Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 14, Number 1, pages 58–67, January 2025

Towards Hybrid CPU-GPU Computing in a Backtracking-based Load Balancing Framework

Jing Xu
Graduate School of Informatics
Kyoto University
Kyoto, Japan
jing.xu.38s@st.kyoto-u.ac.jp

Tasuku Hiraishi
Department of Information and Computer Science,
Faculty of Engineering
Kyoto Tachibana University
Kyoto, Japan
hiraishi@tachibana-u.ac.jp

Zhengyang Bai
RIKEN Center for Computational Science
RIKEN
Tokyo, Japan
zhengyang.bai@riken.jp

Keiichiro Fukazawa
Academic Center for Computing and Media Studies
Kyoto University
Kyoto, Japan
fukazawa@media.kyoto-u.ac.jp

Masahiro Yasugi
Department of Computer Science and Networks
Kyushu Institute of Technology
Iizuka, Japan
yasugi@csn.kyutech.ac.jp

***Abstract*—General-purpose computing on graphics processing units (GPGPUs) has become increasingly prevalent, with hybrid CPU-GPU systems at the forefront of parallel computing. Dynamic load balancing is highly effective for maximizing CPU and GPU utilization in such environments. Backtracking-based load balancing, utilizing work-stealing, offers a promising strategy for task parallelism. However, Tascell, a task-parallel language implementing this mechanism, currently lacks GPU support, limiting its potential for hybrid CPU-GPU parallelism and constraining its application in computationally intensive tasks. In this paper, we propose enabling Tascell to fully utilize the computational power of CPU-GPU hybrid environments by writing both CPU-oriented and GPU-oriented code for workers to execute, allowing any worker to run GPU-oriented code based on task size and GPU availability. Using this technique, we implemented hybrid CPU-GPU programs for three applications using Tascell: recursive block matrix multiplication, 2D stencil computations and Mandelbrot Set calculations. The GPU-oriented code was implemented using OpenACC or the NVBLAS library. We conducted performance evaluations on both high-performance and workstation-grade CPU-GPU hybrid computing environments. Results demonstrated that in the workstation-grade environment, the hybrid approach outperformed both CPU-only and GPU-only configurations. Notably, hybrid CPU-GPU executions achieved performance improvements of up to 12.85% and 25.19% in 2D stencil applications compared to GPU-only and CPU-only executions, respectively. These findings provide valuable insights into effectively leveraging hybrid CPU-GPU systems within a backtracking-based load balancing framework.**

***Index Terms*—GPGPU, hybrid CPU-GPU, task parallel language, dynamic load balancing, 2D stencil computation, recursive block matrix multiplication, Mandelbrot Set calculations**

I. Introduction

As data-intensive applications continue to grow in complexity, the need for more powerful and efficient computing solutions is increasing, driving the development of hybrid CPU-GPU systems. These systems leverage the strengths of CPUs for handling complex, sequential tasks with intricate logic, and the capabilities of GPUs for executing highly parallel operations, providing enhanced computational power and efficiency to meet the demands of modern data-driven applications [1] [2] [3], such as scientific simulations, machine learning, and real-time data processing.

To fully utilize the resources of hybrid CPU-GPU systems, many language frameworks have been developed. CUDA and OpenCL are the primary tools for GPU programming. CUDA, developed by NVIDIA, provides direct access to GPU hardware and a broad ecosystem of development tools and libraries, making it very efficient for NVIDIA GPUs, but is specific to a certain type of hardware (such as NVIDIA GPUs), limiting the portability of the code [4]. The open standard OpenCL provides cross-platform compatibility across a variety of hardware, but performance inconsistencies and compatibility issues may still be encountered on different platforms [5]. Additionally, hybrid CPU-GPU systems, supported by frameworks like OpenACC [6], OpenMP [7], MPI [8], and SYCL [9], enable efficient parallelism across heterogeneous computing environments. High-level libraries and APIs, such as cuBLAS [10] and cuDNN [11] for CUDA, further simplify development and enhance performance. By leveraging

these tools and frameworks, developers can create robust and efficient applications to meet the demands of contemporary computing challenges, particularly for problems with regular structures. However, these programming models all face some common challenges due to the fundamental differences in architecture and performance characteristics of CPUs and GPUs. For example, the issue of dynamic task parallelism: these models are mainly designed for conventional applications and do not handle subtasks dynamically created at runtime well, often leading to workload imbalance [12]. In addition, irregular problem handling is challenging. Irregular computing tasks are difficult to predict in terms of task completion time and the number of new tasks created dynamically. Porting these tasks directly to the GPU is challenging because GPUs rely on uniform work distribution to fully exploit their data parallelism capabilities. Thus, an efficient load balancing mechanism is essential to fully utilize GPU resources.

Previous studies have explored a variety of approaches, both hardware and software-based, to address load imbalance in hybrid CPU-GPU parallel executions. Hardware-based solutions, such as using a host thread for task distribution or employing a shared task pool, have notable drawbacks. Host thread-based distribution can lead to underutilization of CPU resources [13], while shared task pools may suffer from overhead due to access violations and synchronization issues [14]. Software-based methods, including static load balancing, present challenges. Static approaches lack adaptability to varying workloads [15].

Tascell [16] is a promising task-parallel language, which employs a unique load balancing mechanism called backtracking-based load balancing. A Tascell worker performs sequential computation without creating any logical task until it receives a task request from another idle worker. To generate tasks of the largest possible granularity, when a worker receives a task request, it temporarily backtracks to a previous state before creating the task. This technique achieves efficient load balancing with minimal overhead for irregular applications. While the current implementation of Tascell does not support GPU computing, enabling workers that acquire substantially large tasks to use GPUs could potentially maximize the utilization of computational resources in CPU-GPU hybrid environments. This would be particularly beneficial for numerical computations involving irregularities.

To investigate the potential benefits of such functionality, we enabled Tascell programs to use GPUs as well as CPUs by incorporating both CPU-oriented and GPU-oriented code executions. We enabled Tascell workers to run either GPU-oriented or CPU-oriented code in parallel, based on task size and GPU availability. Additionally, using this technique, we developed hybrid CPU-GPU programs for three applications using Tascell: recursive block matrix multiplication, 2D stencil computations, and Mandelbrot set calculations [17]. Considering performance, compatibility, and implementation simplicity, GPU-oriented code was implemented using OpenACC or the NVBLAS library [18]. Performance evaluations were conducted on two CPU-GPU hybrid environments: a high-performance system with a 64-core AMD EPYC 7513 CPU and an NVIDIA A100 GPU, and a workstation-grade system with a 4-core Intel Xeon Gold 6140 CPU and an NVIDIA Quadro P4000 GPU. The contributions of this paper are summarized as follows.

- We propose a method for Tascell to fully utilize hybrid CPU-GPU systems. This is achieved by allowing workers to execute both CPU-oriented and GPU-oriented code. This method enables any worker to dynamically execute GPU-oriented tasks depending on task size and the availability of GPU resources, optimizing overall performance.
- We apply this method to implement hybrid CPU-GPU programs for three applications in Tascell: recursive block matrix multiplication, 2D stencil computations, and Mandelbrot Set calculations. The GPU-oriented code was implemented using OpenACC or NVBLAS library.
- Comprehensive performance evaluations were conducted on high-performance and workstation-grade CPU-GPU hybrid environments. The results showed that in the workstation-grade setup, the hybrid approach outperformed both CPU-only and GPU-only configurations. Specifically, hybrid CPU-GPU executions improved performance by up to 12.85% compared to GPU-only and 25.19% compared to CPU-only in 2D stencil applications. These findings provide valuable insights into the effective use of hybrid CPU-GPU systems within a backtracking-based load balancing framework.

The organization of this paper is as follows. In Section II, we provide an overview of the Tascell framework. Next, we present the details of the proposed methodology and implementation using Tascell framework on hybrid CPU-GPU systems in Section III. Section IV shows performance evaluation. We review related work in Section V. Finally, we draw conclusions and outline potential directions for future research in Section VI.

II. Tascell Framework

This section introduces Tascell framework, which we used in our implementation.

A. Dynamic Load Balancing in Tascell

Tascell [16] is a task parallel language that employs a *backtracking-based work stealing* strategy. Unlike fine-grained multi-threaded languages like Cilk [19] that use Lazy Task Creation (LTC) [20], a Tascell worker executes its tasks sequentially and only spawns a task in response to a work-stealing request from another worker. When encountering a spawnable task (e.g., in a parallel loop), the worker simply

```
double A[N][N];   //Matrix of size N x N
double B[N][N];   //Matrix of size N x N
double T_B[N][N]; //Transposed version of matrix B
double C[N][N];   //Matrix to store the result of A * B
int th;        //Threshold value
// Initialization
Initialization(A, B)
// Row and column indeices for the top-left corner
//of the matrix block
c_r = 0;
c_c = 0;
n = N;

// Recursive block matrix multiplication
block_recursive_mm(int c_r, int c_c, int n) {
    if (n <= th) {
        // Direct matrix multiplication
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                C[i + c_r][j + c_c] =
                    ∑_{k=0}^{N-1} A[i + c_r][k] * T_B[j + c_c][k];}
    } else {
        // Submatrix indices
        // Top-left
        block_recursive_mm(c_r, c_c, n/2);
        // Top-right
        block_recursive_mm(c_r, c_c + n/2, n/2);
        // Bottom-left
        block_recursive_mm(c_r + n/2, c_c, n/2);
        // Bottom-right
        block_recursive_mm(c_r + n/2, c_c + n/2, n/2); }
}
```

Fig. 1: Sequential C Program for Recursive Block Matrix Multiplication.

notes the opportunity and continues execution *as if on a fully sequential path*. Each worker maintains a workspace with the necessary data, updated at each step.

For efficient load balancing, idle workers request tasks from busy workers. An idle worker can send a task request to either a specific worker or any available one. When a worker (the victim) receives a request from another worker (the thief), it backtracks to the earliest parallelizable point, where the largest task can be spawned, and spawns a task *as if switching from sequential to parallel execution*. The victim then creates and initializes a new workspace for the task by copying its current workspace after backtracking. In short, when a worker receives a task request:

- it backtracks (returns to a previous state),
- it spawns a task (and changes the execution path to receive the result of the task),
- it returns from the backtracking (restore the time), and
- then it resumes its own task.

B. Example

We explain the work stealing mechanism in Tascell and Tascell programming using recursive block matrix multiplication as an example, which will also be used in the subsequent explanation of the proposed method and performance evaluation.

```
double A[N][N];
double B[N][N];
double T_B[N][N];
double C[N][N];
int th;
// The definitioin of a task named tmm
task tmm {
    in: int th;   // input
};
//The entry point of tmm
//The task object this is declared implicitly
task_exec tmm {
    block_para(this.param, this.n, this.i1, this.i2);
}
worker block_recursive_mm(int c_r, intc_c, int n){
    if(n ≤ th){
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                C[i+c_r][j+c_c] =
                    ∑_{k=0}^{N-1} A[i + c_r][k] * T_B[j + c_c][k];}
    else{
        // Compute the indices of the first element
        // of each submatrix of C
        int param[4][2] = {
            {c_r, c_c},
            {c_r, c_c + n/2},
            {c_r + n/2, c_c},
            {c_r + n/2, c_c + n/2}};
        block_para(param, n/2, 0, 4);}
}
worker block_para(int (*param)[2], int n, int i1, int i2)
{
    do_many for i from i1 to i2     // paralell loop
       block_recursive_mm(param[i][0], param[i][1], n/2);
    handles tmm from j1 to j2
        //put part (performed before sending a task)
        {   this.param = param;
            this.n = n;
            this.i1 = j1;
            this.i2 = j2;  }  // end of do_many
}
```

Fig. 2: Tascell Program for Block Recursive Matrix Multiplication (without GPU).

Figure 1 shows a C program that implements blocked matrix multiplication using recursive algorithm. The Tascell language is an extended C language. Figure 2 shows a parallelized Tascell program (Tascell extensions are underlined) for recursive blocked matrix multiplication based on the C code in Figure 1. We can write a worker program with the following constructs in Tascell, starting with an existing sequential program.

The top-level task declaration `task tmm {···};` defines the structure of a task object named `tmm`. Several fields with `in:` attribute are declared as the computing input[1].

`task_exec` defines the computation of a `tmm` task. In the body `block_para(this.param, this.n, this.i1, this.i2)`, the task object can be refereed to by the keyword `this`.

[1]Tascell provides the capability to specify task outputs using the `:out` attribute. However, this feature is not employed in this implementation of recursive block matrix multiplication because the computation results are written to the global variable `C`.

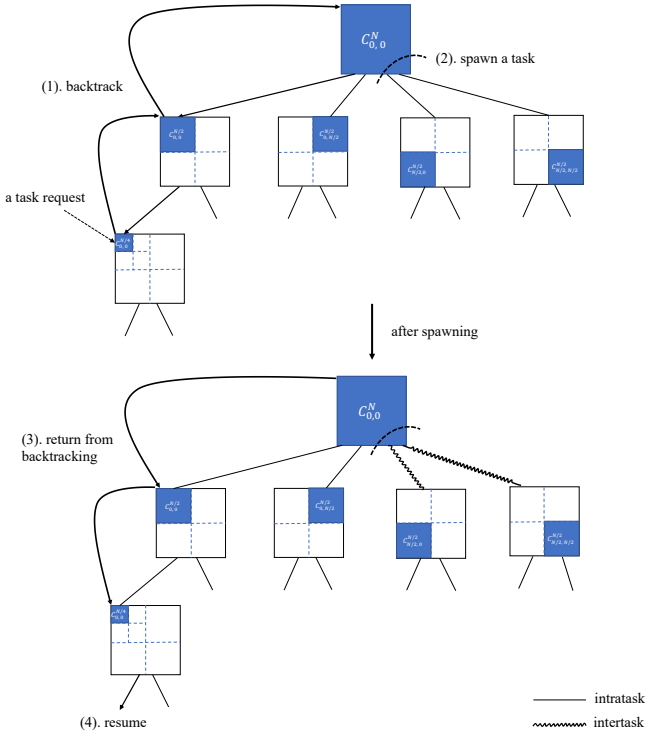


Fig. 3: Spawning a task lazily while computing $C^N_{0,0}$ in Tascell. When a worker detects a task request at $C^{N/4}_{0,0}$, it (1) backtracks to the oldest task-spawnable point $C^N_{0,0}$, (2) spawns a task for $C^{N/2}_{N/2,0}$ and $C^{N/2}_{N/2,N/2}$, (3) returns from backtracking and (4) resumes its own computation.

Worker functions in Tascell, defined with the keyword `worker`, can include task division constructs. In line 33 in Figure 2, the `do_many` statement is used to divide an iterative computation. This is syntactically denoted by:

`for(int` *identifier* `:` $expr_{\text{from}}$`,` $expr_{\text{to}}$`)`
$statement_{\text{body}}$
`handles` *task-name* `(int` $identifier_{\text{from}}$`, int` $identifier_{\text{to}}$`)`
{ $statement_{\text{put}}$ $statement_{\text{get}}$}.

This iterates $statement_{body}$ over integers from $expr_{from}$ (inclusive) to $expr_{to}$ (exclusive). During the execution of a loop, when the implicit task request handler (available during the iterative execution of $statement_{body}$) is triggered, it dynamically spawns a new task for the upper half of the remaining iterations. The specific range for this new task is indicated in $statement_{put}$ by $identifier_{from}$ and $identifier_{to}$.

Figure 3 illustrates how a task is spawned lazily in an execution of the program of Figure 2. In Figure 3, each tree node being computation step is represented by a corresponding symbol like $C^n_{i,j}$. Here, C is the result of $A \times B$. In the notation $C^n_{i,j}$, i and j are the indices of the first element of a submatrix of C, while n represents the size of this submatrix. In the upper part of the figure, suppose that a worker w_v executes the computation and $C^N_{0,0}$ is the oldest task spawnable point of w_v. When w_v receives a task request from another worker w_t at $C^{N/4}_{0,0}$,

- it *temporarily* backtracks to $C^N_{0,0}$,
- spawns a task to perform the computation of the right subtree, which corresponds to the remaining half of the unexecuted iterations of the parallel loop (in line 33 in Figure 2) at $C^N_{0,0}$, and sends the task to w_t, and
- returns from the backtracking and resumes its own task from $C^{N/4}_{0,0}$

Each task and its result are transmitted as a task object among workers. The structure of this object is defined in a Tascell program by the user. It can be transferred by passing the pointer in shared memory environments.

III. Methodology and Implementation

A. Applications

We employ three applications as examples to demonstrate the effective utilization of computational power in CPU-GPU hybrid environments in Tascell: recursive blocked matrix multiplication, 5-point 2D stencil computations, and Mandelbrot Set calculations.

1) Recursive Block Matrix Multiplication: Matrix multiplication is essential in many scientific applications and has recently been proposed as an alternative to convolution operations in Deep Neural Networks (DNNs) using the im2col transformation. Recursive blocked matrix multiplication, an optimization technique, can refine memory access patterns and reduce cache misses. Furthermore, high-performance linear algebra libraries, such as BLAS and cuBLAS, implement blocked matrix multiplication to efficiently solve linear systems, eigenvalue problems, and singular value decomposition on both CPUs and GPUs. These libraries utilize blocking techniques to optimize memory access and leverage advanced hardware features, making them crucial tools for large-scale computational tasks.

2) 5-point 2D Stencil Computations: Stencil computations are a type of numerical data processing technique used to update array or grid elements based on a fixed pattern known as a stencil. This technique involves an update rule that combines the values of neighboring elements according to the stencil pattern to compute a new value for each element in the grid. Typically applied to multi-dimensional arrays or grids—such as matrices in 2D problems—stencil computations are widely used in scientific and engineering applications to solve spatially dependent problems effectively. Given that stencil computations involve tight coupling of neighboring cells, where each cell's update depends on its neighbors, they are often memory-bound. This characteristic has led to exploration of various parallel architectures for efficient execution.

```
1  double A[N][N];
2  double B[N][N];
3  double T_B[N][N];
4  double C[N][N];
5  int th1, th2, th_cpu;
6  // The definitioin of a task named tmm
7  task tmm{
8      in: int n;   // input
9      in: int (*param)[2];;   // input
10     in: int i1;   // input
11     in: int i2;   // input
12 };
13 //The entry point of tmm
14 //The task object this is declared implicitly
15 task_exec tmm{
16     block_para(this.param, this.n, this.i1, this.i2);
17 }
18 worker block_recursive_mm(int c_r, intc_c, int n){
19     if((th1 ≤ n ≤ th2) &&
20                 (pthread_mutex_trylock(&myMutex) == 0)){
21         mm_sub_gpu(c_r, c_c, n);
22         pthread_mutex_unlock(&myMutex); }
23     elseif(n ≤ th_cpu){
24         mm_sub_cpu(c_r, c_c, n)};
25     else{
26         // Compute the indices of the first element
27         // of each submatrix of C
28         int param[4][2] = {
29             {c_r, c_c},
30             {c_r, c_c + n/2},
31             {c_r + n/2, c_c},
32             {c_r + n/2, c_c + n/2}};
33         block_para(param, n/2, 0, 4);}
34 }
35 worker block_para(int (*param)[2], int n, int i1, int i2)
36 {
37     do_many for i from i1 to i2     // parallel loop
38         block_recursive_mm(param[i][0],param[i][1],n/2);
39     handles tmm from j1 to j2
40         //put part (performed before sending a task)
41         {   this.param = param;
42             this.n = n;
43             this.i1 = j1;
44             this.i2 = j2;  }  // end of do_many
45 }
46 // GPU-oriented function
47 void mm_sub_gpu(int c_r, int c_c, int n) {
48     double *a = &A[c_r][0], *b = &T_B[c_c][0];
49     double *mm = (double *)malloc(sizeof(double) * n
50                                                   * n);
51     dgemm_("T", "N", &n, &n, &N_Z, 1.0, b, &N_Z, a,
52                                     &N_Z, 0.0, mm, &n);
53     // Update matrix C
54     for (i = 0; i < n; i++)
55         for (j = 0; j < n; j++)
56             C[i + c_r][j + c_c] = mm[i * n + j];
57     free(mm);
58 }
59 //CPU-oriented function
60 void mm_sub_cpu(int c_r, int c_c, int n);{
61     for (i = 0; i < n; i++)
62         for (j = 0; j < n; j++)
63             C[i+c_r][j+c_c] =
64                     ∑_{k=0}^{N-1} A[i + c_r][k] * T_B[j + c_c][k];
65 }
```

Fig. 4: Tascell Program for Recursive Block Matrix Multiplication (with GPU).

Currently, GPGPUs have demonstrated the high efficiency for stencil computations, thanks to their ability to handle parallel operations and manage memory effectively [21].

3) Mandelbrot Set Calculations: The Mandelbrot Set is a fascinating mathematical object, known for its intricate and beautiful fractal patterns [17]. The Mandelbrot Set M consists of all complex numbers c such that the sequence defined by the iterative function

$$f_c(z) = z^2 + c$$

starting from $z = 0$ does not diverge to infinity. In other words, c is in the Mandelbrot Set if the iteration

$$z_{n+1} = z_n^2 + c$$

remains bounded for all $n \geq 0$. The Mandelbrot Set is typically visualized in the complex plane. For each point c in the complex plane, the iterative process is performed to check if the magnitude of z_n grows without bound. Points where z_n remains bounded are colored differently to represent the set.

B. Proposed Implementation

To enable Tascell to effectively utilize the computational power of CPU-GPU hybrid environments, we wrote both CPU-oriented and GPU-oriented code, each of which any worker can execute selectively. A worker executes GPU-oriented code under two conditions: when its assigned task size is within a predetermined range, and when the GPU is not currently utilized by another worker. If either of these conditions is not met, the worker executes CPU-oriented code.

In the following sections, we detail the implementations for the three applications based on this approach.

1) Recursive Block Matrix Multiplication: Figure 4 shows an enhanced Tascell program based on the program in Figure 2, modified to utilize both CPU and GPU resources. In this program, The parameters `th1` and `th2` serve as thresholds governing the execution of GPU-oriented code, whereas `th_cpu` functions as a threshold controlling task granularity.

Let us consider the execution when a worker calls the `block_recursive_mm` function is called to compute the product of two submatrices of size `n`×`n`. As shown in lines 19 to 22, the worker calls the `mm_sub_gpu` function to perform matrix multiplication on GPU when the matrix dimension `n` satisfies `th1`≤`n`≤`th2` and the GPU is not being used by other workers. GPU exclusivity control is implemented using pthread mutex. Specifically, a worker calls `pthread_mutex_trylock` to check availability and acquire the GPU ownership if available, and calls `pthread_mutex_unlock` to release the ownership. The thresholds `th1` and `th2` should be set to optimize resource utilization. Specifically, `th1` is set to avoid the execution of small tasks on GPUs. The threshold `th2` is set to maintain an adequate computational load for other workers, to prevent CPU resources from becoming underutilized. Note that all workers are capable of performing GPU-oriented code as long as only one worker uses the GPU at a time.

When the condition th1≤n≤th2 is not met, the worker (recursively) divides the given computation into four sub-computations if n is larger than the threshold th_cpu. Otherwise, the worker executes the computation on a CPU core by invoking the mm_sub_cpu function. Here, th_cpu should be set considering the trade-off between task granularity and the overhead of task division.

The function mm_sub_cpu employs a naïve matrix multiplication algorithm, which uses triply nested for loops.

For GPU-oriented code, we used the NVBLAS library [18], which is provided by NVIDIA and offers GPU-optimized implementation of the Basic Linear Algebra Subprograms (BLAS). We used the NVBLAS library as a drop-in replacement for CPU-based BLAS libraries, which enables us to leverage GPU acceleration with minimal adjustments.

Additionally, NVBLAS allows configuration through environment variables and a configuration file, providing users with effective control over GPU offloading. Specifically, adding NVBLAS_GPU_DISABLED_DGEMM=1 in the nvblas.conf file disables the DGEMM operation from executing on the GPU, thus facilitating CPU-only execution. Conversely, by default, NVBLAS allocates the computational workload between the CPU and GPU based on hardware configuration and task size. In our implementation, however, we aim to maximize performance by directing larger tasks to the GPU-oriented functions. Consequently, NVBLAS consistently utilizes the GPU in the mm_sub_gpu function, thereby optimizing overall performance.

In the peformance evaluations in the next section, we compare the performance among CPU-only, GPU-only, and hybrid CPU-GPU executions. Depending on the execution settings, we adjusted the Tascell program and the environment variable NVBLAS_GPU_DISABLED_DGEMM as follows:

- For CPU-only executions, we activated NVBLAS_GPU_DISABLED_DGEMM and removed the pthread_mutex_trylock call in Figure 4.
- For GPU-only and hybrid CPU-GPU executions, we deactivated NVBLAS_GPU_DISABLED_DGEMM.

Notably, in CPU-only executions, all computations are executed in the mm_sub_gpu function using NVBLAS on CPU cores. In GPU-only executions, all computations are executed in the mm_sub_gpu function with NVBLAS accelerated by the GPU. In hybrid executions, large tasks are executed employing the mm_sub_gpu function with GPU-accelerated NVBLAS, while smaller tasks are handled in the mm_sub_cpu function with the naïve CPU algorithm.

2) 5-pooint 2D Stencil Computations and Mandelbrot Set Calculations: The code structures and workflows for 2D stencil computations and Mandelbrot Set calculations closely resemble those used for recursive block matrix multiplication.

```
1  // GPU Operations with OpenACC
2  void gpu_task(void Parameters) {
3      // In <caluse>, data regions to be copied to and
4      // from the GPU are specified.
5      #pragma acc data <clause>
6      {   //parallelize the nested loops on the GPU.
7          #pragma acc parallel loop
8          for (···) {
9              ···
10 }}}
11 // CPU Operations
12 void cpu_task(void Parameters) {
13     // Allow for SIMD vectorization of the inner loop.
14     #pragma omp simd
15     {   for (···){
16             ···
17 }}}
```

Fig. 5: CPU- and GPU-oriented Code Written in 2D Stencil Computations and Mandelbrot Set Calculations.

The both applications achieve parallelization by recursively dividing the space and assigning independent ranges to each worker. Similar to matrix multiplication, the choice between GPU execution and CPU execution is made based on the size of the assigned space.

The key difference is that 2D stencil computations and Mandelbrot Set calculations utilize a combination of OpenACC for GPU acceleration and SIMD (Single Instruction, Multiple Data) for CPU optimization. As shown in Figure 5, directive-based approaches such as OpenACC and SIMD provide compiler directives that facilitate the offloading of computations to accelerators and the vectorization of loops. This approach ensures efficient execution of these tasks across heterogeneous computing environments.

In the 5-point stencil computations, we adopted the temporal blocking technique for both CPU and GPU implementations. To illustrate this with the GPU computation, consider calculation on an area of size n^2 using the temporal blocking technique with the blocking depth D. A worker transfers an area of size $(n+2D)^2$ from host memory to the GPU, which includes the computation target and a halo region of width D. The GPU then executes D time steps of computation on the target area. Subsequently, the computation results are transferred back from the GPU to host memory. This approach not only enhances data locality but also reduces the frequency of data transfers between the host and GPU.

IV. Performance Evaluation

A. Evaluation Setup

This section provides the performance evaluation of the three applications: recursive block matrix multiplication, 2D stencil computations, and Mandelbrot Set calculations. We conducted experiments and analyzed the performance on both *high-performance* and *workstation-grade* platforms provided by Academic Center for Computing and Media Studies at

TABLE I: Hardware Specifications of the High-Performance Platform.

	DELL PowerEdge XE8545 (Gardenia)
CPU	AMD EPYC 7513, 32 cores 2.6 GHz × 2
Memory	DDR4-3200 512 GB × 16
GPU	NVIDIA A100 80 GB SXM × 4[a] + 80 GB × 4 of HBM2e (High Bandwidth Memory 2e)

[a]Although this hardware is equipped with four GPUs, we used only one GPU for the experiments in this study.

TABLE II: Hardware Specifications of the Workstation-Grade Platform.

	Visualization node 1 (gp - 001)
CPU	Intel Xeon Gold 6140 2.30 GHz (18 cores) × 2 # available cores for each user is limited to 4.
Memory	DDR4-2666 720 GB
GPU	NVIDIA Quadro P4000 (8 GB GDDR5 SDRAM) × 2 # available GPUs for each user is limited to 1.

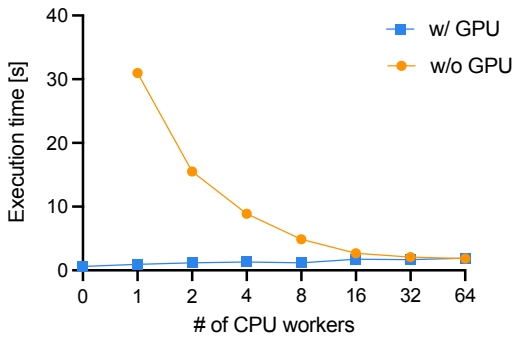
TABLE III: Software Environment (common to both evaluation environments).

	Software on hybrid CPU-GPU environment
OS	Red Hat Enterprise Linux 8
Compiler	Tascell: Tascell Compiler version of Jan. 21, 2019 + NVC 23.9 with -O3 -acc=gpu -mp -march=native options + CL-SC implementation of nested functions [22]
OpenACC	OpenACC 2.6
OpenMP	OpenMP 5.0
CUDA	CUDA 12.2.2

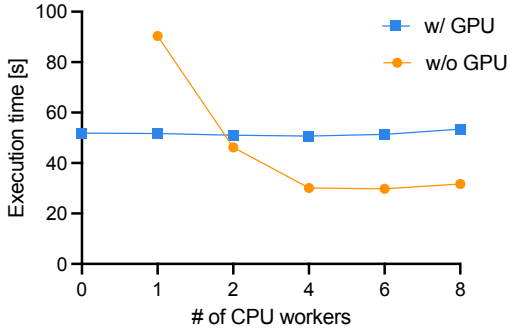
Kyoto University. The high-performance platform is a single node of the Gardenia supercomputer, while the workstation-grade platform is provided as a visualization server. The details of the platforms are summarized in Tables I and II. The software environment is shown in Table III. All experiments were performed in a single node. The detailed parameters setting and description for each application have been shown in Table IV.

B. Evaluation Results

In the performance results shown in Figures 6, 7 and 8, *# of CPU workers* indicates the number of workers that can use CPU cores simultaneously. The legends *w/ GPU* and *w/o GPU* denotes whether an additional worker for using a GPU alongside those using CPU cores. For instance, in Figure 6a, the line marked with *w/ GPU* indicates the performance when a GPU is used. Note that when the number of CPU workers is 0, it indicates an execution with only 1 worker, which means a GPU-only execution. When *# of CPU workers* is 8, it indicates an execution with 9 workers, where any worker has the potential to use the GPU but only one worker is permitted to do so at a time, while the remaining 8 workers can use CPU cores for their computations.



(a) Performance of recursive block matrix multiplication with and without GPU on the high-performance platform.



(b) Performance of recursive block matrix multiplication with and without GPU on the workstation-grade platform.

Fig. 6: Evaluation Results of Recursive Block Matrix Multiplication.

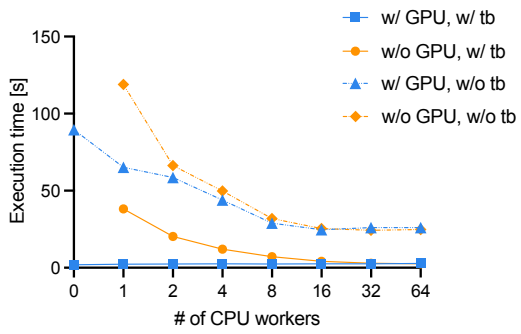
We executed the programs five times for each measurement setting and present the result whose execution time is the median.

1) Recursive Block Matrix Multiplication: From the Figure 6a, we can observe that the execution time for CPU-only tasks with a single worker is 30.98 s. By increasing the number of workers to 64, the execution time decreases to 1.843 s, achieving a speedup factor of 16.81 on the high-performance platform. In the GPU-only execution (0 CPU workers and w/ GPU), the computation will be offloaded entirely to the GPU. Its execution time is 0.6245 s, which surpasses the best execution time achieved with a hybrid approach, which is 0.9540 s using one CPU worker.

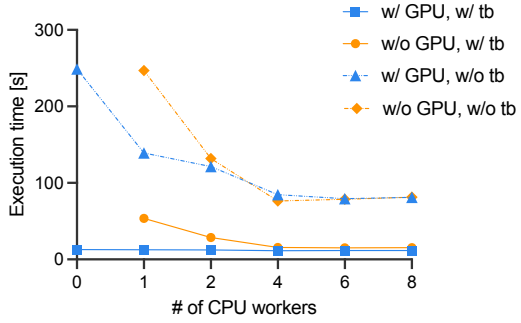
As shown in Figure 6b, on the workstation-grade platform, speedups of CPU-only execution program improve with the number of workers up to 6, because the number of available CPU cores for each user is limited to 4. The execution time improved from 90.41 s with one worker to 29.83 s with 6 workers, resulting in a speedup of 3.03 times. In the executions with GPU, the execution time of the hybrid execution with 4 CPU workers is 50.72 s, which is 1.873 % faster than the execution time of the GPU-only execution (51.68 s). Additionally,

TABLE IV: Parameters for Different Applications on both platform

Application	Parameter	Description
Recursive Block Matrix Multiplication	Matrix size N	8192^2 on the high-performance platform; 16384^2 on the workstation-grade platform.
	`th1`, `th2`, `th_cpu`	`th2` $= N$ for GPU-only executions and `th2` $= N/2$ for hybrid executions. For other parameters, we conducted optimal parameter search and took the best setting for each configuration. For instance, we used (`th1`, `th_cpu`) $= (8, 4)$ for executions with GPU and 64 CPU workers on the high performance platform.
5-point 2D Stencil Computations	Grid size N	8192^2 cells, each containing a double-precision floating-point value
	# total time steps IT	1000
	Boundary conditions	Dirichlet boundary condition
	`th1`, `th2`, `th_cpu`, D	We tuned these parameters as in matrix multiplication. For instance, we used (`th1`, `th2`, `th_cpu`, D) $= (0, 4096, 1024, 250)$ for executions with GPU and 64 CPU workers on the high performance platform.
Mandelbrot Set Calculations	*width*×*height*	8000×8000
	Maximum # iterations IT	50000 iterations for each cell
	Region of interest r	The region $[-2.5, 1.5] \times [-2.0, 2.0]$ on the complex plane.
	`th1`, `th2`, `th_cpu`	We tuned these parameters as in matrix multiplication. For instance, we used (`th1`, `th2`, `th_cpu`) $= (4, 2000, 4)$ for executions with GPU and 64 CPU workers.



(a) Performance of 2D stencil computation on the high-performance platform.



(b) Performance of 2D stencil computation on the workstation-grade platform.

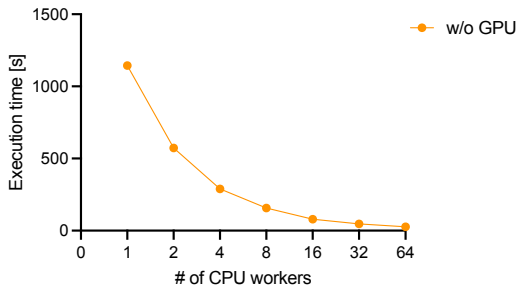
Fig. 7: Evaluation Results of 2D Stencil Computations.

CPU-only execution with 6 workers outperforms both hybrid and GPU-only executions. This may be because, in CPU-only execution, all workers can execute DGEMM operations in parallel on the CPUs, whereas in hybrid execution, only one worker can access the GPU to execute DGEMM operations while the other workers use a less efficient naïve algorithm.

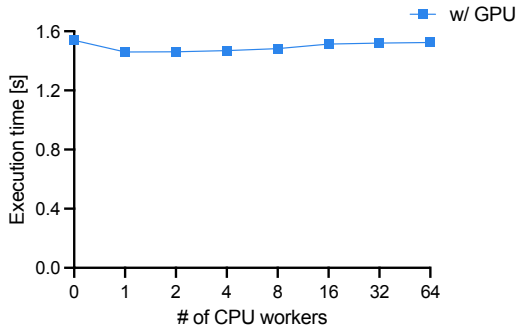
2) 5-Point 2D Stencil Computations: Figure 7 presents comparisons of performance both with and without GPU utilization, as well as with and without the application of the temporal blocking technique (denoted as w/ and w/o tb in the figures). On both platforms, temporal blocking significantly improves the performance. On the high-performance platform, we achieved a 47.15-fold speedup in GPU-only executions, a 28.39-fold speedup in hybrid executions using 2 CPU workers, and a 9.76-fold speedup in CPU-only executions with 64 workers. On the workstation-grade platform, we achieved a 19.18-fold speedup in GPU-only executions, 10.93-fold speedup in hybrid executions involving 2 CPU workers, and 5.31-fold speedups in CPU-only executions with 64 workers. Moreover, Figure 7a demonstrates that the performance of CPU-only execution improves significantly with up to 16 workers, beyond which the performance gain diminishes.

In the two graphs of Figure 7, we focus on the implementations using temporal blocking to compare the performance with and without GPU utilization. On the high-performance platform, the performance in CPU-only executions rise with the number of workers, reaching a 15.04-fold speedup with 64 workers, where the execution time is 2.542 s. In comparison, the execution time of GPU-only execution is 1.898 s, which is approximately 17.35 % faster than the best hybrid execution time (2.297 s with 2 CPU workers) and also 25.3 % faster than the CPU-only execution with 64 workers. On the workstation-grade platform, compared to the single CPU-core execution, we obtained a 3.54-fold speedup in the CPU-only execution with 6 workers, where the execution time is 15.11 s. The best hybrid execution time is 11.31 s, which is achieved with 5 CPU workers. The GPU-only execution time is 12.98 s. Thus, the hybrid CPU-GPU execution achieved performance gains of 12.85 % over the GPU-only execution and 25.19 % over the CPU-only executions for the 2D stencil computation.

3) Mandelbrot Set Calculations: We conducted the performance evaluations only on the high-performance platform for



(a) Performance for Mandelbrot Set Calculations in CPU-only executions.



(b) Performance for Mandelbrot Set Calculations in GPU-only and hybrid executions.

Fig. 8: Evaluation results for Mandelbrot Set Calculations on the high-performance platform.

Mandelbrot Set calculations. Figure 8a shows the performance of CPU-only executions. The execution time decreases notably from 1144 s to 27.38 s as CPU workers increase from 1 to 64, achieving a 41.8-fold speedup. The graph confirms that this speedup demonstrates effective parallelization, with substantial computation time reduction and efficient scaling in a CPU-only environment. As depicted in Figure 8b, the execution time with GPU, including hybrid configurations, is considerably lower than that for the CPU-only executions. The execution time of the GPU-only execution is 1.539 s. In the hybrid executions, while the addition of CPU workers leads to a slight increase in execution times, the overall performance still generally exceeds that of the GPU-only execution. Specifically, when one CPU worker is employed, the hybrid execution achieved an execution time of 1.460 s, which is 5.170 % faster than the GPU-only execution and 94.67 % faster than the best CPU-only execution.

C. Analysis and Discussion

1) Effectiveness of GPU Utilization: Adding a GPU significantly cuts execution time across various applications. This is particularly clear in the stencil computations and the Mandelbrot Set calculations, as shown in Figures 7 and 8, where parallelism can be effectively exploited by the GPU. However, recursive block matrix multiplication on the workstation-grade platform is an exception, since all workers in CPU-only execution benefit from the NVBLAS library, as noted in Section III-B1.

2) Hybrid Execution Efficiency: Combining CPU and GPU resources through hybrid execution methods often leads to the best performance in certain environments, particularly for irregular applications. The stencil application, for example, performs best with hybrid execution on the workstation-grade platform. For the Mandelbrot Set calculations, which is considered as an irregular application, adding CPU workers to GPU computation maximizes the utilization of computational resources and achieves the best results. This demonstrates that effective workload distribution between the GPU and CPU on hybrid environments is achieved on Tascell framework.

3) Impact of Hardware Environment and Application Characteristics: In workstation-grade heterogeneous systems with GPUs of lower performance, CPU-GPU hybrid executions can effectively utilize the CPU's computational power to improve performance. In high-performance heterogeneous systems with GPUs of higher performance, the advantages of hybrid execution may be overshadowed by the efficiency of single-GPU execution. Nevertheless, for irregular computations like the Mandelbrot Set calculations, hybrid executions can still achieve superior performance with strong GPUs. Overall, while GPU-only executions are often preferable on high-performance platforms, hybrid executions remain valuable for specific systems or applications that benefit from the combined strengths of CPUs and GPUs.

V. Related Work

In heterogeneous scheduling, a common approach is the use of task schedulers, where the problem is divided into tasks represented as a directed acyclic graph (DAG) based on their dependencies. This DAG is passed to a runtime system that monitors performance and assigns tasks to different processing units. Several advanced systems use this method. For example, StarPU [23] is a task-based runtime system that schedules tasks across heterogeneous resources, including CPUs and GPUs. It utilizes a shared task pool, where tasks are accessible by both CPUs and GPUs. Systems like Halide [24] and Legion [25] also follow this approach, with Halide managing tasks via the host thread and Legion using a shared task pool. These systems have been applied to practical problems like image processing, fluid simulation, etc.

Previous research on hybrid CPU-GPU computing has explored various frameworks and techniques to improve performance and resource utilization. For example, [26] reviewed cooperative CPU-GPU computing systems, highlighting task allocation schemes and runtimes that enhance collaborative execution but also pointed out issues like workload imbalance and performance overhead. In another study, [27] focused

on hybrid CPU-GPU systems for applications like Sparse Matrix-Vector Multiplication (SpMV), optimizing for irregular workloads in heterogeneous systems. Despite these advances, many frameworks struggle to manage dynamic task parallelism effectively, particularly in hybrid systems. Distributing tasks generated at runtime and handling irregular workloads is challenging due to GPUs' preference for uniform work distribution, which can cause inefficiencies with dynamic or irregular tasks [28].

VI. Conclusions

In this paper, we propose a method in Tascell that optimizes the use of hybrid CPU-GPU systems by enabling workers to handle both CPU-oriented and GPU-oriented tasks, dynamically allocating GPU tasks based on size and availability. This approach is applied to three applications: recursive block matrix multiplication, 2D stencil computations, and Mandelbrot Set calculations. Our evaluation shows that the hybrid approach offers significant performance gains on the workstation-grade CPU-GPU platform, achieving up to 12.85 % and 25.19 % improvements for 2D stencil computations over GPU-only and CPU-only executions, respectively. For Mandelbrot Set calculations, which involve irregular computations, the hybrid method leads to up to 5.17 % and 94.67 % gains over GPU-only and CPU-only executions, respectively, on the high-performance platform.

This proposal investigates the feasibility and performance of combining GPU parallel computing power with Tascell's dynamic load balancing to fully utilize the resources of hybrid CPU-GPU systems. It provides a reference for efficiently parallelizing large-scale, complex, and irregular applications. This study successfully demonstrates Tascell's potential for fully utilizing hybrid systems. Future research will focus on integrating GPU support directly into the Tascell framework to further enhance its capabilities and optimize GPU resource utilization. Exploring applications where the proposed approach works more effectively is also future work.

References

[1] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[2] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.

[3] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[4] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[5] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[6] Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, Barbara Cantalupo, Carlo Cavazzoni, and Maurizio Drocco. Practical parallelization of scientific applications with openmp, openacc and mpi. *Journal of parallel and distributed computing*, 157:13–29, 2021.

[7] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[8] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[9] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[10] NVIDIA Corporation. cublas library. https://docs.nvidia.com/cuda/cublas/, 2023. Accessed: 2024-08-05.

[11] NVIDIA Corporation. cudnn library. https://docs.nvidia.com/cuda/cudnn/, 2023. Accessed: 2024-08-05.

[12] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[13] Juan Fang, Jiaxing Zhang, Shuaibing Lu, and Hui Zhao. Exploration on task scheduling strategy for cpu-gpu heterogeneous computing system. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 306–311. IEEE, 2020.

[14] Shuai Zhang, Tao Li, Qiankun Dong, Xuechen Liu, and Yulu Yang. Cpu-assisted gpu thread pool model for dynamic task parallelism. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 135–140. IEEE, 2015.

[15] Nadeem Shah and Mohammed Farik. Static load balancing algorithms in cloud computing: challenges & solutions. *International Journal of Scientific & Technology Research*, 4(10):365–367, 2015.

[16] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based Load Balancing. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, *PPoPP '09*, pages 55–64, 2009.

[17] Bodil Branner. The mandelbrot set. In *Proc. symp. appl. math*, volume 39, pages 75–105, 1989.

[18] NVIDIA Corporation. Nvblas library. https://docs.nvidia.com/cuda/nvblas/, 2024. Accessed: 2024-08-05.

[19] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[20] Eric Mohr, David A Kranz, and Robert H Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 185–197, 1990.

[21] Andreas Schäfer and Dietmar Fey. High performance stencil code algorithms for gpgpus. *Procedia Computer Science*, 4:2027–2036, 2011.

[22] Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. A Transformation-Based Implementation of Lightweight Nested Functions. *IPSJ Trans. Programming*, 47(SIG 6(PRO 29)):50–67, 2006.

[23] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*, pages 863–874. Springer, 2009.

[24] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[25] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[26] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[27] Wangdong Yang, Kenli Li, and Keqin Li. A hybrid computing method of spmv on cpu–gpu heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 104:49–60, 2017.

[28] John Owens. Gpu architecture overview. In *ACM SIGGRAPH 2007 courses*, pages 2–es. Association for Computing Machinery, 2007.