codes from TRACO and DAPT also exhibited faster execution compared to the CPU. With larger sizes, we observed a significant reduction in computation time on the graphics card, showcasing its advantage over processor computations. The GPT-driven model played a pivotal role in adapting this code for GPU architecture with CUDA, significantly expanding the applicability of commonly known polyhedral optimizers. The CPU and GPU codes consistently produced the same outputs in every experimental case.

The latest versions of OpenMP [5] have introduced support for graphics cards using the pragma omp target. In our study, we aimed to compare the time results obtained using the GPT model with OpenMP codes. We selected the fastest code from the Pluto compiler and implemented it for OpenMP with GPU offload. The code, presented in Listing 8, was compiled with the *nvc -mp=gpu -O3* compiler flags. The time results demonstrate the superiority of CUDA codes over OpenMP codes with GPU offloading (Figure 2). Therefore, our target compilers are native CUDA and OpenCL codes, which, although more challenging to generate, allow for better performance.

Table 1. Time results for Nussinov's algorithm optimized using polyhedral compilers and GPT-driven transformation.

Size	Original	CPU + OpenMP				GPU + CUDA		
		Parallel	Parallel and Tiled			Parallel + GPT		
		TRACO	TRACO	Pluto	DAPT	TRACO	Pluto	DAPT
5000	165.32	15.06	7.84	**4.33**	6.71	11.70	7.53	9.38
7500	754.11	54.95	21.29	**16.46**	21.09	31.03	17.97	25.48
10,000	1696.16	149.64	44.83	115.71	48.49	58.58	**32.44**	45.58
15,000	5183.39	538.95	133.99	398.54	127.81	137.57	**75.92**	108.58
20,000	13,924.45	1426.72	295.48	1100.78	283.14	250.61	**134.98**	193.44
30,000	60,565.98	5272.49	972.52	4056.29	917.96	585.67	**318.94**	458.44

The shortest time for each size was highlighted in bold in the table.

Listing 8. Pluto kernel implemented with OpenMP target offload for GPU.

```
#pragma omp target map(tofrom:S)
 for( c1 = 1; c1 < 2 * n - 2; c1 += 1)
 #pragma omp parallel for private(c3,c5) shared(c1)
  for( c3 = max(0, -n + c1 + 1); c3 < (c1 + 1) / 2; c3 += 1) {
    for( c5 = 0; c5 <= c3; c5 += 1)
      S[(n-c1+c3-1)][(n-c1+2*c3)] = max(S[(n-c1+c3-1)][(n-c1+c3+c5-1)
          ] + S[(n-c1+c3+c5-1)+1][(n-c1+2*c3)], S[(n-c1+c3-1)][(n-c1
          +2*c3)]);
      S[(n-c1+c3-1)][(n-c1+2*c3)] = max(S[(n-c1+c3-1)][(n-c1+2*c3)],
          S[(n-c1+c3-1)+1][(n-c1+2*c3)-1] + sigma((n-c1+c3-1), (n-c1
          +2*c3)));
 }
```

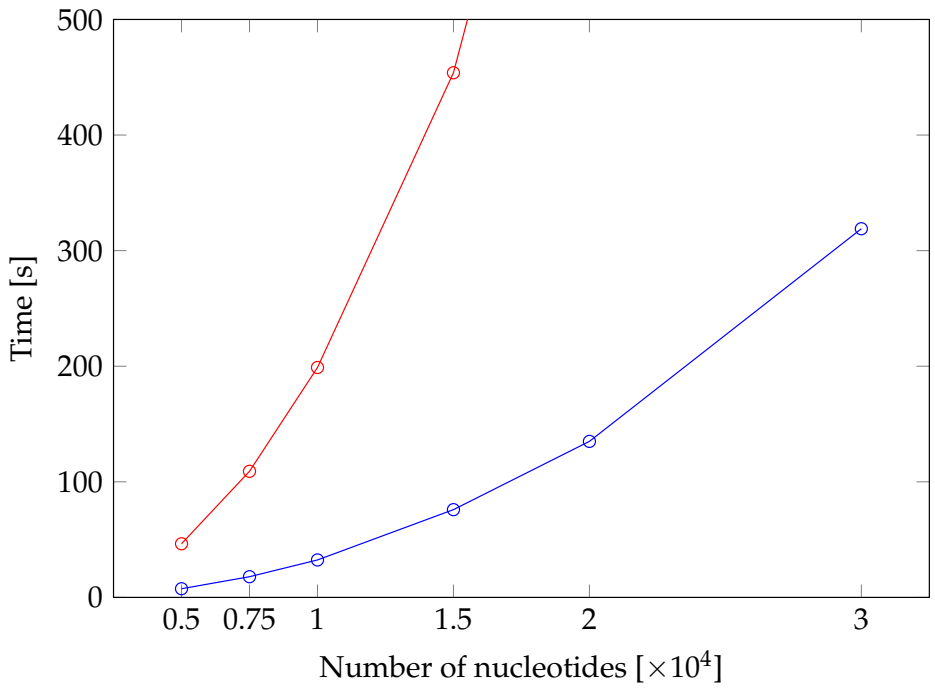

Figure 2. Time comparison for OpenMP with GPU offload (red) vs. CUDA codes generated with GPT (blue).

4. Discussion

Polyhedral compilers consist of three stages: input code analysis and dependence test, loop transformations, and code generation. While dependency analysis and, especially, transformations currently surpass the capabilities of substitution by chatbots—primarily for achieving correct parallelization and code optimization—AI algorithms have made significant progress in parsing, syntax analysis, and code generation. However, there are still many tasks they cannot perform at a human level.

Without specifying parallelism through polyhedral compilers, GPT generates sequential code using atomic operations (without concurrency), or it generates code without honoring dependencies. GPT (Generative Pre-trained Transformer) and similar language models are primarily designed for generating human-like texts based on the training data provided to them. They are not specifically created for the analysis or understanding of the structure of computer programs, including dependencies in program loops.

In our experimental study, we observed that GPT has a limited and stateless ability to generate target code in one step without input from compilers and human experts. However, it is able to reproduce a result from a pattern consisting of input and output code at a satisfactory level. Suggestions by experts must be precise and clear. Importantly, we were able to create code without any manual effort or programming any line of code.

The use of tools like GPT is not entirely straightforward. Firstly, there is no established method for validating the generated output. We relied on our experience and examined the code in terms of the expected result, scrutinizing the structural form of kernels, ensuring correct parallelism implementation, variable passing to kernels as arguments and memory transfer. GPT, however, requires a clear definition of the parallelism location. It lacks its own dependence analysis engine at the input code analysis stage, and we cannot replace data from polyhedral compilers. Ultimately, it could potentially replace certain elements of a compiler, for example, in the structural analysis phase of the input loop.

The usefulness of the codes was assessed after validating the generated results. Surprisingly, these codes were syntactically correct from the first compilations, and further corrections were more related to establishing more detailed input data, such as characteristics of NPDP codes.

The use of TRACO CUDA code to generate DAPT and Pluto code is the most significant observation in this article. The example provided, along with the generated code for the next instance, immediately tuned GPT to produce the desired result. This code underwent minimal editing for error correction and was immediately ready for practical use. In the context of solving similar problems, GPT effectively replaces human effort and can generate many correct solutions with minimal tuning. GPT is also useful for finding errors in code and advising on how to fix them.

In technical terms, the code quality from GPT is very good; memory handling functions did not contain errors, and pointers were handled correctly. We had to specify the form of parallel code, indicating where the synchronization points were (in the OpenMP code, these are implicit places, e.g., with the end of the parallel pragma). Additionally, we instructed that copying data in each iteration of the outer loop was unnecessary. The sequential outer loop forced kernels to be executed many times in NPDP-like codes, which is not a template solution that GPT could learn from other source codes. GPT tended to forget the state and insisted on its solutions, but it responded to commands such as converting array handling in kernels from 1 to 2 dimensions or adding code to check results with the host. Parsing techniques were at a high level and could provide significant support for developers, especially during the prototyping of their future tools.

When source-to-source polyhedral compilers automatically generate OpenMP-optimized target code with parallelism and cache efficiency, chatbots can assist in generating the target code for CUDA or other formats like Kokkos and SyCL. Evaluating the performance and portability of high-level programming models necessitates extensive experimental study and is contingent on the target machine [40]. Therefore, automating compilers that can generate code for any of these frameworks is preferable. The establish-

ment of a new branch of optimizing compilers with practical implementation holds the potential to significantly reduce the time required for such evaluations.

In this article, we utilized the GPT-3.5 version, which proved to be adequate in terms of both code performance and correctness. This decision serves as an initial step for future research, laying the groundwork for exploration with subsequent versions of the GPT model, which exhibit considerable variation [41].

5. Conclusions

In this paper, we have presented the utilization of increasingly popular large language models (LLMs), specifically GPT-3.5, for the automatic generation of code based on the results of polyhedral compilers for graphics cards. This allows for saving programmers time, avoiding errors, and expanding the applicability of optimizing compilers.

We experimented with several approaches to train the chatbot for complete code generation. Unfortunately, there were instances where it tended to forget certain code generation guidelines. Therefore, in the future, we will consider adopting a divide-and-conquer strategy. Nevertheless, the results of the generated code are promising. We demonstrated that the generated code is correct and exhibits satisfactory performance. It is noteworthy that GPT-3.5, with examples at its disposal, is more adept at generating target codes for similar parallel loops.

GPT functions as a statistical model, producing outputs with a degree of randomness. While our primary goal of generating valid and efficient code has been achieved, we recognize the need to delve into the consistency of these outputs. We are committed to investigating this aspect and exploring potential strategies to address any observed inconsistencies. In the future, we also plan to validate the proposed approach with other polyhedral benchmarks using different chatbots and NPDP algorithms.

Despite concerns about the unforeseen impact of new artificial intelligence tools on various aspects of life, code generation automation is particularly attractive, especially for challenging tasks such as high-performance computing (HPC). Providers of parallel hardware and libraries introduce new interfaces each year, requiring the adaptation of existing programming practices. Large language models (LLMs) provide an opportunity for developers to keep up with the creation of efficient code for emerging architectures.

Author Contributions: Conceptualization and methodology, M.P. and M.G.; software, M.G.; validation, M.P.; formal analysis, M.G.; investigation, M.P. and M.G.; resources, M.P.; data curation, M.G.; writing—original draft preparation, M.P. and M.G.; writing—review and editing, M.P. and M.G.; visualization, M.P.; supervision, M.P. and M.G.; project administration, M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The input OpenMP and output CUDA codes of all compilers using chats to reproduce all the results described in this paper can be found at the following link: https://github.com/markpal/electronics_jan_2024, (accessed on 11 January 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

NPDP	Non-serial polyadic dynamic programming
GPT	Generative Pre-trained Transformer
LLM	Large language model
AI	Artificial Intelligence
ATF	Affine transformation framework
RNA	Ribonucleic Acid

HPC	High-performance computing
DAPT	Dependence Approximation Program Transformation
NN	Neural Network

References

1. Verdoolaege, S. Integer Set Library—Manual. 2011. Available online: www.kotnet.org/~skimo//isl/manual.pdf (accessed on 11 January 2024).
2. Bielecki, W.; Palkowski, M. A Parallelizing and Optimizing Compiler—TRACO. 2013. Available online: http://traco.sourceforge.net (accessed on 11 January 2024).
3. Bielecki, W.; Poliwoda, M. Automatic Parallel Tiled Code Generation Based on Dependence Approximation. In *Parallel Computing Technologies*; Malyshkin, V., Ed.; Springer: Cham, Switzerland, 2021; pp. 260–275.
4. Bondhugula, U.; Hartono, A.; Ramanujam, J.; Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008; Volume 43, pp. 101–113.
5. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 5.2. 2021. Available online: https://www.openmp.org/specifications (accessed on 22 October 2023).
6. Nvidia Corporation, CUDA Programming Guide 12.3. 2023. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 22 October 2023).
7. SYSCL 2020 Specification. 2023. Available online: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf (accessed on 26 November 2023).
8. The Kokkos C++ Performance Portability EcoSystem is a Solution for Writing Modern C++ Applications in a Hardware-Agnostic Way. 2023. Available online: https://kokkos.org (accessed on 26 November 2023).
9. Verdoolaege, S.; Carlos Juega, J.; Cohen, A.; Ignacio Gómez, J.; Tenllado, C.; Catthoor, F. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* **2013**, *9*, 1–23. [CrossRef]
10. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. In Proceedings of the Advances in Neural Information Processing Systems 33 (NeurIPS), Virtual, 6–12 December 2020. [CrossRef]
11. Google Bard. Available online: https://bard.google.com/ (accessed on 22 October 2023).
12. Introducing Llama 2, The Next Generation of Our Open Source Large Language Model. 2023. Available online: https://ai.meta.com/llama/ (accessed on 22 October 2023).
13. Writesonic. ChatSonic ChatBot. 2023. Available online: https://writesonic.com/chat (accessed on 22 October 2023).
14. Team, G.C. GitHub Copilot. An AI Pair Programmer for GitHub. 2022. Available online: https://copilot.github.com/ (accessed on 22 October 2023).
15. Tabnine Is an AI Assistant That Speeds Up Delivery and Keeps Your Code Safe. 2023. Available online: https://www.tabnine.com (accessed on 26 November 2023).
16. CodeT5 and CodeT5+. 2023. Available online: https://github.com/salesforce/CodeT5 (accessed on 26 November 2023).
17. Intro to Ghostwriter. 2023. Available online: https://replit.com/learn/intro-to-ghostwriter (accessed on 26 November 2023).
18. Welcome to Generative AI for Data. 2023. Available online: https://www.seek.ai (accessed on 26 November 2023).
19. We're Building the Only AI Coding Assistant that Knows Your Entire Codebase. 2023. Available online: https://about.sourcegraph.com/cody (accessed on 26 November 2023).
20. Build Together with AI. 2023. Available online: https://mutable.ai (accessed on 26 November 2023).
21. Large Models of Source Code. 2023. Available online: https://github.com/VHellendoorn/Code-LMs (accessed on 26 November 2023).
22. Nichols, D.; Marathe, A.; Menon, H.; Gamblin, T.; Bhatele, A. Modeling Parallel Programs using Large Language Models. *arXiv* **2023**, arXiv:2306.17281. Available online: http://arxiv.org/abs/2306.17281 (accessed on 22 October 2023).
23. Chen, L.; Lin, P.H.; Vanderbruggen, T.; Liao, C.; Emani, M.; de Supinski, B. LM4HPC: Towards Effective Language Model Application in High-Performance Computing. In *Lecture Notes in Computer Science*; Springer Nature: Cham, Switzerland, 2023; pp. 18–33. [CrossRef]
24. Godoy, W.; Valero-Lara, P.; Teranishi, K.; Balaprakash, P.; Vetter, J. Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In Proceedings of the 52nd International Conference on Parallel Processing Workshops, Salt Lake City, UT, USA, 7–10 August 2023; ICPP-W 2023; ACM: New York, NY, USA, 2023. [CrossRef]
25. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
26. Valero-Lara, P.; Huante, A.; Lail, M.A.; Godoy, W.F.; Teranishi, K.; Balaprakash, P.; Vetter, J.S. Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. *arXiv* **2023**, arXiv:2309.07103. [CrossRef]
27. Nussinov, R.; Pieczenik, G.; Griggs, J.R.; Kleitman, D.J. Algorithms for loop matchings. *SIAM J. Appl. Math.* **1978**, *35*, 68–82. [CrossRef]
28. Mullapudi, R.T.; Bondhugula, U. Tiling for Dynamic Scheduling. In Proceedings of the International Workshop on Polyhedral Compilation Techniques, Vienna, Austria, 20 January 2014.

29. Chowdhury, R.; Ganapathi, P.; Tschudi, S.; Tithi, J.J.; Bachmeier, C.; Leiserson, C.E.; Solar-Lezama, A.; Kuszmaul, B.C.; Tang, Y. Autogen: Automatic Discovery of Efficient Recursive Divide-8-Conquer Algorithms for Solving Dynamic Programming Problems. *ACM Trans. Parallel Comput.* **2017**, *4*, 1–30. [CrossRef]
30. Bielecki, W.; Blaszynski, P.; Poliwoda, M. 3D parallel tiled code implementing a modified Knuth's optimal binary search tree algorithm. *J. Comput. Sci.* **2021**, *48*, 101246. [CrossRef]
31. Wonnacott, D.; Jin, T.; Lake, A. Automatic tiling of "mostly-tileable" loop nests. In Proceedings of the IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques, Amsterdam, The Netherlands, 19–21 January 2015.
32. Palkowski, M.; Bielecki, W. Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing. *BMC Bioinform.* **2017**, *18*, 290. [CrossRef] [PubMed]
33. Bondhugula, U. Compiling affine loop nests for distributed-memory parallel architectures. In Proceedings of the SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, New York, NY, USA, 17 November 2013; SC '13; pp. 33:1–33:12. [CrossRef]
34. Tchendji, V.K.; Youmbi, F.I.K.; Djamegni, C.T.; Zeutouo, J.L. A Parallel Tiled and Sparsified Four-Russians Algorithm for Nussinov's RNA Folding. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2023**, *20*, 1795–1806. [CrossRef] [PubMed]
35. Li, J.; Ranka, S.; Sahni, S. Multicore and GPU algorithms for Nussinov RNA folding. *BMC Bioinform.* **2014**, *15*, S1. [CrossRef] [PubMed]
36. Lim, A.W.; Lam, M.S. Communication-free parallelization via affine transformations. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Portland, OR, USA, 17–21 January 1994; Springer: Berlin/Heidelberg, Germany, 1994; pp. 92–106.
37. NVIDIA Ampere Architecture Whitepaper. 2020. Available online: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (accessed on 22 October 2023).
38. Palkowski, M.; Bielecki, W. NPDP benchmark suite for the evaluation of the effectiveness of automatic optimizing compilers. *Parallel Comput.* **2023**, *116*, 103016. [CrossRef]
39. Palkowski, M.; Gruzewski, M. Time and Energy Benefits of Using Automatic Optimization Compilers for NPDP Tasks. *Electronics* **2023**, *12*, 3579. [CrossRef]
40. Godoy, W.F.; Valero-Lara, P.; Dettling, T.E.; Trefftz, C.; Jorquera, I.; Sheehy, T.; Miller, R.G.; Gonzalez-Tallada, M.; Vetter, J.S.; Churavy, V. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. In Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), St. Petersburg, FL, USA, 15–19 May 2023; pp. 373–382. [CrossRef]
41. Chen, L.; Zaharia, M.; Zou, J. How is ChatGPT's behavior changing over time? *arXiv* **2023**, arXiv:2307.09009. Available online: http://arxiv.org/abs/2307.09009 (accessed on 22 October 2023).