Louisiana State University
LSU Digital Commons

LSU Master's Theses | Graduate School

2014

Performance Comparison Between Patus and Pluto Compilers on Stencils

Pratik Prabhu Hanagodimath
Louisiana State University and Agricultural and Mechanical College, phanag1@tigers.lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Hanagodimath, Pratik Prabhu, "Performance Comparison Between Patus and Pluto Compilers on Stencils" (2014). *LSU Master's Theses*. 2636.
https://digitalcommons.lsu.edu/gradschool_theses/2636

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

PERFORMANCE COMPARISON BETWEEN PATUS AND PLUTO COMPILERS ON STENCILS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Electrical and Computer Engineering

by
Pratik Hanagodimath
B.E., Visvesvaraya Technological University, 2010
May 2014

Acknowledgements

I thank Dr J Ramanujam for giving me an opportunity to research in the field of compiler optimization. I thank him for his guidance and providing moral and financial support towards my completion of master's program. I also thank Sameer Abu Asal for his extensive support at every stage in my research work. His guidance was truly helpful. I thank Ajay and Abhilash for helping me with troubleshooting linux related problems. I thank all my friends and colleagues who have helped me directly and indirectly in this tenure. It was truly a good learning experience.

Table of Contents

Abstract

The main aim of this research is to compare the performance of two compilers namely patus and pluto on stencils. In order to observe the performances of these two compilers, two coding styles have been implemented. Those are:

- Jacobi style
- Seidel style

The report discusses and reasons the performances of both the compilers on each style of coding. Many performance graphs have been plotted to illustrate the difference in performance.

Pluto compiler uses an optimization technique called tiling. Tiling is an important technique which transforms the code for better data locality and parallelism. To achieve this, appropriate tiling size and tiling shape must be considered. Therefore, we use the following to get the best tiling performance:

- Pluto tiling: Uses rectangular shaped tiles to transform the code for pipelined execution of tiles
- Sica tiling: This is an extension of pluto tiling to find better tile sizes
- Diamond tiling: Uses diamond shaped tiles to transform the code for concurrent execution of tiles

In order to find better tile sizes for pluto and diamond tiling, a shell script is used as a tuner which changes the tile sizes from 8 to 512 to find an optimal tile size value.

Chapter 1 Introduction

1.1 Introduction

A stencil is a class of an iterative kernel in which each point in the kernel is updated by a fixed pattern of memory access and computations. Despite the stencil kernels simplicity in updating the points, these stencils achieve a very low fraction of peak performance. A stencils computation performs much iteration over the spatial domain in order to update a time or a non-time dependent point in the kernel. Due to this, the computations will take up several core hours on supercomputers. Because of their fixed pattern of computation stencils always exhibit parallelism. Exploiting any of these parallelism leads to an optimized code with significant performance in terms of execution speedup and data locality.

Stencils are used in a wide range of applications ranging from many scientific to engineering sectors. In this report two image processing stencils namely blur and edge stencils, two partial differential equation solvers namely laplacian and seidel stencils, one cellular automata namely game-of-life stencil and one wave propagation namely wave stencils are considered for experimentation. These stencils have intense arithmetic computations.

In this report, two compilers namely pluto and patus have been selected to observe the performances of these two compilers on the stencil. Each compiler optimizes the code in its unique way by exploiting the parallelism in the stencils. Our experiments on stencils are conducted using two coding styles:

- Jacobi style
- Seidel style

Performance illustrations are done using graphs such as execution time vs grid size, Gflops vs grid size and speed-up charts. A brief reasoning on the compiler performance is also discussed at the end of each experiment.

1.2 Literature review

The authors Matthias, Schenk and Burkhart present a code generation and auto tuning framework for parallel stencil computations (Christen, Schenk, & Burkhart, 2011). This framework is known as Patus which stands for Parallel Auto Tuned Stencil. In this paper, the authors present their strategies for optimizing stencil computations. And discuss their results with appropriate tables and graphs. Uday designed and implemented a fully automatic polyhedral source to source transformation framework that can optimize regular programs (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008). This paper discusses a model driven automatic transformation in polyhedral model and presents approach in an end to end integer linear optimization framework that is capable of find good tiling for data locality and parallelism (Bondhugula, Hartono, Ramanujam, & Sadayappan, 2008). One of the important necessities to tile the code is to have an appropriate tile size and tile shape. The authors Uday, Bandishti and Pananilath present a new tiling technique that enable tile-wise concurrent startup to maximize parallelism and provides load balancing (Bondhugula, Pananilath, & Bandishti, 2012). In this paper, the authors present their implementation on the new tiling technique called diamond tiling and discuss their results on its performance. As mentioned earlier, choosing a tile size places an important role in using tiling optimization technique. The authors Soddemann et.al present a new hardware-aware and adaptive loop tiling approach that is based on the polyhedral transformation (Dustin, Thomas, Michael, & Sven, 2013). In this paper, the authors claim that their model chooses a better tile size and also improves auto-vectorization.

Considerable research is done in finding a framework that provides a good optimized code for general stencils and applications. Much research is also done in the use of tiling optimization technique and to find efficient ways to come up with better tile shapes and sizes. However, there is no available research that compares the two framework, patus and pluto, for their efficiency on optimizing codes. This report aims to compare the performances of patus and pluto compiler on stencils.

Chapter 2 Basic optimization techniques

2.1 Introduction

In this section, the report provides information regarding the basic optimization techniques. These techniques are used by the compiler to generate optimized code. Section 2.2 explains the different types of data dependencies and touch up on what a dependency distance and direction vector is. Section 2.3 illustrates different transformation techniques with an example.

2.2 Data flow analysis

- Flow dependency

  A dependency is termed a flow dependency if a memory location is written in one statement and read in subsequent iterations. An example shown below illustrates flow dependency due to access to array A:

```
for ( i ….)  {
        A[i] = b[i] + c[i]
        D[i] = A[i]
}
```

- Anti-dependency

  Occurs when a memory location is read in some statement and in later iterations some value is written to the same memory access. An example below illustrates anti-dependence due to array b.

```
 for ( i ….)  {
        A[i] = b[i] + c[i]
        b[i] = d[i]
}
```

- Output dependency

  Occurs when a memory location which is written in one statement; and in later iterations the same memory location is written again. Array D in the example shown below illustrates output dependency.

```
For ( i ….)  {
        D[i] = b[i] + c[i]
        D[i] = A[i]
}
```

- Input dependency

  Occurs when a memory location is read in two different iterations. Array b in the example shown below illustrates input dependency.

```
for ( i ….)  {// iterations
        A[i] = b[i] + c[i]
        D[i] = b[i]
}
```

2.2.1 Dependency distance and direction vectors

A dependency distance vector represents the dependency distance between the memory access of source iteration and sink iteration. Direction vector represents the sign { +, 0, - } of the distance vector. Consider the following example:

```
For ( i … ) {
        For ( j …) {
                a[i][j] = a[i][j] + a[i-1][j+1] + a[i][j-1];
        }
}
```

The dependency distance vector which is calculated as sink instance minus the source instance; for the above example distance vector due to accesses to array "a" is as shown:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

The direction vector for the example is as shown:

$$\begin{pmatrix} 0 & + & 0 \\ 0 & - & + \end{pmatrix}$$

It is very important to note that the first non-negative component should always be a 0 or positive, it must never be negative. The level at which the first non-negative component occurs defines the dependency for that level. For instance, there is a level 1 and level 2 dependencies for the above example.

2.3 Transformation techniques

- Skewing

  Loop skewing is a technique to change distance vectors to a form where further transformations can be enabled such as loop interchange or tiling. Skewing is always done to the inner loop with respect to the outer loop. Skewing the inner loop with respect to outer loop by a factor ‘f’ changes a distance vector from $\begin{pmatrix} d1 \\ d2 \end{pmatrix}$ to $\begin{pmatrix} d1 \\ (f \times d1) + d2 \end{pmatrix}$. Skewing the loops is always legal because it does not change the order of execution of iterations. Consider the following example:

```
for ( i … ) {
        for ( j …) {
                a[i][j] =  a[i-1][j+1] + a[i][j-1];
        }
}
```

  The distance vector for the example is $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$. Here dependencies are carried by both the loops. If the inner loop is skewed by a factor 2 with respect to the outer loop, then the dependency distance vector will be $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. This enables loop interchange technique which will make the outer loop carry all the dependencies and inner loop parallel.

- Loop interchange

  This technique as the name says, loops can be interchanged to make a loop level carry all the dependencies. This technique is often used after loop skewing. Consider the pervious example where after skewing the loops the new dependency distance vector is $\begin{pmatrix}1 & 0\\1 & 1\end{pmatrix}$. Now, if loop interchange is applied, the dependency distance vector would be $\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$. We can see that the dependencies are carried by the outer loop and inner loop is parallel. One should be careful while applying loop interchange; the outer most loop after the interchange must always be positive or 0. It is not a valid transformation if the first level component is a negative value.

- Vectorization

  Vectorization is an optimization technique to execute the statements in the loop on different available processors in the same time step. Vectorization is only possible if there exists no dependency loops between one statement to another statement. For example, consider the code shown below, vectorization is not possible because there exists a dependency cycle between statements S1 and S2.

```
for ( i …) {
        for (j … ) {
                S1: a[i][j] = b[i][j];
                S2: b[i+1][j] = a[i][j]
        }
}
```

  Consider the following example, In this example the inner loop “j” is parallel as is does not carry any dependencies and is vectorizable; and all the dependencies is carried by the outer loop “i”.

```
for ( i …) {
        for (j … ) {
                S1: a[i][j] = a[i-1][j] + a[i-2][j-1]
        }
}
```

In the vectorized form the above code look like:

```
for ( i …) {
        S1: a[i][1 : N] = a[i-1][1 : N] + a[i-2][1 : N].
        }
```

- Loop tiling

Tiling is an optimization technique that partitions the iteration space of the loops into smaller tiles for improving data locality and to incur smaller execution overhead. A tile is defined by a set of boundaries regularly spaced apart. A tile is valid as long as there are no negative dependencies in it. Consider the code shown below:

```
for ( i=0; i<N; i++) {
        for ( j=0; j<N; j++) {
                S1: a[i][j] = a[i-1][j] + a[i-2][j-1]
        }
}
```

The general form of tiling with a tile size of “T” would look like:

```
for ( ii=0; ii<N; ii+T ){    // In steps of T tiles
        for (jj=0; jj<N; jj+T){     //In steps of T tiles
                for (i=ii; i<ii+T; i++) {
                        for (j=jj; j<jj+T; j++){
                                S1: a[i][j] = a[i-1][j] + a[i-2][j-1]
                        }
                }
        }
}
```

Chapter 3 Brief overview of Patus compiler

3.1 Patus

Patus stands for Parallel Auto-Tuned Stencil. It is an auto tuning as well as a code generation tool for the class of stencils (Christen, Schenk, & Burkhart, 2011). The main goal of patus compilers is to accept a DSL (domain specific language) which contains a stencil specification and turn this DSL to a high performance C code which is optimized. The resultant C code is generated considering the architecture of the system. Patus targets the architectures such as many and multi core processors. As mentioned earlier, patus takes in a DSL input for producing an optimized C code. This DSL allows the programmer to express the stencil computations in a comprehensive way independently of hardware architecture specific details (Christen, Schenk, & Burkhart, 2011). Expressing the stencil computations as a DSL also adds to the programmer's productivity and the programmer need not concern with hardware specific details and low level programming issues. The other uses of DSL are its portability and re-use. Once the DSL is specified, it can be re-used on different platforms without having to change it as required by architecture. Hence the code can be re-used and is portable across different platforms.

3.2 Features of Patus compiler

Patus compiler has its own features through which it generates an optimized C code. Its features are:

1) Cache blocking strategy
2) Auto tuner

3.2.1 Cache blocking strategy

The DSL when compiled with patus uses cache blocking strategy to generate the optimized C code. A basic structure of this strategy is as shown (Christen, Schenk, & Burkhart, 2011):

```
Strategy cacheblocking ( … ) // parameters like grid size, blocks and chunk are passed here.
{
        //iterates over time specified
        For t = ….
        {
                // iterates over subdomain of the grid
                For subdomain v (cb) in u(:; t)
                   Parallel schedule chunck
                {
                        // calculates the stencil for each point in subdomain of the grid
                        For point p in v(:; t)
                                v[p; t+1] = stencil (v[p; t]);
                }
        }
}
```

The parameters like grid size, cache blocks and chunks are passed as arguments to this strategy. The cache block and chunk parameter are link to auto tuner to find an optimal value for these parameters. A brief explanation of auto tuner is given in the next paragraph. This strategy iterates over the time specified for all subdomains of size "v" as shown in the strategy. The value of size "v" depends on "cb" which is obtained from auto tuner. The "parallel" key word specifies parallel execution of blocks "v" dealt to worker threads. The "chunk" and "schedule" keywords specify how many consecutive blocks one thread is given. At the end, the stencils computation is done for "p" points in the subdomain of "v" block.

3.2.2 Auto tuner

Auto tuner is a tuning feature of patus that tunes to find the best configuration parameters for cache blocking strategy. As explained in the previous paragraph, the auto tuner is passed with "cb", grid sizes and chunk information. The auto tuner runs the stencil computations with these parameters and produces the output. Many sets of values are tested by the auto tuner and the configuration that ran the best is given as output at the end of testing all the values in its set.

3.3 Optimizations techniques used by Patus

Patus uses cache blocking strategy to find the best configuration for blocking the grid size. This is the same as the tiling optimization technique. Patus also uses loop unrolling technique to unroll the inner most loop and collapses the inner loop to its immediate outer loop. After loop unrolling technique is applied, patus uses vectorization technique to vectorize the inner most loop.

3.4 A walk through DSL example

```
 stencil example ( float grid U )
{
        iterate while t < 1;
        domainsize = ( 3 .. height-3, 3 .. width-3 );
        initial {
                U[ x, y; t ] = x*x + y*y;
                U[ x, y; t+1] = 0;
        };
        operation {
                U[ x, y; t+1] = U[ x, y; t] + U[ x, y-1; t] + U[ x-1, y; t] + …….
        };
}
```

- The “stencil” specifies the stencil of “example” in the domain specific language (DSL). This stencil operates on a grid size “U” specified as an argument. The grid parameter need not be passed in the DSL. It can be specified later as a command line argument when executing for bench mark harness using auto tuner.
- The “iterate” term specifies the number of time steps to be performed for the stencil computation. If this information is not provided, then by default the stencil computation is run for t = 1 time step.
- The “domainsize” defines the iteration space for the stencil computation. The total grid size is mentioned as “U” which takes on 0 … max-1 values. The stencil computation is applied only to the iteration space as specified in “domainsizes” and not to the whole grid size “U”.

- The “initial” specifies the initialization of the grids
- “operation” defines the actual computations for the “example”.

Chapter 4 Brief overview of Pluto compiler

4.1 Pluto

Pluto is an automatic parallelization tool which uses polyhedral model to transform the code for better data locality and parallelism. A polyhedral dependency matrix is a matrix which gives complete information regarding the kernel. Pluto makes a geometric representation by using this matrix and finds affine transformations for effective tiling of the loops. Pluto is not limited to only tiling transformation; it can find affine transformations for effectively fusing the loops and many more. Thus, pluto using the polyhedral model transforms the C code from source to source for a coarse-grained parallelism and data locality.

Tiling is a key transformation technique used by pluto to optimize the code for better data locality and parallelism. To transform the code for better locality, pluto divides the iteration space of the loops in such a way that the data access needed in the innermost tiled loop fits appropriately in cache. By doing so, a better reuse of the data placed in the cache is enabled and this improves locality of data. During this process pluto generates many tiles which can be concurrently executed on different processors with reduced frequency and communication between the tiles. This way pluto achieves parallelism among the tiles for faster execution. Therefore, finding an appropriate tile is an important and integral part of pluto.

The task of program optimization in polyhedral model may be viewed in phases as (Bondhugula, Ramanujam, & Sadayappan, 2008):

- Static dependency analysis of input program
- Transformation in polyhedral abstraction
- Generation of code for the transformed program

Pluto uses the mathematical models in an algorithm to find appropriate tiles. The mathematical models are briefly shown:

In order to find legal tiles with affine dependencies on multiple domains, the following must be satisfied (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008):

$\emptyset sj(t) - \emptyset si(s) \geq 0,\ where\ (s,t) \in Pe$ and $\emptyset sk$ and Pe are legal tiling hyerplanes and dependency polyhedron respectively.

In order to find minimum latency schedules (Bondhugula, Hartono, Ramanujam, & Sadayappan, 2008):

$v(p) - \left(\emptyset(t) - \emptyset\left(he(t)\right)\right) \geq 0$ under dependency polyhedron.

An example shown below illustrates the original code and the transformed code with dependency polyhedron from accesses to array A (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008):

// original code:

```
for ( i=0; i< N; i++ )
        for ( j=0; j<N; j++ )
                S1: A[i][j] = A[i][j] + u[i] * v[j];

for ( i=0; i<N; i++ )
        for ( j=0; j<N; j++ )
                S2: x[i] = x[i] + A[j][i] * y[j];
```

The dependency polyhedron (Pe) for inter statement dependence on A is (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ i' \\ j' \\ N \\ 1 \end{pmatrix} \geq 0$$

The transformation is as shown (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008):

For statement S1:

$$\begin{array}{cccc} & i & j & \text{const} \\ c1: & 0 & 1 & 0 \\ c2: & 1 & 0 & 0 \\ c3: & 0 & 0 & 0 \end{array}$$

For statement S2:

$$\begin{array}{cccc} & i & j & \text{const} \\ c1: & 1 & 0 & 0 \\ c2: & 0 & 1 & 0 \\ c3: & 0 & 0 & 1 \end{array}$$

The transformed code is as shown (Bondhugula, Baskaran, Krishnamoorthy, Ramanujam, Rountev & Sadayappan, 2008):

```
for ( c1=0; c1<N; c1++) {
        for ( c2=0; c2<N; c2++) {
                A[c2][c1] = A[c2][c1] + u[c2] * v[c1];
                x[c1] = x[c1] + A[c2][c1] * y[c1];
        }
}
```

4.2 Diamond tiling

Stencil computations are computed by updating gird points using neighboring grid point's values. Hence stencils exhibit properties for data locality and parallelism optimizations. With pluto tiling framework, the tile hyper planes are chosen in such a way that the tiles would be enabled for pipelined execution. But at the start of the computation, not all the processors are busy. Therefore this leads to a load imbalance start up (Bondhugula, Pananilath, & Bandishti, 2012). This issue can be overcome by using diamond tiling. There always exists a face of the iteration space and a set of hyper planes to select, such that the combination could lead to a tile wise concurrent start up. This provides a good load balance by eliminating pipeline fill ups and drain-delays, and maximizes parallelism (Bondhugula, Pananilath, & Bandishti, 2012). For getting a concurrent start up at least one level of outer loop parallelism is expected. This nature is observed in Jacobi style, which is explained in results section. We can observe from the graphs that diamond tiling gives significant results. If the dependencies are present all over the iteration space then getting a concurrent startup is not possible. This nature is observed in seidel style, also explained in results section, the dependencies are spread throughout the iterations and hence we observe no concurrent start up.

Brief mathematical notations and constraints for concurrent startup are discussed below:

In order to find valid hyper planes from source iteration to sink iteration, the following should be valid (Bondhugula, Pananilath, & Bandishti, 2012):

$\emptyset sj(t) - \emptyset si(s) \geq 0$, where (s, t) ∈ Pe.

Pe – Polyhedron dependency matrix

$\emptyset sj(t)$ *and* $\emptyset si(s)$ are hyper planes for statements

Theorem 1 (Bondhugula, Pananilath, & Bandishti, 2012):

For a statement, a transformation enables tile-wise concurrent start along a face f iff the tile schedule is in the same direction as the face and carries all inter-tile dependences.

This is denoted as: f . C ≥ 1, where f is the face allowing concurrent startup and C is a matrix containing inter-tile dependencies of original iteration space.

Theorem 2 (Bondhugula, Pananilath, & Bandishti, 2012):

Concurrent start along a face f can be exposed by a set of hyper-planes iff f lies strictly inside the cone formed by the hyper-planes, i.e., iff f is a strict conic combination of all the hyperplanes

This is denoted as: $kf =$ lambda1 . h1 + lambda2 . h2 + … + lambdaN . hN

Theorem 3 (Bondhugula, Pananilath, & Bandishti, 2012):

A transformation T allows concurrent startup along f iff f . inverse(Tr) ≥ 1.

An algorithm is proposed by Uday to find the appropriate hyper-planes which enable concurrent startup.

4.3 Sica tiling

Tile size selection plays an important role in tiling optimization technique. Pluto uses a default tile selection (32) when compiling the C code for optimization using tiling technique. But this default tile selection does not necessarily yield an optimized code that always performs well on every application. There always exists a tile size for which the optimized code performs best.

Sica tiling presents a hardware aware and adaptive loop tiling approach that is based on pluto's polyhedral transformations and dedicated to improve auto-vectorization (Dustin, Thomas, Michael, & Sven, 2013). They use an adaptive strategy to find near optimal tile sizes for vectorizable loops through dynamic tile size calculations. The tile size for first level of tiling should fit to the ratio of the amount of data read in one iteration of the loop which is vectorizable and the size of the L1 cache. In the same way, the tile size for the second level of tiling should fit to the ratio of L2 cache size and L1 cache size (Dustin, Thomas, Michael, & Sven, 2013). The calculation of different levels of tiling is as shown below (Dustin, Thomas, Michael, & Sven, 2013):

First level tile size = q (L1) = floor ($\rho * CL1 * 8192 / R * \varepsilon$) $* R / (8 * D)$

Where:

ρ = ratio of cache to use

CL1 and CL2 = the size of L1 and L2 cache in Kbytes

R = SIMD register width in bits

ε = Elements per iteration

D = size of data type

Second level tile size = q (L2) = CL2 / CL1

Chapter 5 Results

5.1 Results

In this section, the report provides the performance results obtained by executing stencil benchmarks. Two experimental styles were chosen to compare the performances of the compilers on stencils:

1) Jacobi style
2) Seidel style

Five out of six stencil benchmarks are taken from patus compiler. These stencil benchmarks include blur, edge, game of life, laplacian and wave. Seidel stencil is a benchmark that is written in seidel style and hence it is not considered in Jacobi style of evaluation. In seidel style, all the six stencils are written in seidel style of coding to compare the performances.

The following gives information regarding the use of compiler specific command line options and commands to compile the C code for optimizations:

- **Pluto:**
  - polycc -- tile -- parallel filename.c –o filename.par.c

    “parallel“ command line option uses skewing transformation technique. This will either skew to make outer loop carry all dependency and inner loops vectorizable or makes outer loop parallel for tiling purpose. The “tile” option tiles the loops using tiling technique. This option gives a pipelined execution of tiles and not concurrent tile execution.
  - polycc -- partlbtile -- parallel filename.c –o filename.diamond.c

    The “parallel” flag will perform as explained above. The “partlbtile” flag tiles the loops in such a way that it would enable concurrent execution of tiles.

- Polycc –tile –sica –parallel filename.c –o filename.sica.o

  The “tile” and “parallel” flags perform as explained above. The “sica” flag tiles the code using sica model of tiling.

The table 5.1gives information of the compiler and the compiler flags used to compile the C and optimized codes generated by pluto.

TABLE 5.1. Compiler and compiler flags for pluto

Compiler	icc 14.0.1
Compiler flags	-O3 –fp-model precise –mavx -openmp

- **Patus:**
  - patus filename.stc

    This command uses “patus” as a wrapper on the DSL and compiles the DSL to produce an optimized C code with other related files for benchmark harness.
  - Make tune [options]

    This command will start the auto tuner feature of patus for benchmark harness and provides the configuration information that ran the best. These configuration are used as command line options in the below shown command.
  - ./bench [options]

    This command uses the configuration provided by the auto tuner as command line option to execute the optimized C code.

The table 5.2 provides the information on the compiler and compiler flags used to compile the optimized C code generated by patus.

TABLE 5.2. Compiler and compiler flags for patus

Compiler	gcc 4.4.7 (Jacobi style) / icc 14.0.1 (Seidel style)
Compiler flags	-O3 –mavx –openmp

The hardware configuration used is:

TABLE 5.3. Hardware configuration

Microarchitecture	Details
Model	Intel ® Xeon ® CPU E5 – 2624
Clock	2.00 GHz
Cores	6
Processors	12
L1 cache / core	32KB
L2 cache / core	256KB
L3 cache / socket	15KB

5.2 Jacobi style

In Jacobi style the stencil computations are written to a temporary array and values from temporary array are copied back to the actual array in the later computation. An example is shown below to illustrate Jacobi style of coding. Due to the use of such a coding style, dependencies do not exists inside loop nest but dependencies exist over time steps and between loop nests of a particular time. In the example below, there are no dependencies in loop nest 1 (or 2) but dependencies exist between loop nest 1 and loop nest 2 and also over time steps.

```
// Time step
For ( t=0; t<N; t++ )
        // Loop nest 1
        For ( i …)
                For ( j …)
                        temp_array[i][j] = array[i][j] + …..
        // Loop nest 2
        For ( i …)
                For ( j…)
                        array[i][j] = temp_array[i][j]
```

Shown below are the performance of five stencils when compiled with pluto and patus. Each stencil is written in Jacobi style and the compiler specific commands are also shown:

- **Blur stencil:**

  Pluto commands:

  - polycc –tile –parallel blur.c –o blur.par.c
  - polycc –partlbtile –parallel blur.c –o blur.diamond.c
  - polycc –tile –sica –parallel blur.c –o blur.sica.c
  - icc –O3 –fp-model precise –mavx blur.c –o orig (for the C code)
  - icc –O3 –fp-model precise –mavx –openmp blur.par.c –o par (for pipelined parallel code)
  - icc –O3 –fp-model precise –mavx –openmp blur.diamond.c –o diamond (for concurrent parallel code)
  - icc –O3 –fp-model precise –mavx –openmp blur.sica.c –o sica (for sica tiled parallel code)

  Patus Commands:

  - patus blur.stc
  - make tune height=6000 width=6000 (for grid size 6000*6000)
  - ./bench 6000 6000 2998 200 1 4 1
  - make tune height=8000 width=8000 (for gris size 8000*8000)
  - ./bench 8000 8000 3998 148 1 4 1
  - make tune height=10000 width=10000 (for grid size 10000*10000)
  - ./bench 10000 10000 4998 128 1 4 1

The table 5.4 gives the information of performance of blur stencil when compiled via pluto and patus. The numbers mentioned in brackets gives information of tile sizes that worked best.

TABLE 5.4. Performace table for pluto and patus on blur stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	12.24	7.55	5.13(24)	16.39	65.35	1.28	1.52(128)	55.32	6.69	15.67
8000*8000	80	26.07	7.46	10.72(16)	18.60	177.25	0.96	3.64(96)	54.78	12.13	16.03
10000*10000	100	51.36	7.78	14.76(24)	26.39	264.12	1.10	7.34(96)	53.08	17.66	24.34

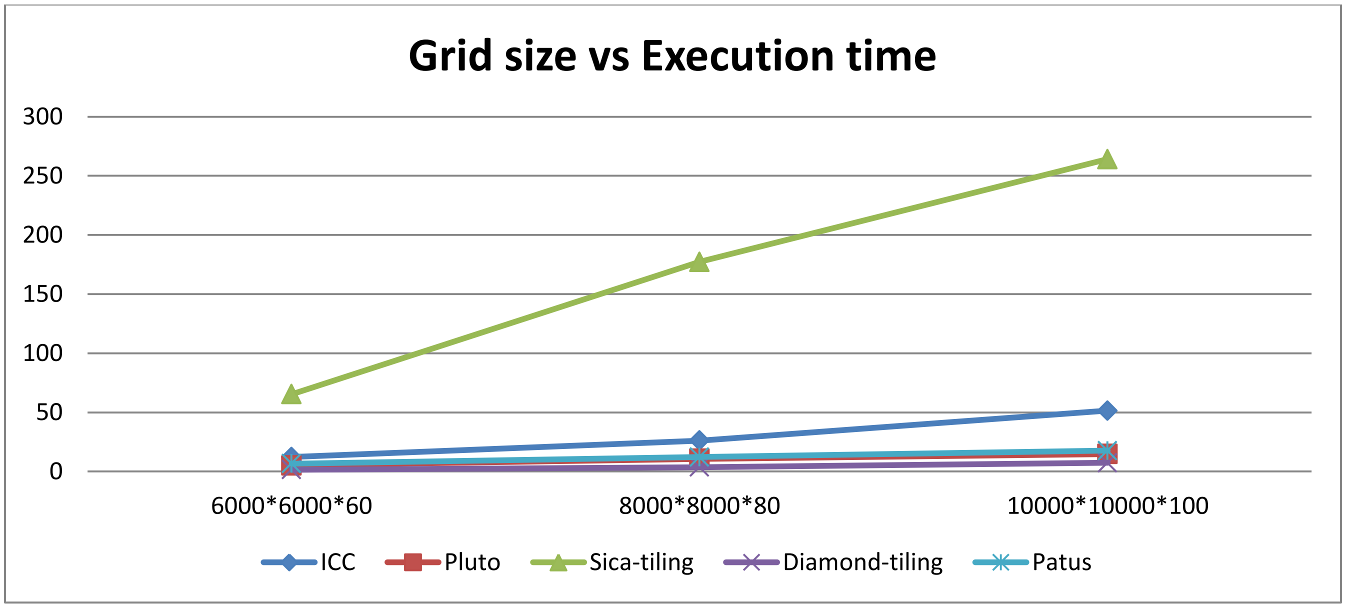


FIGURE 5.1. Grid size versus Execution time

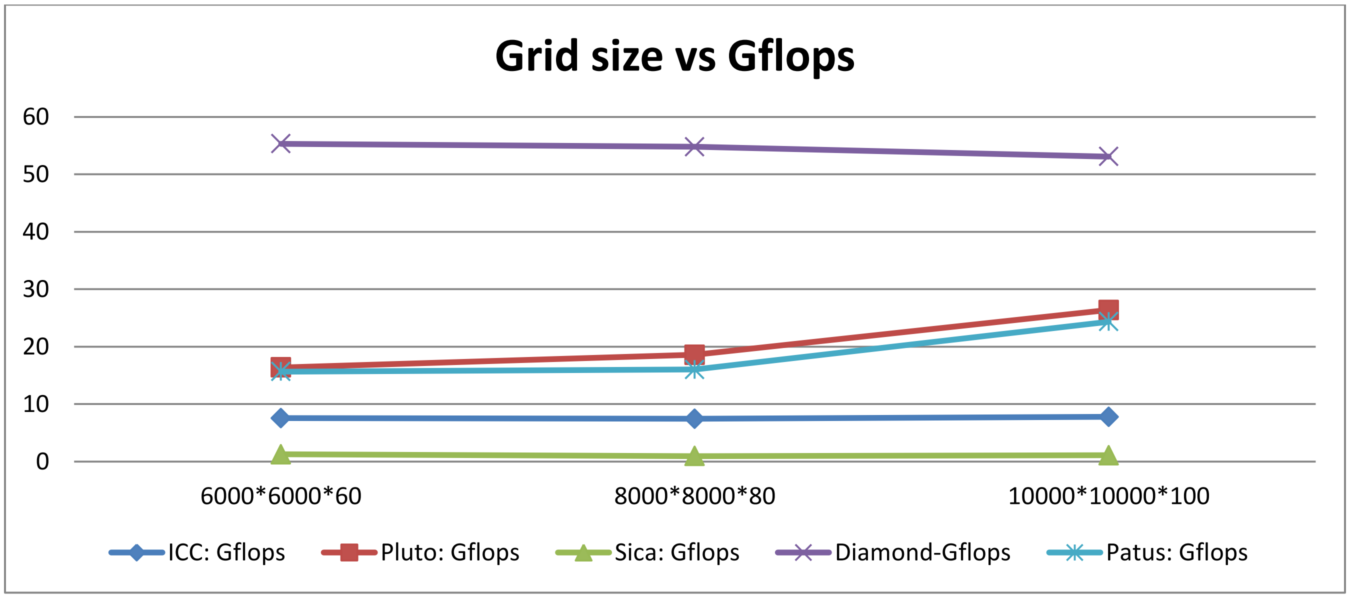


FIGURE 5.2. Grid size versus GFOPLS

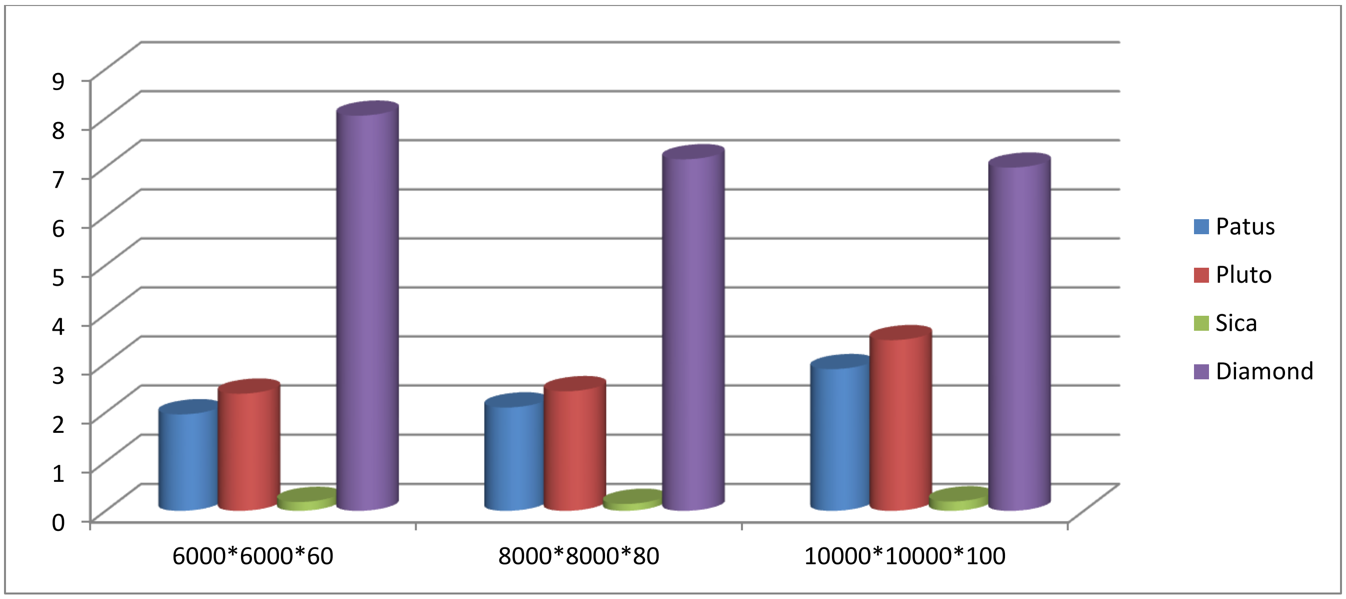


FIGURE 5.3. Speedup performance

- **Edge stencil:**

Pluto commands:

- polycc –tile –parallel edge.c –o edge.par.c
- polycc –partlbtile –parallel edge.c –o egde.diamond.c
- polycc –tile –sica –parallel edge.c –o edge.sica.c
- icc –O3 –fp-model precise –mavx egde.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp edge.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp edge.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp edge.sica.c –o sica (for sica tiled parallel code)

Patus commands:

- patus edge.stc
- make tune height=6000 width=6000 (for grid size 6000*6000)

- ./bench 6000 6000 5998 140 2 4 0
- make tune height=8000 width=8000 (for gris size 8000*8000)
- ./bench 8000 8000 7998 24 4 1 0
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 4999 224 1 2 1

The table 5.5 gives the information of performance of edge stencil when compiled via pluto and patus.

TABLE 5.5. Performance table for patus and pluto on edge stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	8.48	2.54	2.64(16)	8.15	36.55	0.58	1.01(104)	21.33	3.30	6.53
8000*8000	80	18.69	2.73	5.01(16)	10.19	65.79	0.64	2.51(88)	20.30	8.34	6.13
10000*10000	100	41.95	2.38	10.84(16)	9.34	111.83	0.72	4.93(72)	20.23	17.14	5.82

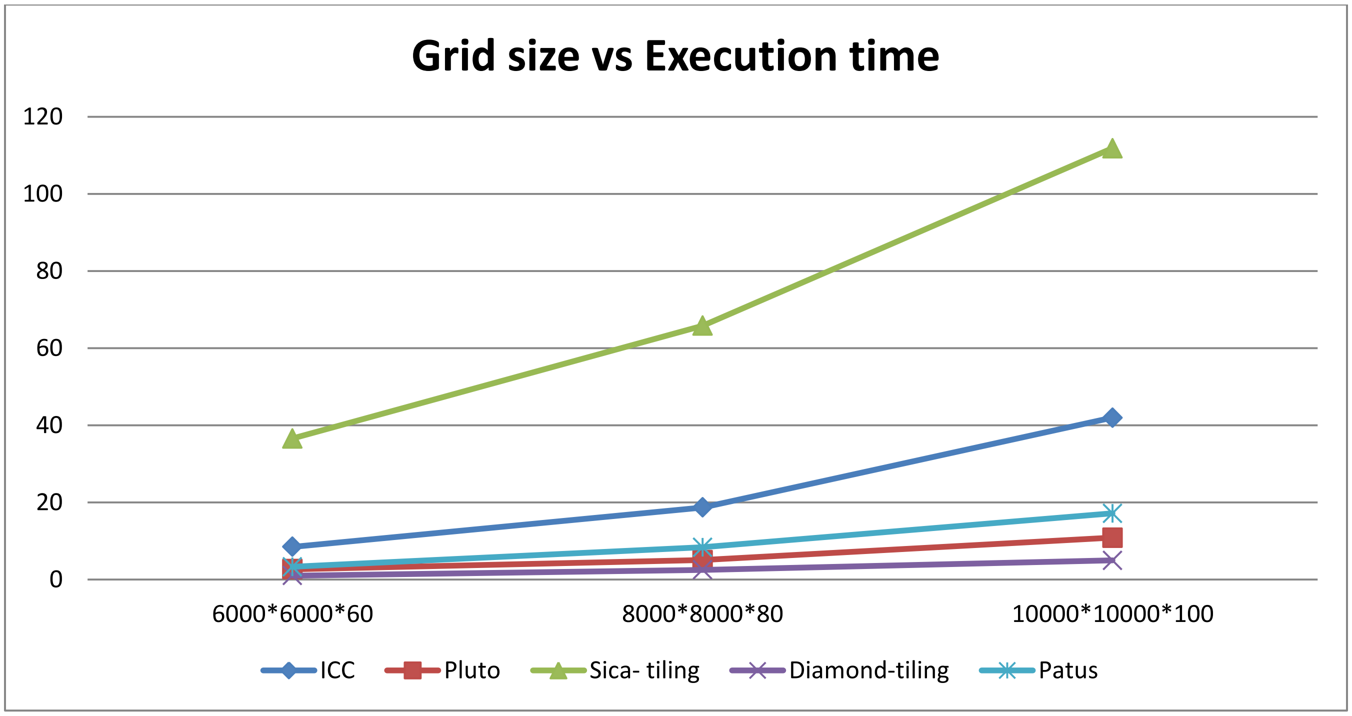


FIGURE 5.4. Grid size versus Execution time

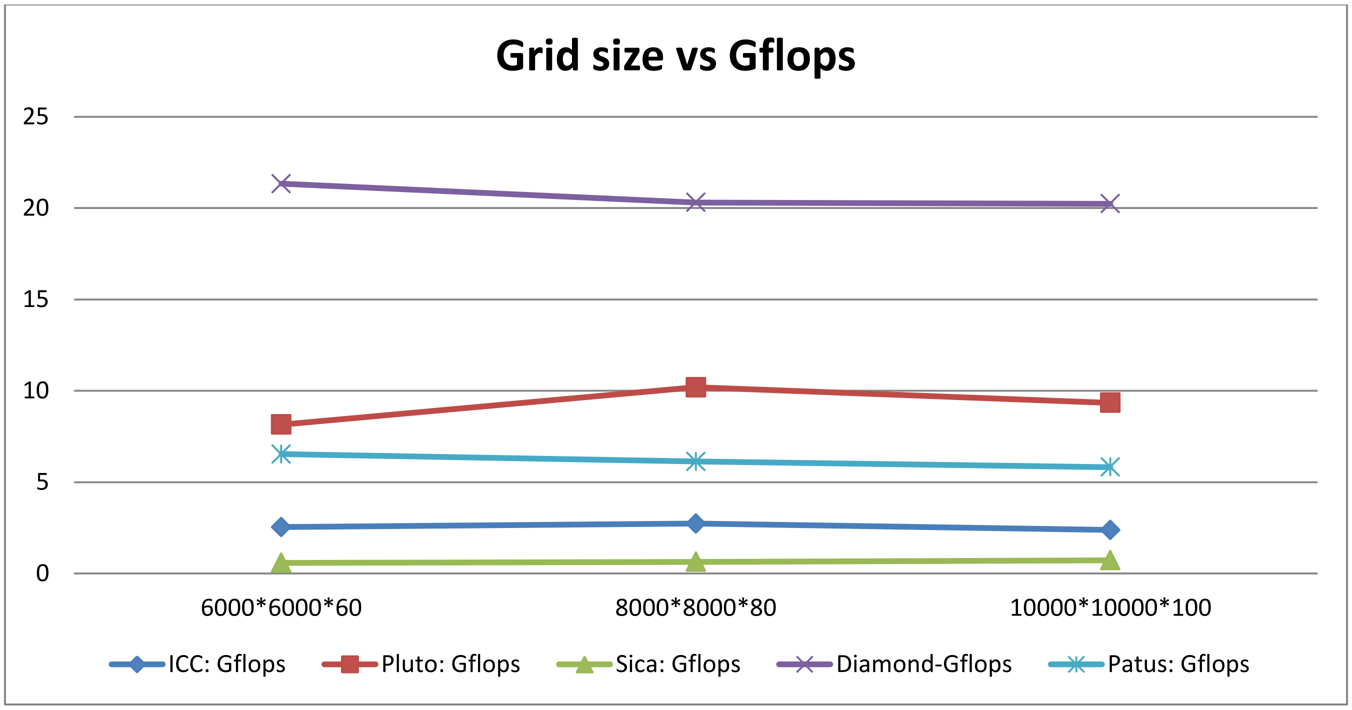


FIGURE 5.5. Grid size versus GFLOPS

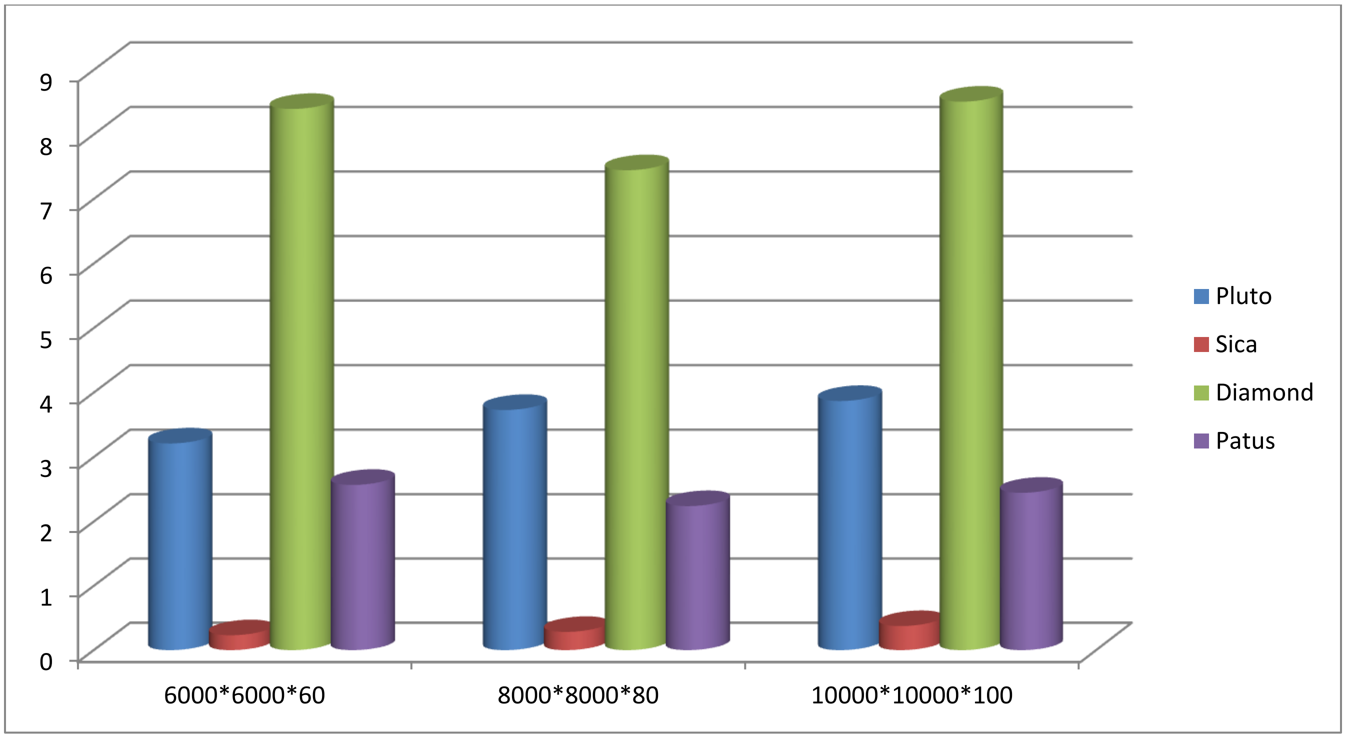


FIGURE 5.6. Speedup performance

- **Game of life stencil:**

Pluto commands:

- polycc –tile –parallel gol.c –o gol.par.c
- polycc –partlbtile –parallel gol.c –o gol.diamond.c
- polycc –tile –sica –parallel gol.c –o gol.sica.c
- icc –O3 –fp-model precise –mavx gol.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp gol.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp gol.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp gol.sica.c –o sica (for sica tiled parallel code)

Patus commands:

- patus game-of-life.stc
- make tune height=6000 width=6000 (for grid size 6000*6000)
- ./bench 6000 6000 2998 144 1 4 0
- make tune height=8000 width=8000 (for gris size 8000*8000)
- ./bench 8000 8000 7998 172 4 4 0
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 4999 280 1 4 1

The table 5.6 gives the information of performance of game of life stencil when compiled via pluto and patus.

TABLE 5.6. Performance table for patus and pluto on game of life stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	11.65	2.58	5.42(16)	5.56	54.56	0.55	2.09(120)	14.42	4.10	7.36
8000*8000	80	27.45	2.60	11.17(16)	6.40	96.17	0.60	5.16(96)	13.85	9.57	7.48
10000*10000	100	54.04	2.58	18.25(16)	7.66	174.01	0.62	10.02(102)	13.95	15.11	9.25

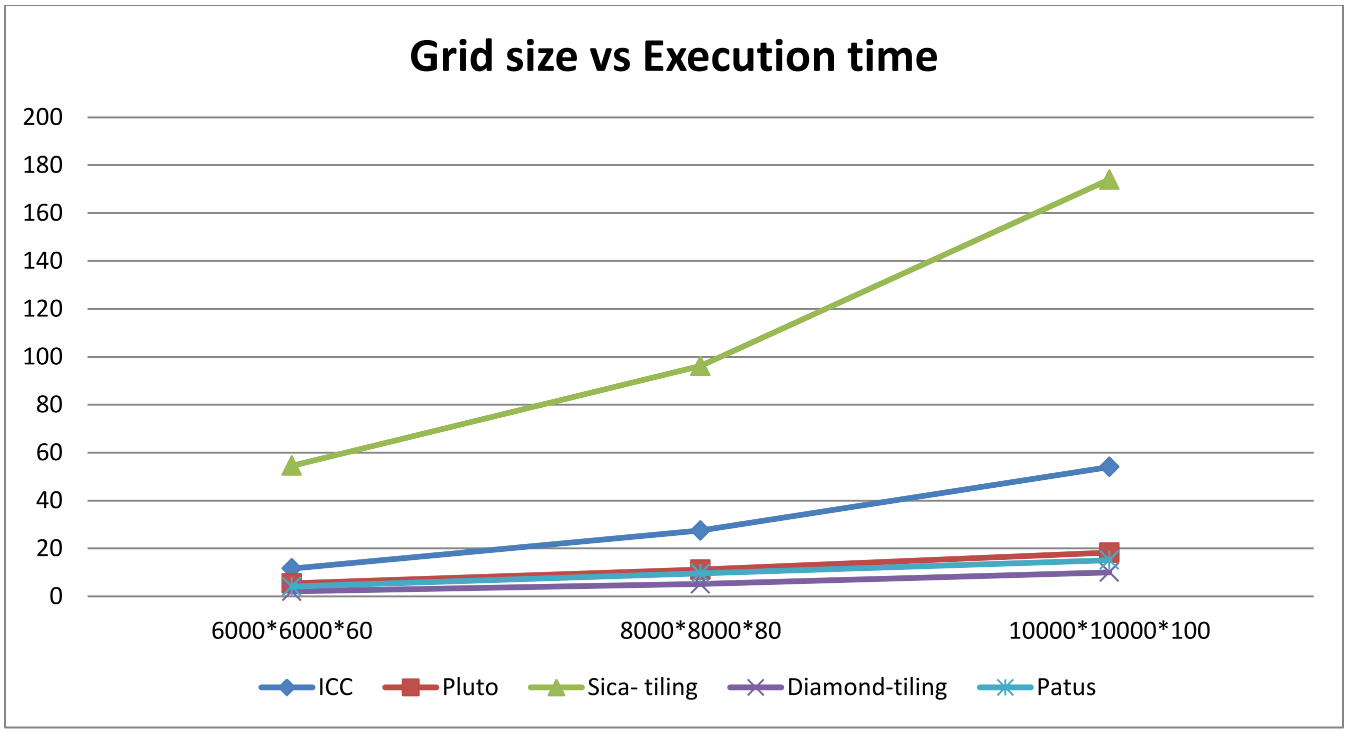


FIGURE 5.7. Grid size versus Execution time

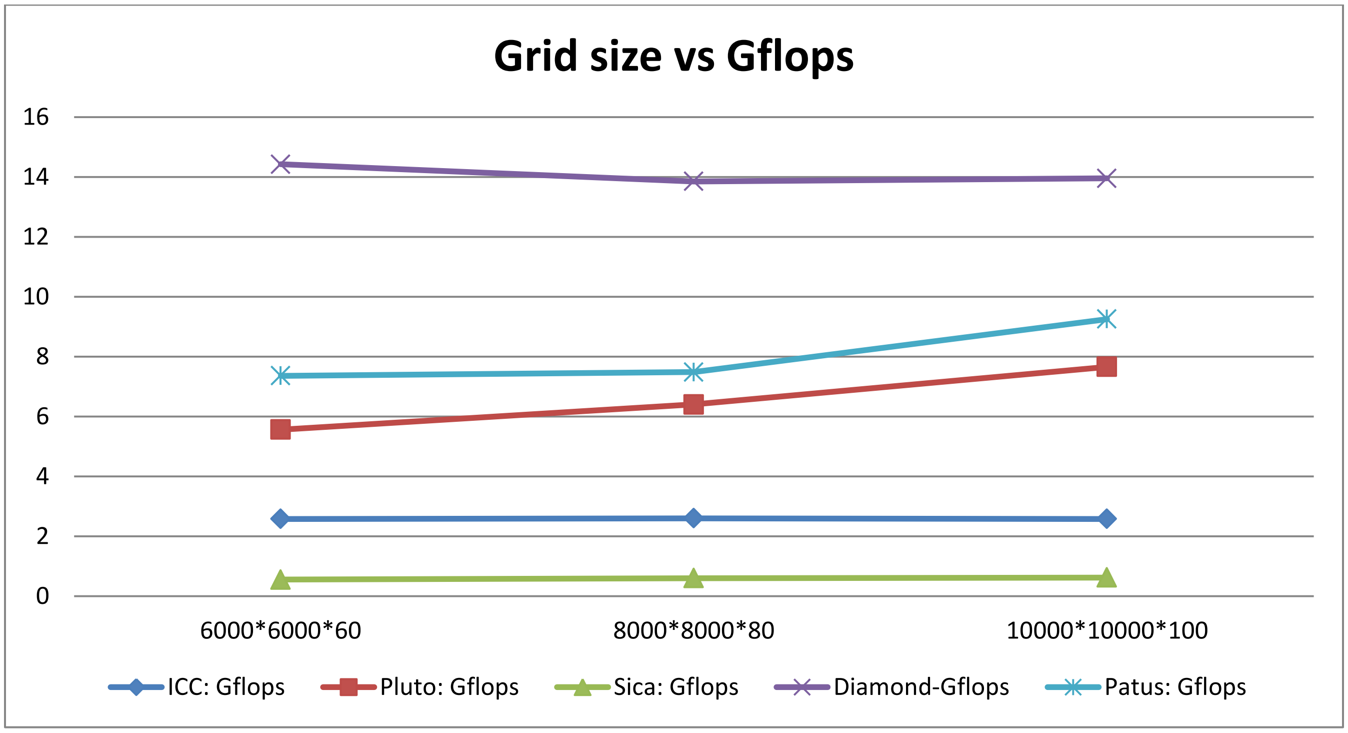


FIGURE 5.8. Grid size versus GFLOPS

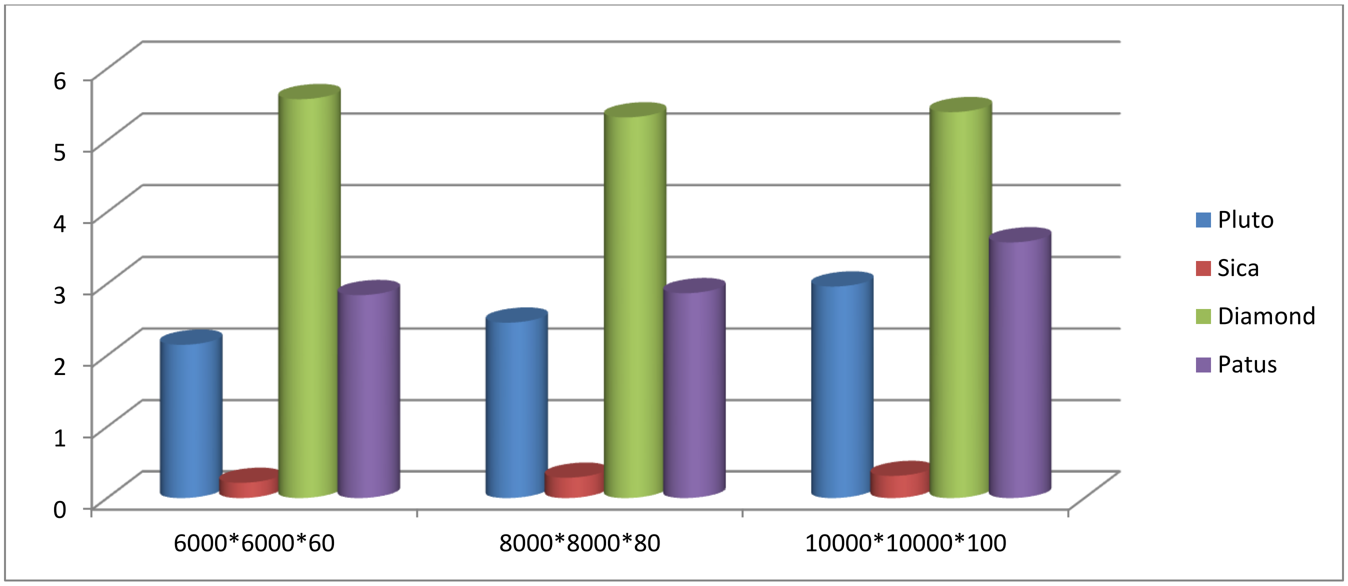


FIGURE 5.9. Speedup performance

- **Laplacian stencil:**

Pluto command:

- polycc –tile –parallel lap.c –o lap.par.c
- polycc –partlbtile –parallel lap.c –o lap.diamond.c
- polycc –tile –sica –parallel lap.c –o lap.sica.c
- icc –O3 –fp-model precise –mavx lap.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp lap.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp lap.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp lap.sica.c –o sica (for sica tiled parallel code)

Patus command:

- patus laplacian.stc
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 149 20 8 1 8 0

- make tune x_max=400 y_max=400 z_max=400 (for grid size 400*400*400)
- ./bench 400 400 400 199 148 148 1 4 1
- make tune x_max=500 y_max=500 z_max=500 (for grid size 500*500*500)
- ./bench 500 500 500 249 4 64 1 4 0

The table 5.7 gives the information of performance of laplacian stencil when compiled via pluto and patus.

TABLE 5.7. Performance chart for patus and pluto on laplacian stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
300*300*300	300	31.15	1.27	10.95(8)	5.62	5.52	11.14	9.40(32)	6.55	21.40	2.87
400*400*400	400	108.70	1.44	27.46(8)	7.18	15.87	12.42	25.94(40)	7.58	72.70	2.71
500*500*500	500	198.21	1.76	61.69(8)	7.86	39.86	12.16	56.01(24)	8.66	144.68	3.35

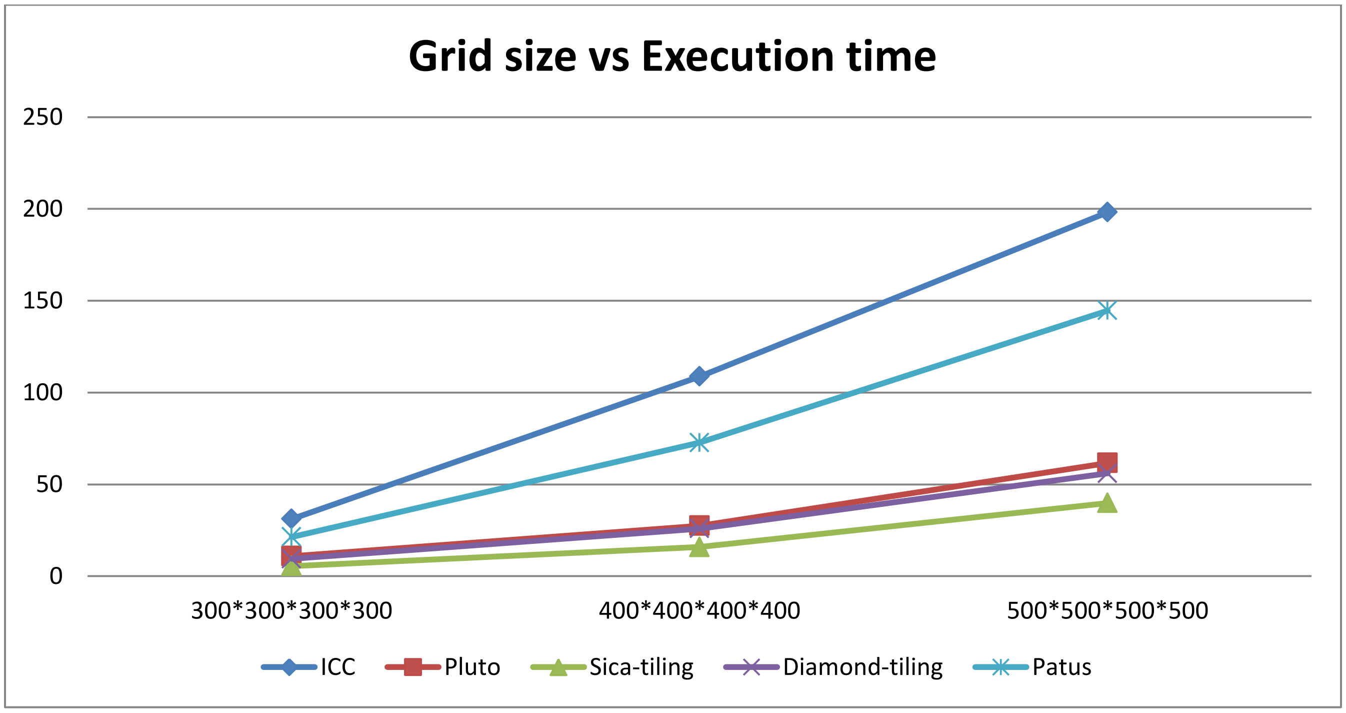


FIGURE 5.10. Grid size versus Execution time

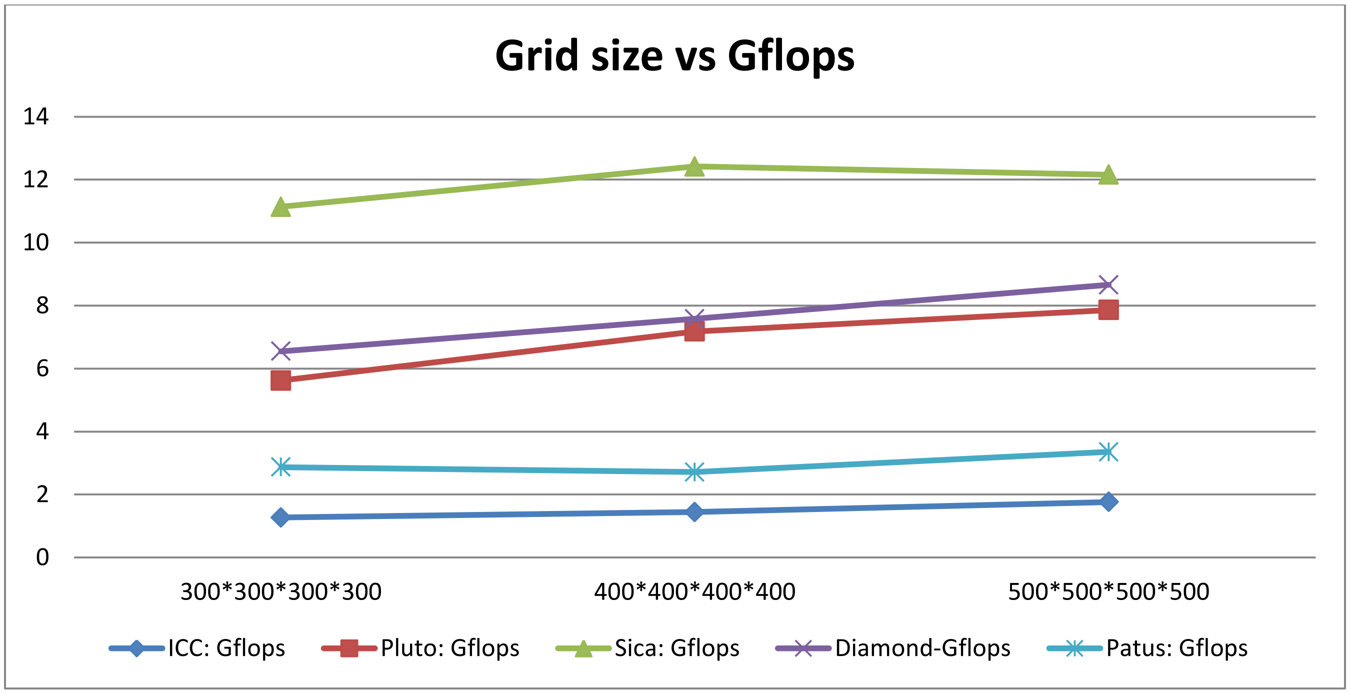


FIGURE 5.11. Grid size versus GFLOPS

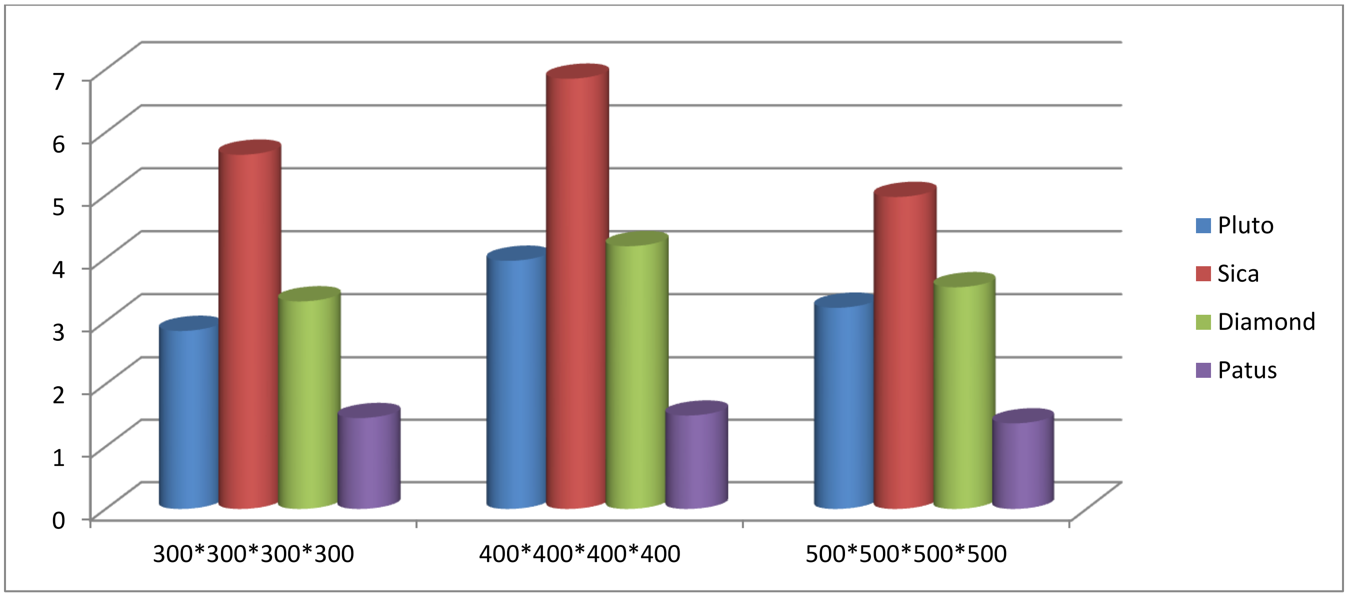


FIGURE 5.12. Speedup performance

- **Wave stencil:**

Pluto command:

```
- polycc –tile –parallel wave.c –o wave.par.c

- polycc –partlbtile –parallel wave.c –o wave.diamond.c
```

- polycc –tile –sica –parallel wave.c –o wave.sica.c
- icc –O3 –fp-model precise –mavx wave.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp wave.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp wave.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp wave.sica.c –o sica (for sica tiled parallel code)

Patus Command:

- patus wave.stc
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 299 8 8 1 4 3
- make tune x_max=400 y_max=400 z_max=400 (for grid size 400*400*400)
- ./bench 400 400 400 395 32 32 1 7 3
- make tune x_max=500 y_max=500 z_max=500 (for grid size 500*500*500)
- ./bench 500 500 500 499 4 44 1 4 8

The table 5.8 gives the information of performance of wave stencil when compiled via pluto and patus.

TABLE 5.8. Performance table for patus and pluto on wave stencils

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
300*300*300	300	25.31	3.31	11.55(8)	13.33	7.36	20.91	8.46(32)	18.19	16.04	9.60
400*400*400	400	67.89	3.31	27.33(8)	18.03	23.09	21.34	26.72(40)	18.44	52.14	9.45
500*500*500	500	175.69	3.31	72.73(32)	16.67	54.32	22.32	58.14(48)	20.86	126.89	12.12

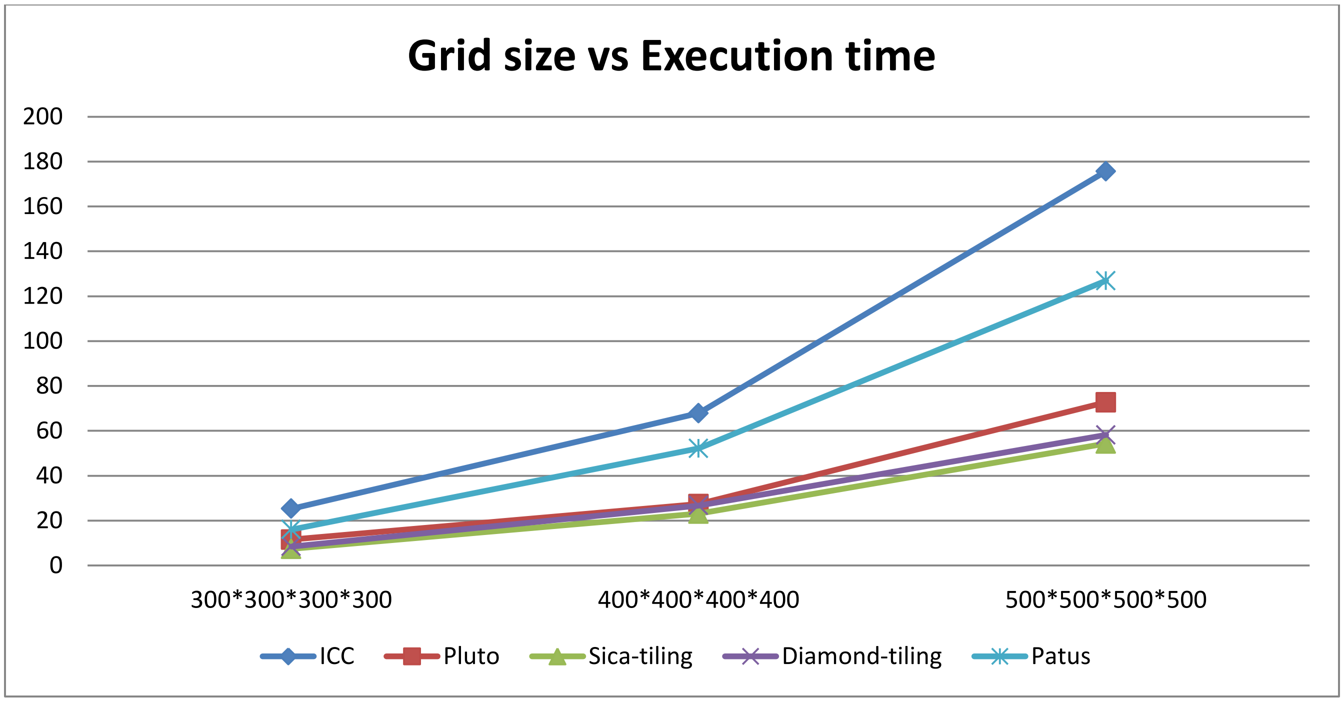


FIGURE 5.13. Grid size versus Execution time

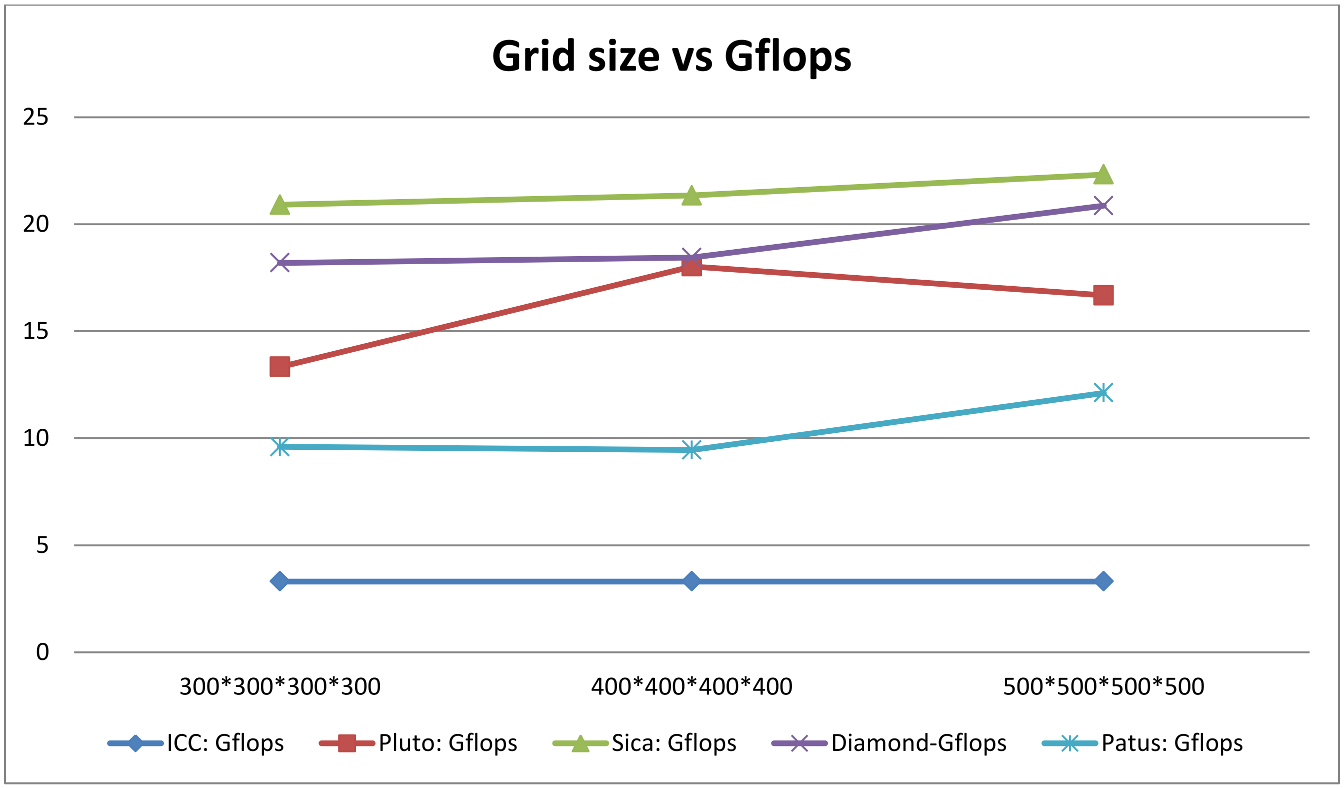


FIGURE 5.14. Grid size versus GFLOPS

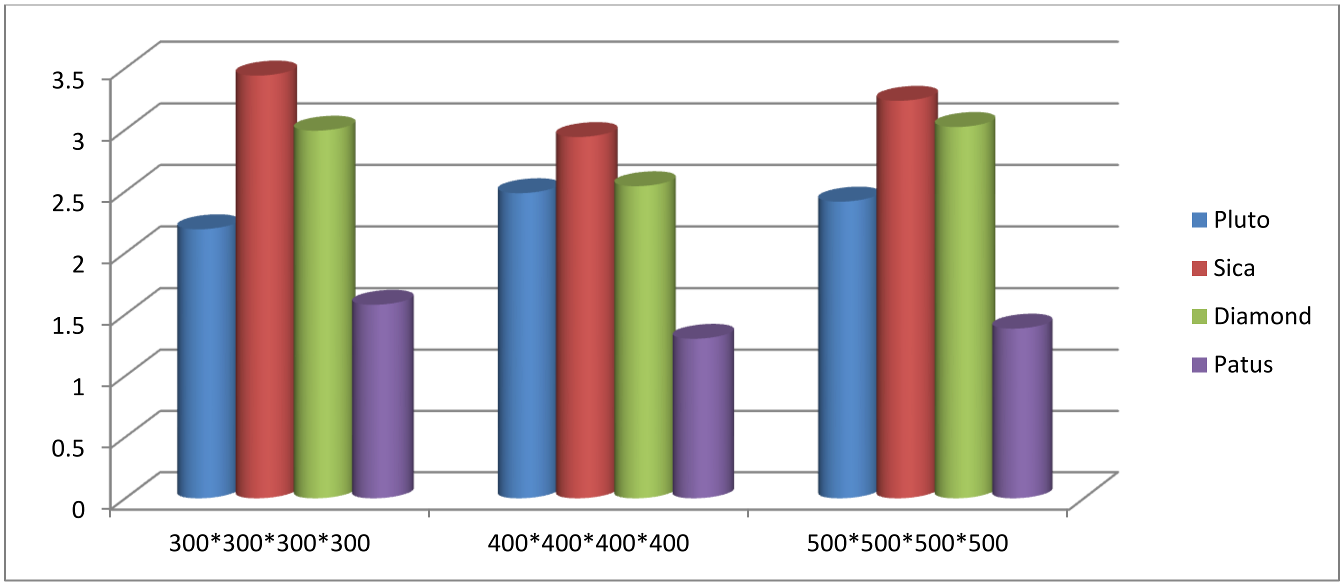


FIGURE 5.15. Performance speedup

Observation and comments:

We can observe that in all the graphs of stencils, pluto has performed better than patus. In terms of optimization approach, both the compilers perform tiling (blocking) technique on the loops and vectorization of inner most loops. But why pluto has performed better than patus?. This is because, patus can only tile (block) the grid sizes and use vectorization on innermost loops, but, pluto on the other hand does tiling (blocking) on both grid sizes and time loop and also use vectorization on inner most loops. To put in simple terms, patus can only perform space tiling, whereas pluto can do both time and space tiling. This feature of pluto gives it an edge over patus in terms of performance. At best, patus, with more tuning to find better tiling (blocking), can reach up to pluto's pipelined tiling performance, this is observed in game of life stencil. However, patus cannot reach to pluto's diamond tiling performance due to the lack of time tiling (blocking).

5.3 Seidel style

Unlike Jacobi style of coding, the stencil computations in seidel style are written to an array and this array might be accessed depending upon the dependencies between them in subsequent iterations. An example is shown below to illustrate seidel style of coding. We can see that the values written to the array are accessed in the subsequent iterations for computation. Hence the dependencies exist within the loop iterations.

```
For (t=0; t<N; t++)
        // Loop nest
        For (i ….)
                For (j …)
                        a[i][j] = a[i-1][j-1] + a[i][j-1] + …
```

Shown below are the performances of six stencils when compiled with pluto and patus. Each stencil is written in seidel style and compiler specific commands are also shown.

- **Blur stencil:**

  Pluto commands:

  - polycc –tile –parallel blur.c –o blur.par.c
  - polycc –partlbtile –parallel blur.c –o blur.diamond.c
  - polycc –tile –sica –parallel blur.c –o blur.sica.c
  - icc –O3 –fp-model precise –mavx blur.c –o orig (for the C code)
  - icc –O3 –fp-model precise –mavx –openmp blur.par.c –o par (for pipelined parallel code)
  - icc –O3 –fp-model precise –mavx –openmp blur.diamond.c –o diamond (for concurrent parallel code)
  - icc –O3 –fp-model precise –mavx –openmp blur.sica.c –o sica (for sica tiled parallel code)

  Patus Commands:

  - patus -- architecture = x86_64 AVX blur.stc
  - make tune height=6000 width=6000 (for grid size 6000*6000)
  - ./bench 6000 6000 1498 224 2 0
  - make tune height=8000 width=8000 (for gris size 8000*8000)

- ./bench 8000 8000 3997 308 2 4
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 4997 152 4 3

The table 5.9 gives the information of performance of blur stencil when compiled via pluto and patus. The numbers mentioned in brackets gives information of tile sizes that has worked best.

TABLE 5.9. Performance table for patus and pluto on Blur stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	28.09	2.99	8.70(8)	9.65	66.44	1.26	7.45(8)	11.27	22.87	3.67
8000*8000	80	66.62	2.99	13.39(8)	14.88	206.26	0.96	13.92(8)	14.32	58.35	3.41
10000*10000	100	130.19	2.99	29.03(8)	13.41	280.23	1.39	29.01(8)	13.42	107.9	3.61

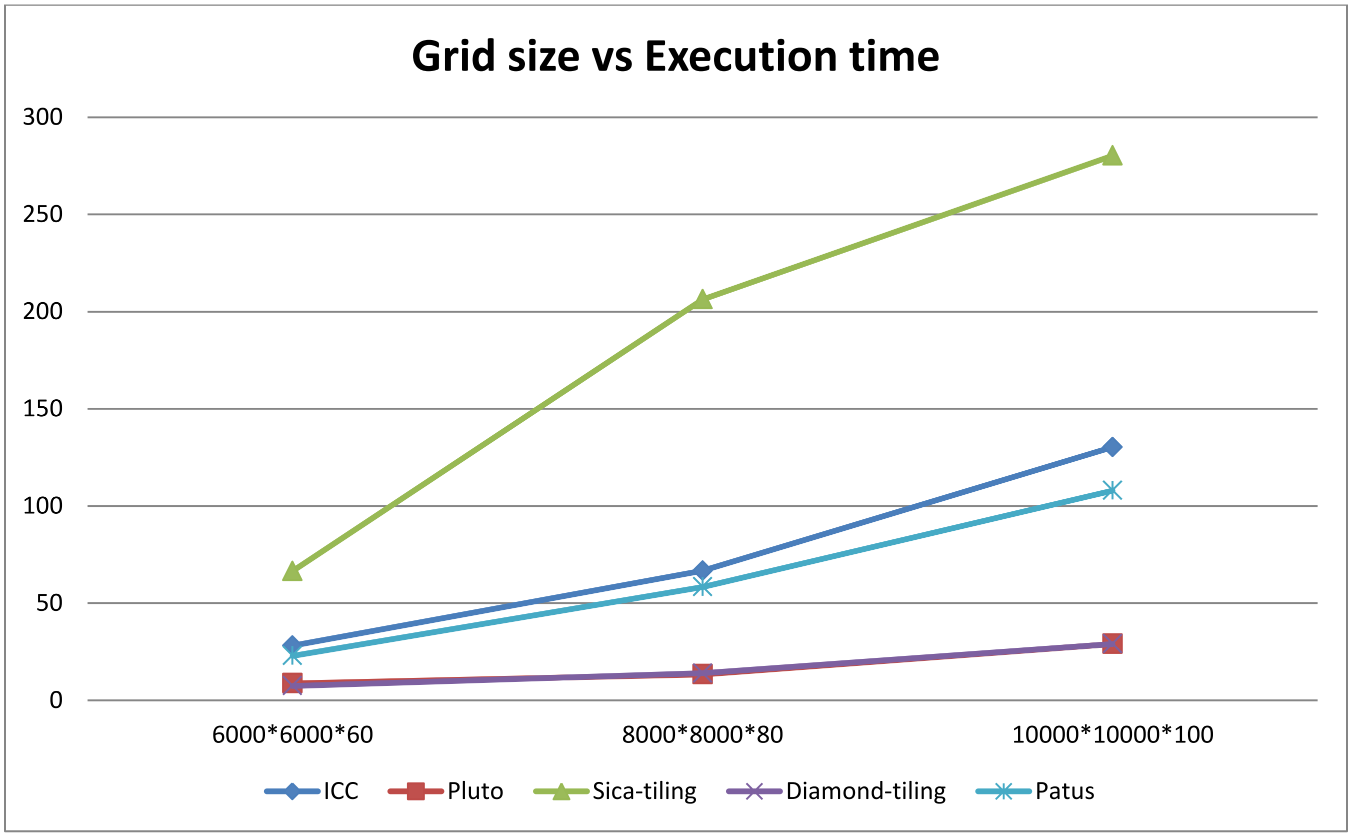


FIGURE 5.16. Grid size versus Execution time

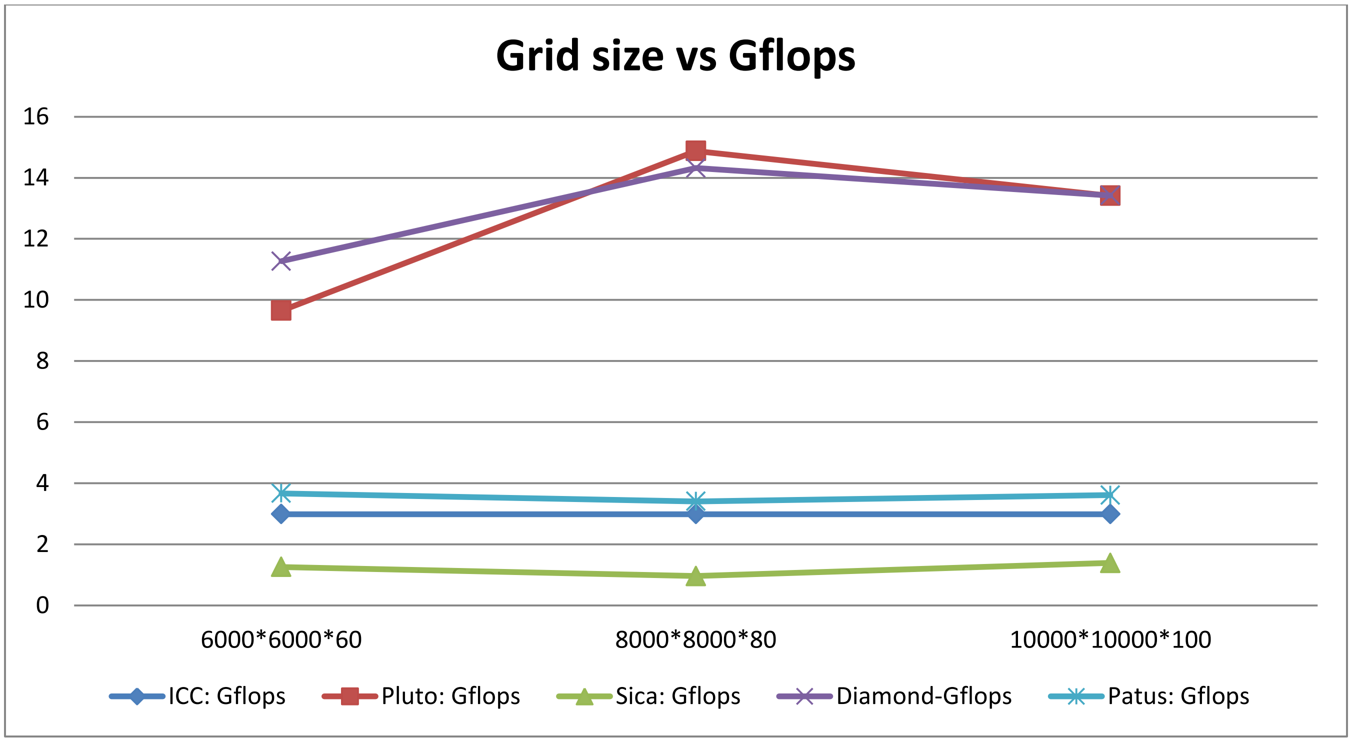


FIGURE 5.17. Grid size versus GFLOPS

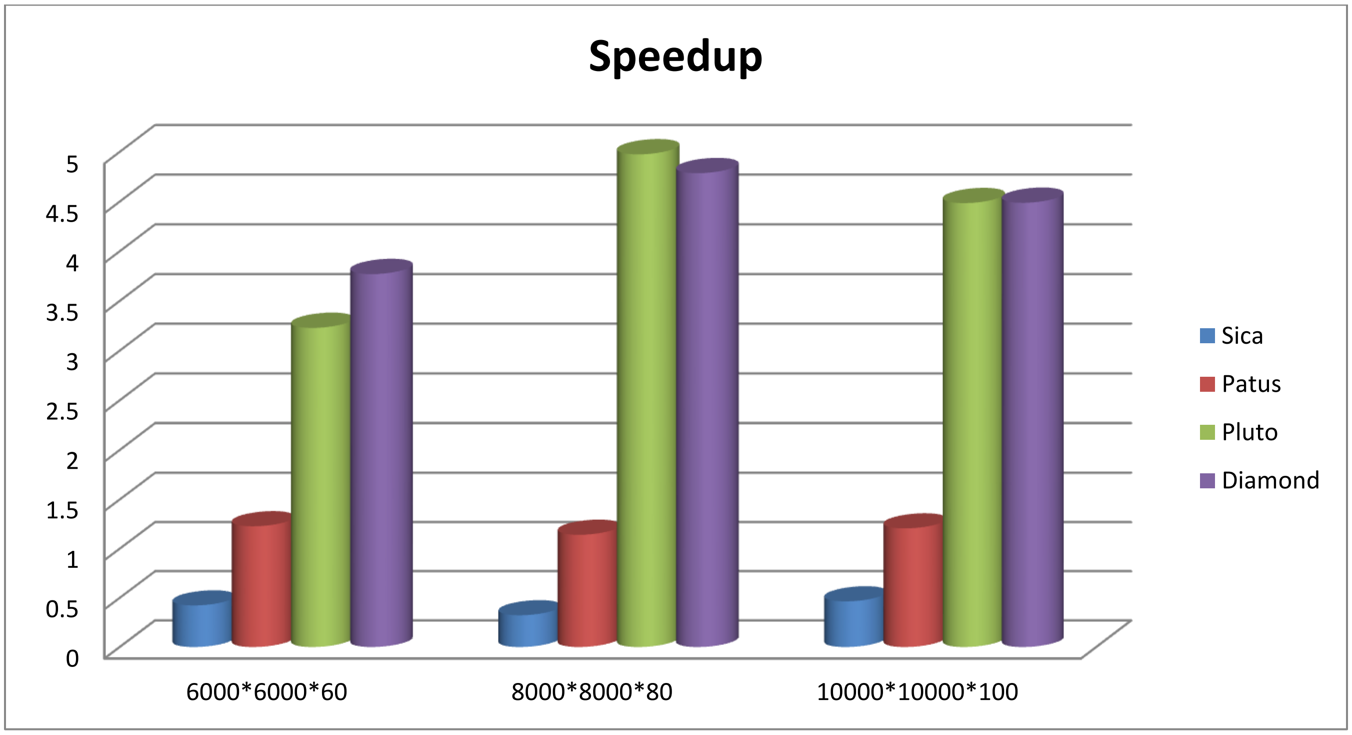


FIGURE 5.18. Performance speedup

- **Edge stencil:**

Pluto commands:

- polycc –tile –parallel edge.c –o edge.par.c
- polycc –partlbtile –parallel edge.c –o egde.diamond.c
- polycc –tile –sica –parallel edge.c –o edge.sica.c
- icc –O3 –fp-model precise –mavx egde.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp edge.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp edge.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp edge.sica.c –o sica (for sica tiled parallel code)

Patus commands:

- patus – architecture = x86_64 AVX edge.stc
- make tune height=6000 width=6000 (for grid size 6000*6000)
- ./bench 6000 6000 2997 332 2 3
- make tune height=8000 width=8000 (for gris size 8000*8000)
- ./bench 8000 8000 3997 332 4 0
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 4997 260 2 3

The table 5.10 gives the information of performance of edge stencil when compiled via pluto and patus.

TABLE 5.10. Performance table for patus and pluto on edge stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	23.58	0.91	4.04(8)	5.32	18.12	1.19	4.25	5.06	13.58	1.58
8000*8000	80	55.92	0.91	6.99(8)	7.30	43.76	1.16	7.46	6.84	38.01	1.34
10000*10000	100	109.27	0.91	15.82(8)	6.31	86.57	1.15	15.55	6.42	47.87	2.08

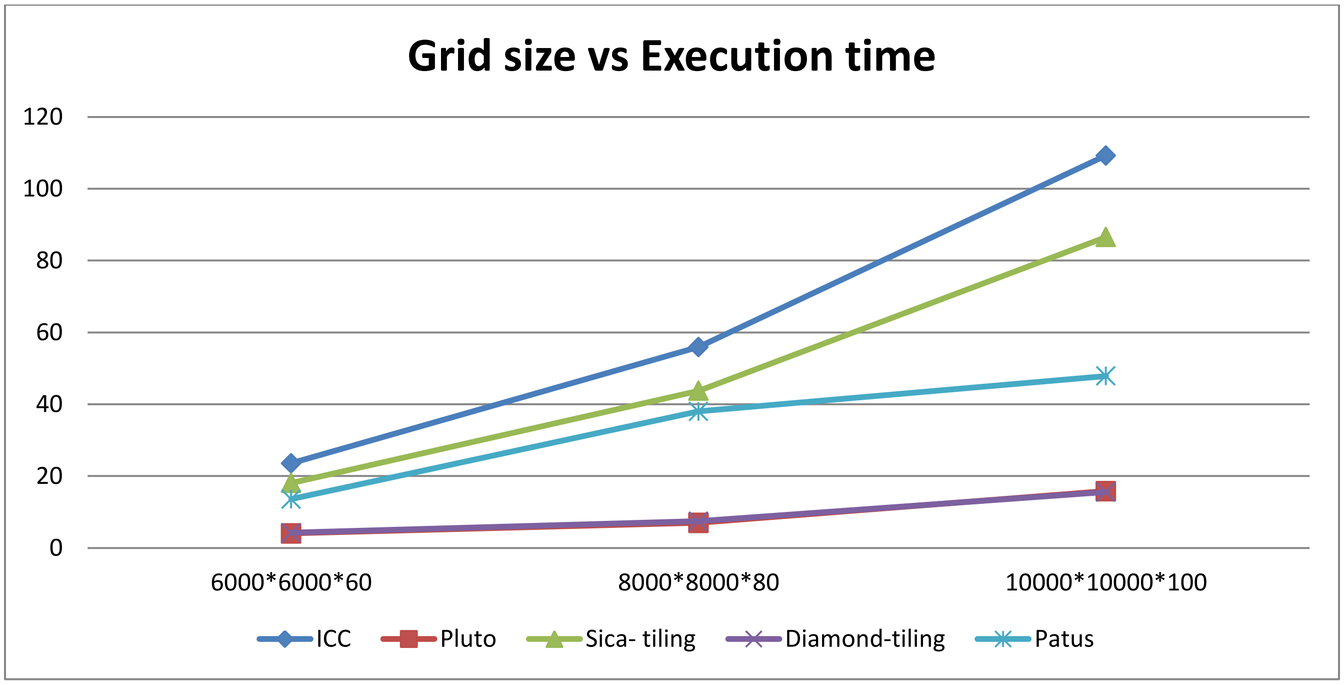


FIGURE 5.19. Grid size versus Execution time

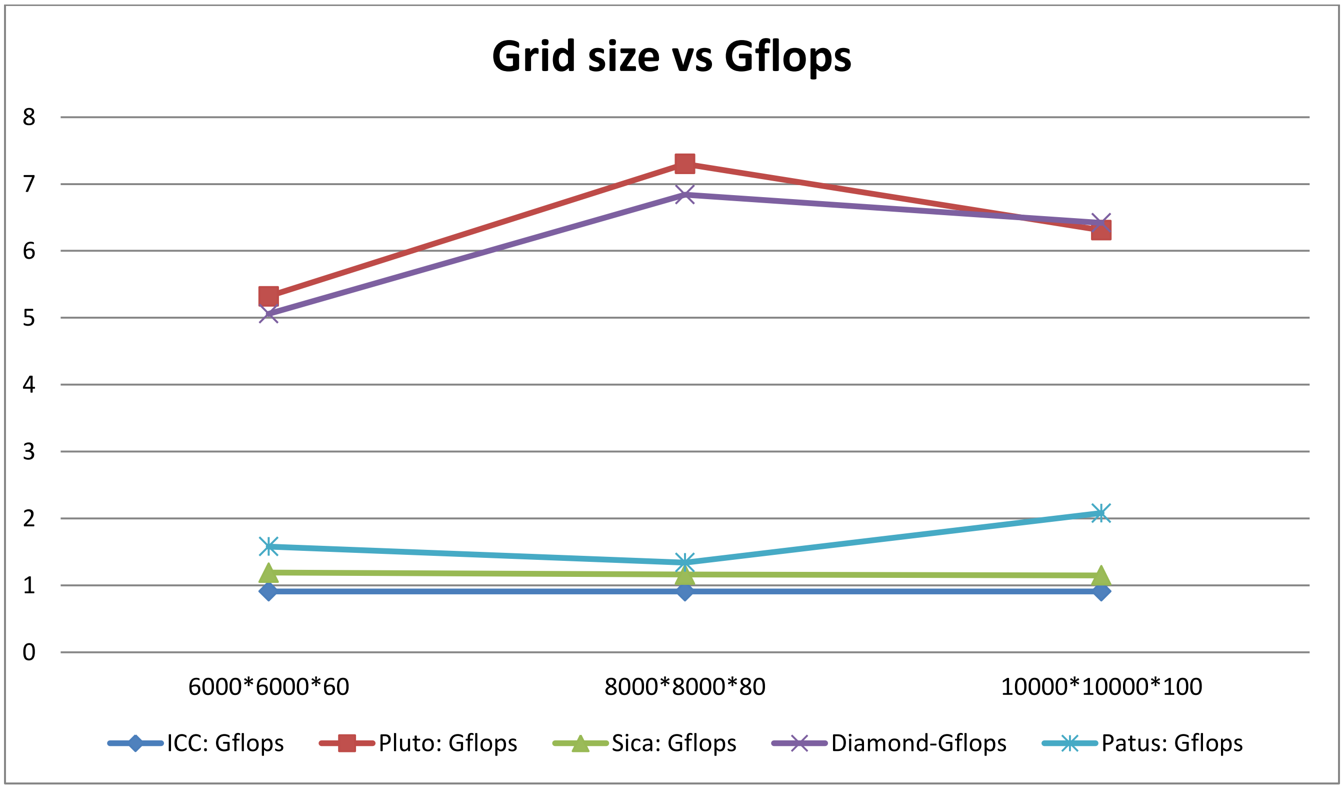


FIGURE 5.20. Grid size versus GFLOPS

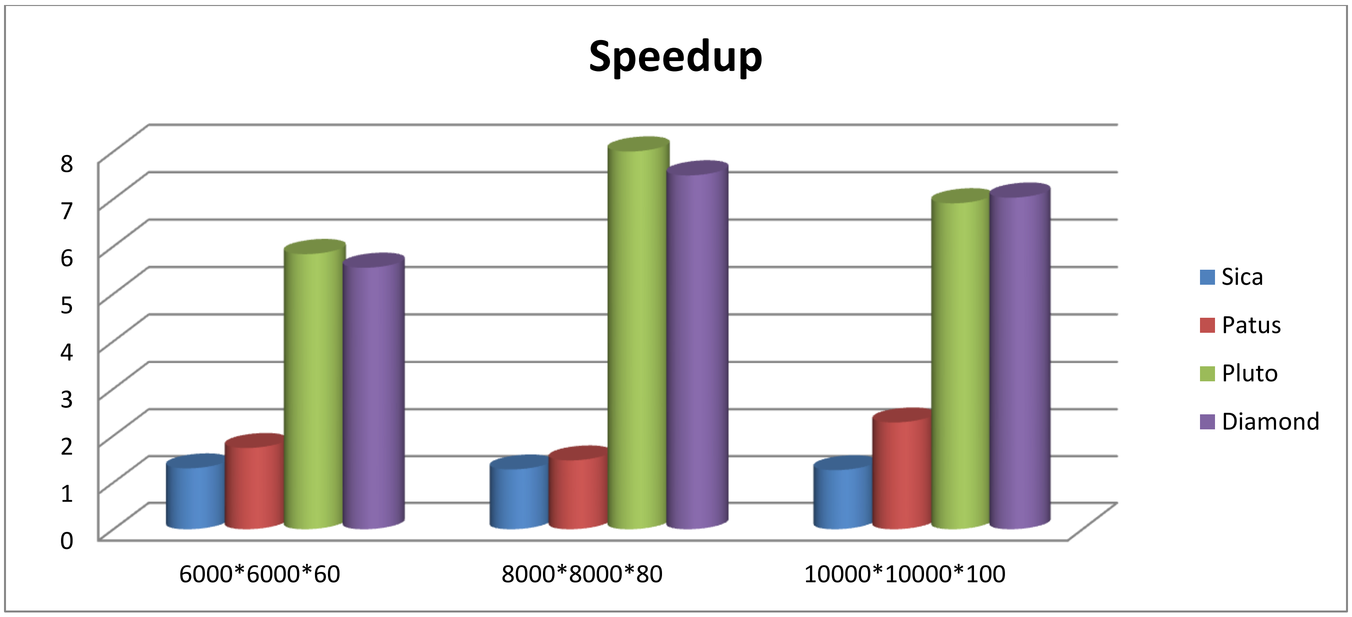


FIGURE 5.21. Performance speedup

- **Game of life stencil:**

Pluto commands:

- polycc –tile –parallel gol.c –o gol.par.c
- polycc –partlbtile –parallel gol.c –o gol.diamond.c
- polycc –tile –sica –parallel gol.c –o gol.sica.c
- icc –O3 –fp-model precise –mavx gol.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp gol.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp gol.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp gol.sica.c –o sica (for sica tiled parallel code)

Patus commands:

- patus -- architecture = x86_64 AVX game-of-life.stc
- make tune height=6000 width=6000 (for grid size 6000*6000)
- ./bench 6000 6000 2997 132 2 3

- make tune height=8000 width=8000 (for gris size 8000*8000)
- ./bench 8000 8000 3997 88 2 3
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 9995 336 2 2

The table 5.11 gives the information of performance of game of life stencil when compiled via pluto and patus.

TABLE 5.11. Performance table for patus and pluto on game of life stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	54.11	0.55	10.01(8)	3.01	48.19	0.62	9.84(8)	3.06	23.58	1.28
8000*8000	80	128.31	0.55	16.19(8)	4.42	113.27	0.64	16.83(8)	4.25	40.29	1.77
10000*10000	100	250.67	0.55	37.24(8)	3.75	238.83	0.67	37.27(8)	3.75	62.26	2.24

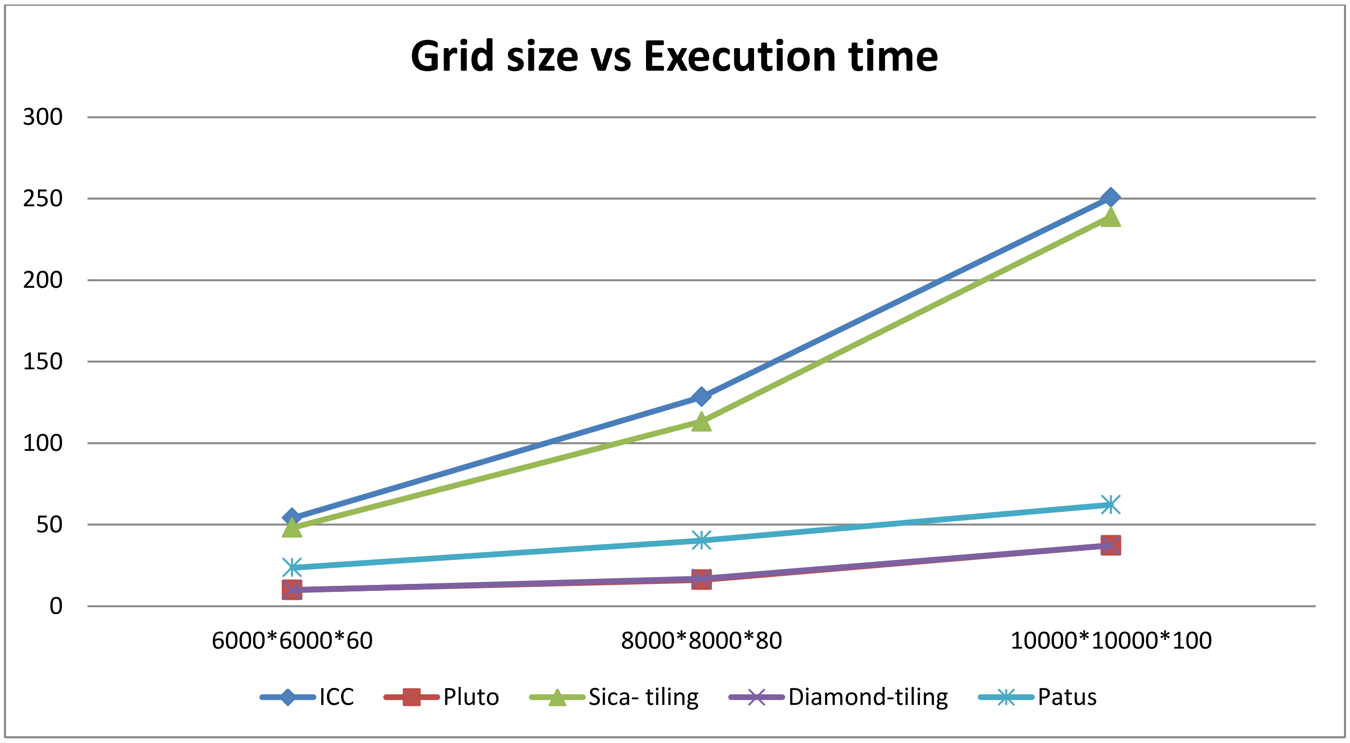


FIGURE 5.22. Grid size versus Execution time

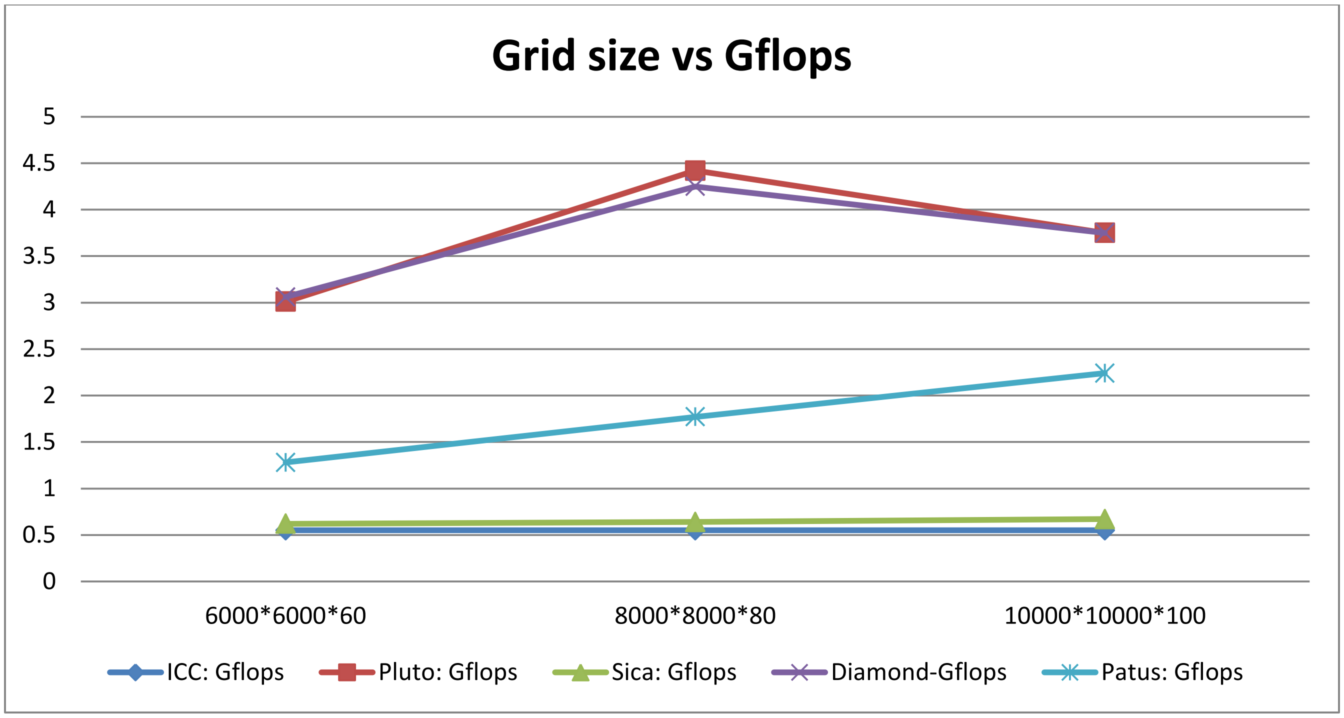


FIGURE 5.23. Grid size versus GFLOPS

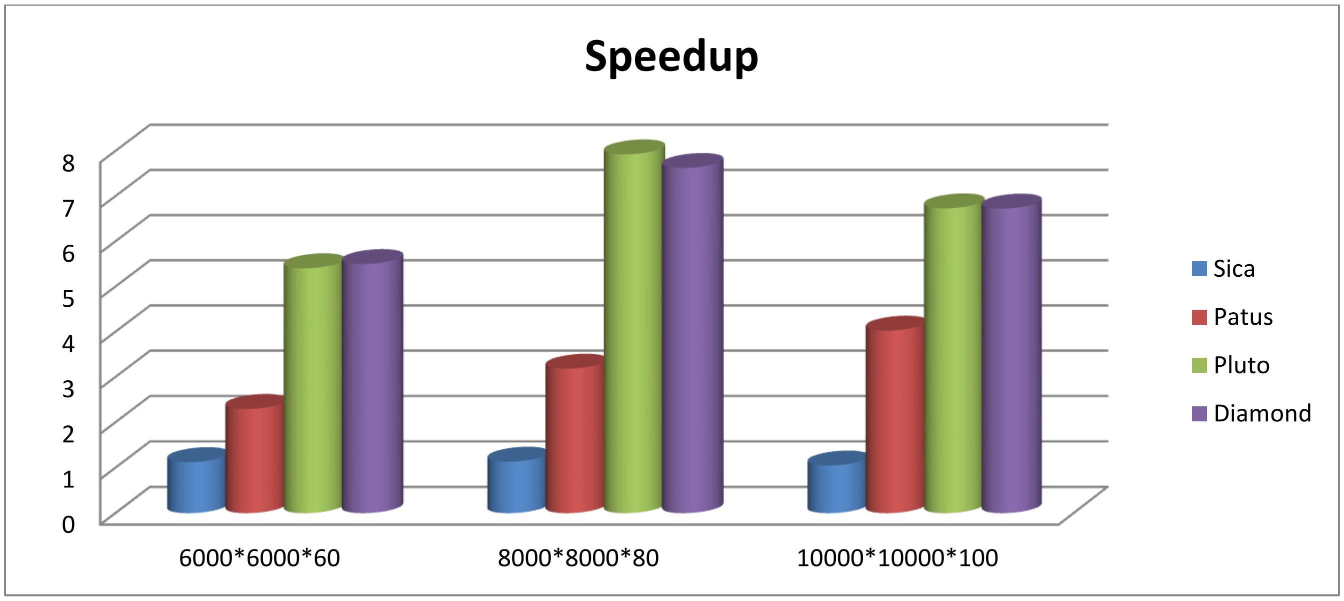


FIGURE 5.24. Performance speedup

- **Seidel stencil:**

Pluto commands:

- polycc –tile –parallel seidel.c –o seidel.par.c

- polycc –partlbtile –parallel seidel.c –o seidel.diamond.c
- polycc –tile –sica –parallel seidel.c –o seidel.sica.c
- icc –O3 –fp-model precise –mavx seidel.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp seidel.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp seidel.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp seidel.sica.c –o sica (for sica tiled parallel code)

Patus commands:

- patus – architecture = x86_64 AVX seidel.stc
- make tune height=6000 width=6000 (for grid size 6000*6000)
- ./bench 6000 6000 1498 324 2 0
- make tune height=8000 width=8000 (for gris size 8000*8000)
- ./bench 8000 8000 1998 184 2 2
- make tune height=10000 width=10000 (for grid size 10000*10000)
- ./bench 10000 10000 2498 72 2 4

The table 5.12 gives the information of performance of seidel stencil when compiled via pluto and patus.

TABLE 5.12. Performance table for patus and pluto for seidel stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
6000*6000	60	20.06	0.96	3.55(8)	5.46	30.46	0.58	3.30(8)	5.87	13.04	1.48
8000*8000	80	47.60	0.96	6.11(8)	7.53	71.19	0.64	5.90(8)	7.79	31.68	1.45
10000*10000	100	93.00	0.96	14.91(8)	6.02	138.44	0.64	13.66(8)	6.57	51.25	1.75

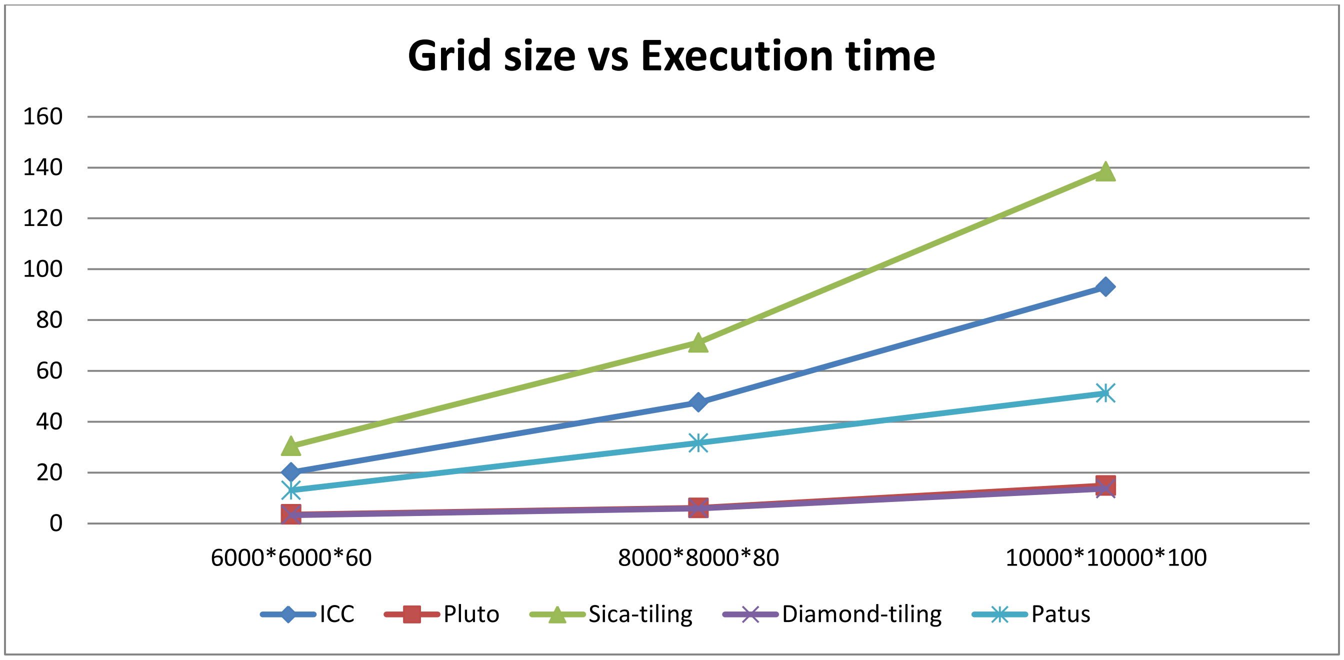


FIGURE 5.25. Grid size versus Execution time

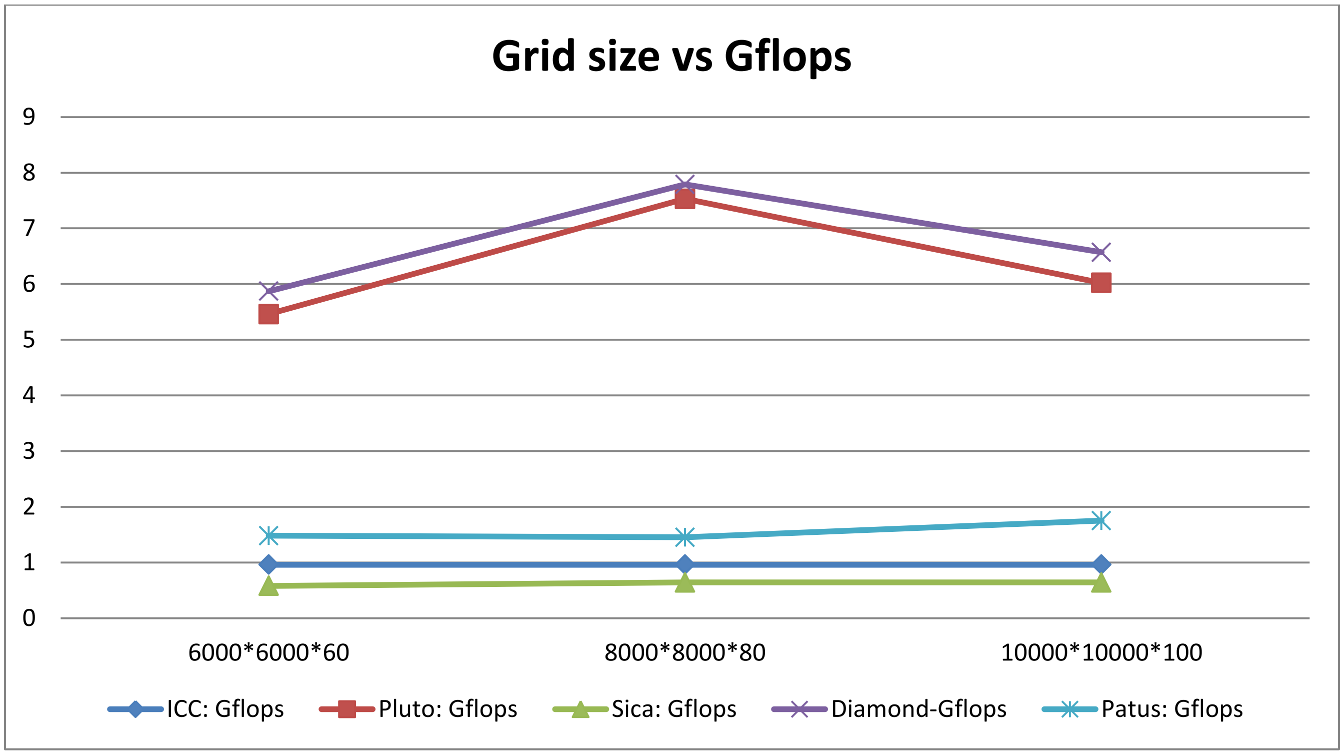


FIGURE 5.26. Grid size versus GFLOPS

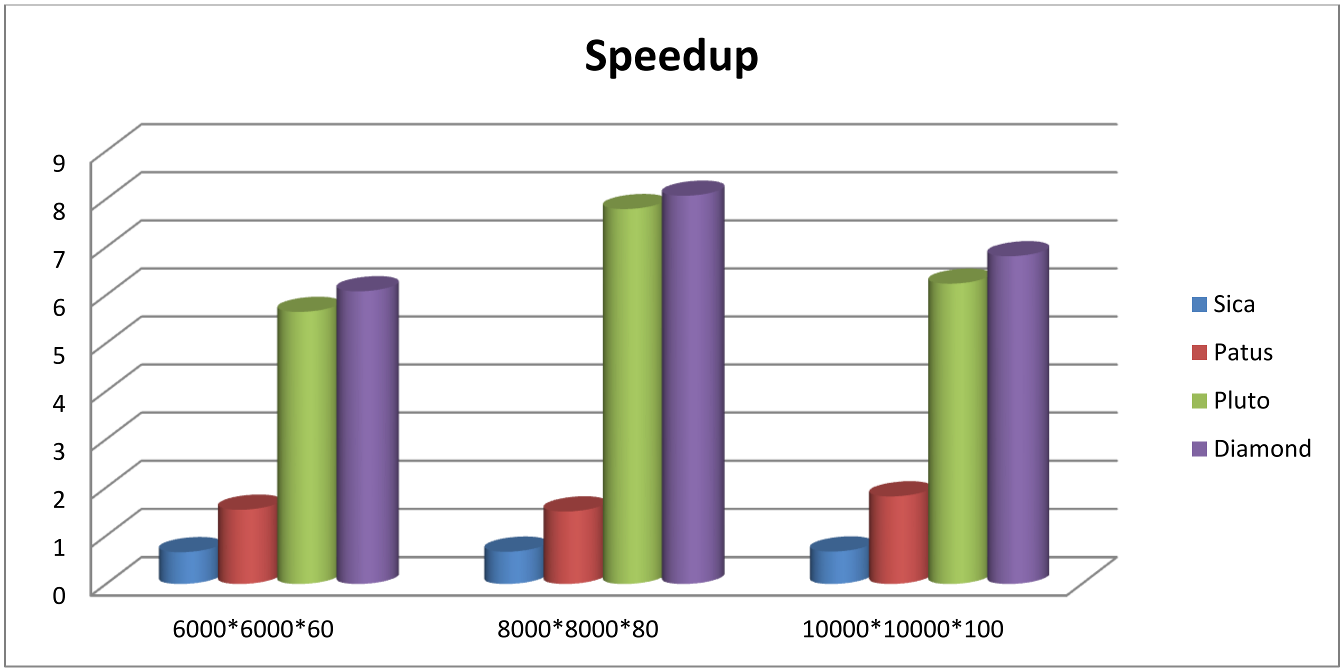


FIGURE 5.27. Performance speedup

- **Laplacian stencil:**

Pluto command:

- polycc –tile –parallel lap.c –o lap.par.c
- polycc –partlbtile –parallel lap.c –o lap.diamond.c
- polycc –tile –sica –parallel lap.c –o lap.sica.c
- icc –O3 –fp-model precise –mavx lap.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp lap.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp lap.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp lap.sica.c –o sica (for sica tiled parallel code)

Patus command:

- patus --architecture = x86_64 AVX laplacian.stc
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)

- ./bench 300 300 300 298 32 4 1 2 1
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 298 32 4 1 2 1
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 298 144 36 4 6 2

The table below gives the information of performance of laplacian stencil when compiled via pluto and patus.

TABLE 5.13. Performance table for patus and pluto on laplacian stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
300*300*300	100	11.15	1.77	2.39(8)	8.56	21.02	0.5	2.60(8)	7.89	7.85	2.32
300*300*300	200	23.14	1.77	4.53(8)	9.06	44.79	0.52	4.49(8)	9.14	16.38	2.50
300*300*300	295	34.14	1.77	6.24(8)	9.70	68.07	0.47	6.67(8)	9.07	25.40	2.38

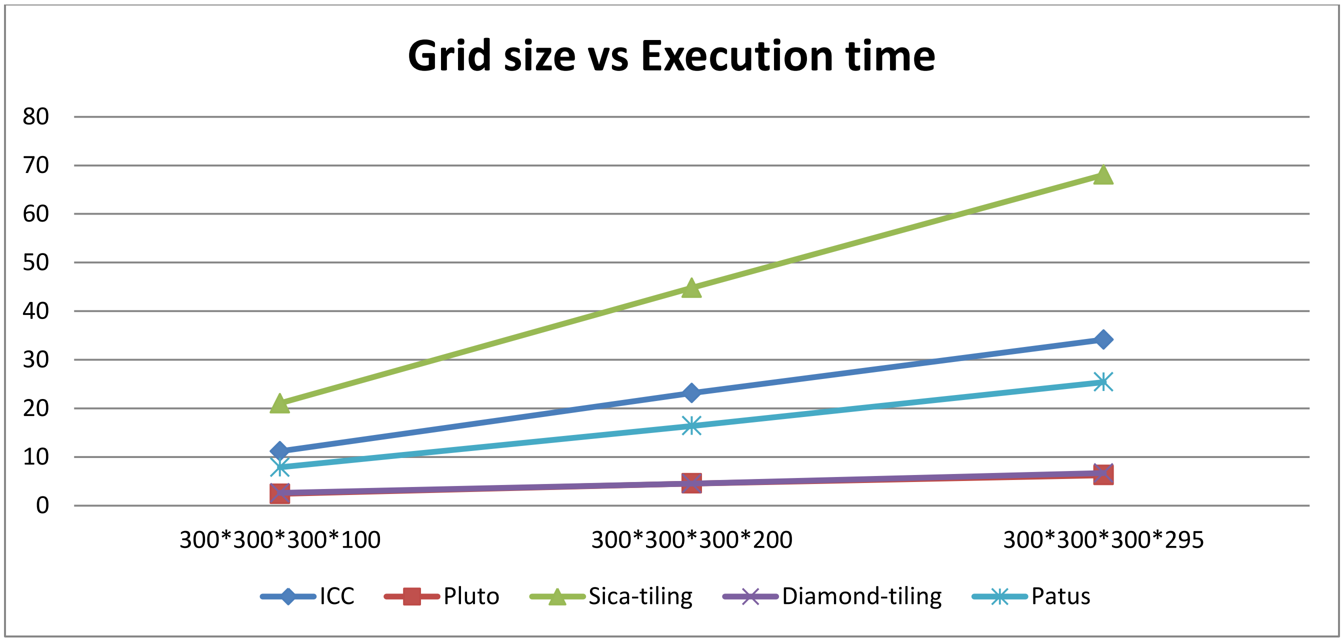


FIGURE 5.28. Grid size versus Execution time

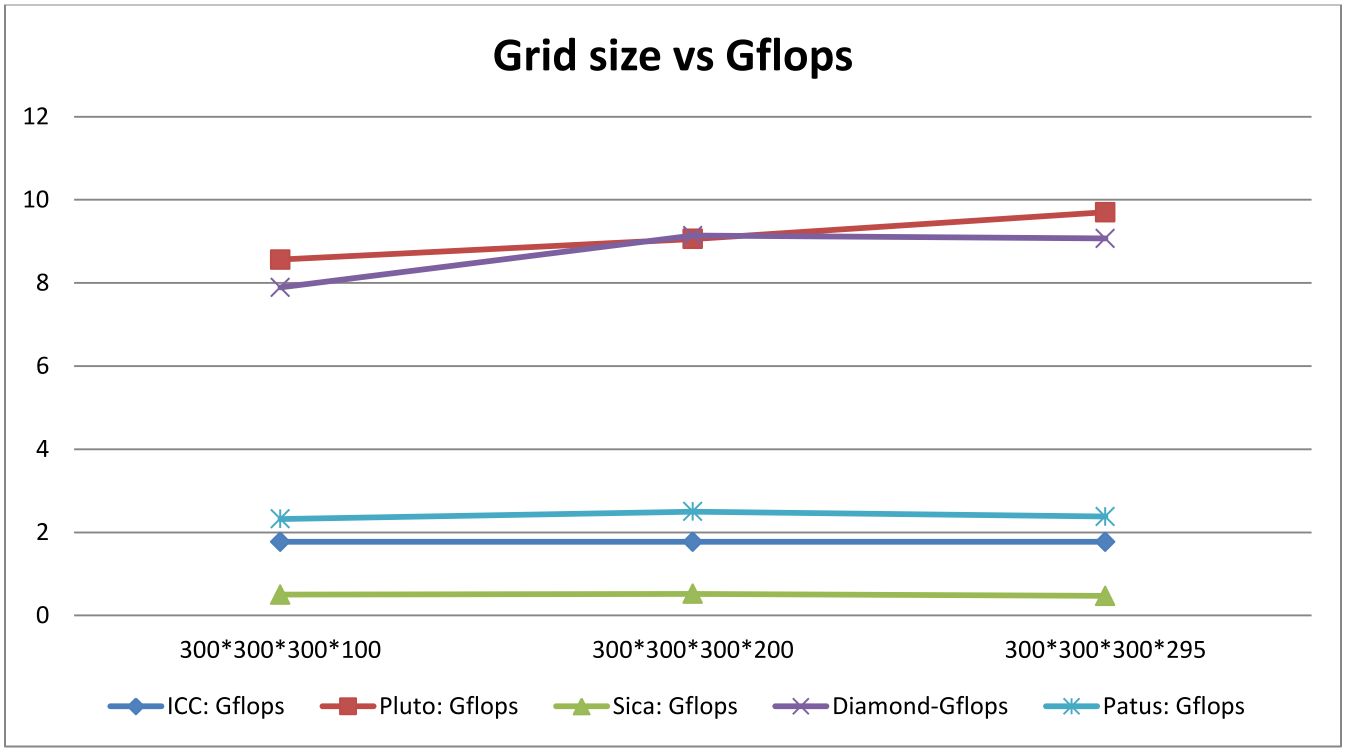


FIGURE 5.29. Grid size versus GFLOPS

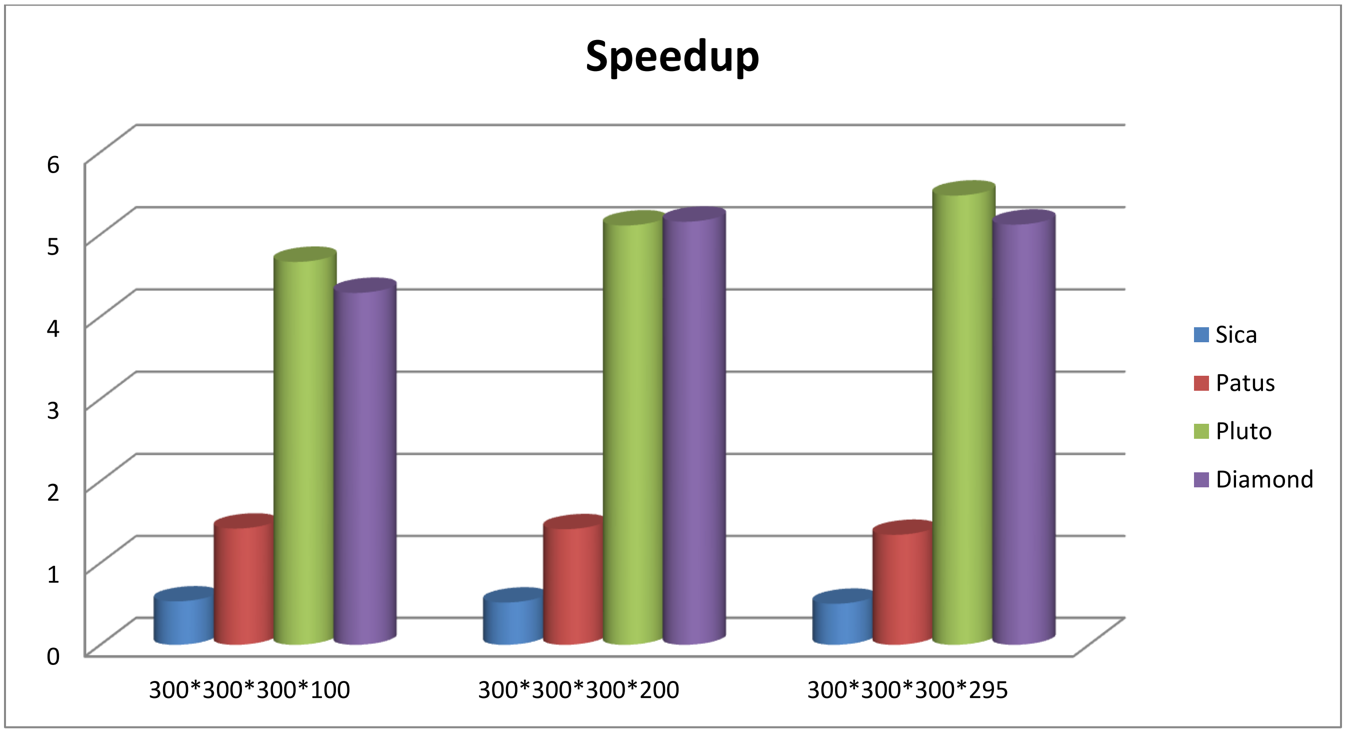


FIGURE 5.30. Performance speedup

- **Wave stencil:**

Pluto command:

- polycc –tile –parallel wave.c –o wave.par.c
- polycc –partlbtile –parallel wave.c –o wave.diamond.c
- polycc –tile –sica –parallel wave.c –o wave.sica.c
- icc –O3 –fp-model precise –mavx wave.c –o orig (for the C code)
- icc –O3 –fp-model precise –mavx –openmp wave.par.c –o par (for pipelined parallel code)
- icc –O3 –fp-model precise –mavx –openmp wave.diamond.c –o diamond (for concurrent parallel code)
- icc –O3 –fp-model precise –mavx –openmp wave.sica.c –o sica (for sica tiled parallel code)

Patus Command:

- patus -- architecture = x86_64 AVX wave.stc
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 299 8 12 1 4 4
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 299 8 12 1 4 4
- make tune x_max=300 y_max=300 z_max=300 (for grid size 300*300*300)
- ./bench 300 300 300 299 8 12 1 4 4

The table 5.14 gives the information of performance of wave stencil when compiled via pluto and patus.

TABLE 5.14. Performance table for patus and pluto on wave stencil

Grid Size	Time	ICC	ICC: Gflops	Pluto	Pluto: Gflops	Sica-tiling	Sica: Gflops	Diamond-tiling	Diamond-Gflops	Patus	Patus: Gflops
300*300*300	100	16.01	3.2	4.97(8)	10.32	63.24	0.5	5.48(8)	9.35	12.66	4.70
300*300*300	200	32.02	3.2	8.64(8)	11.87	123.3	0.52	8.33(8)	12.31	25.75	4.51
300*300*300	295	47.20	3.2	11.88(8)	12.73	169.4	0.56	11.97(8)	12.65	40.23	4.13

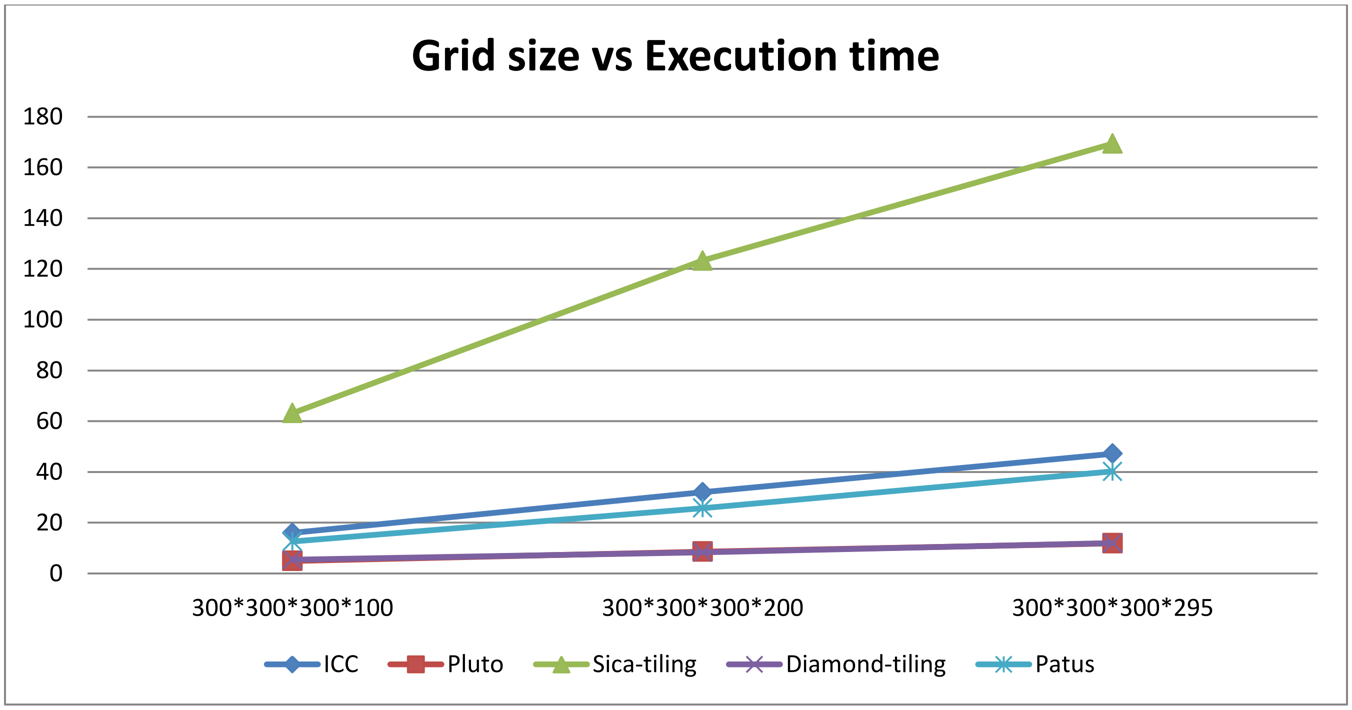


FIGURE 5.31. Grid size versus Execution time

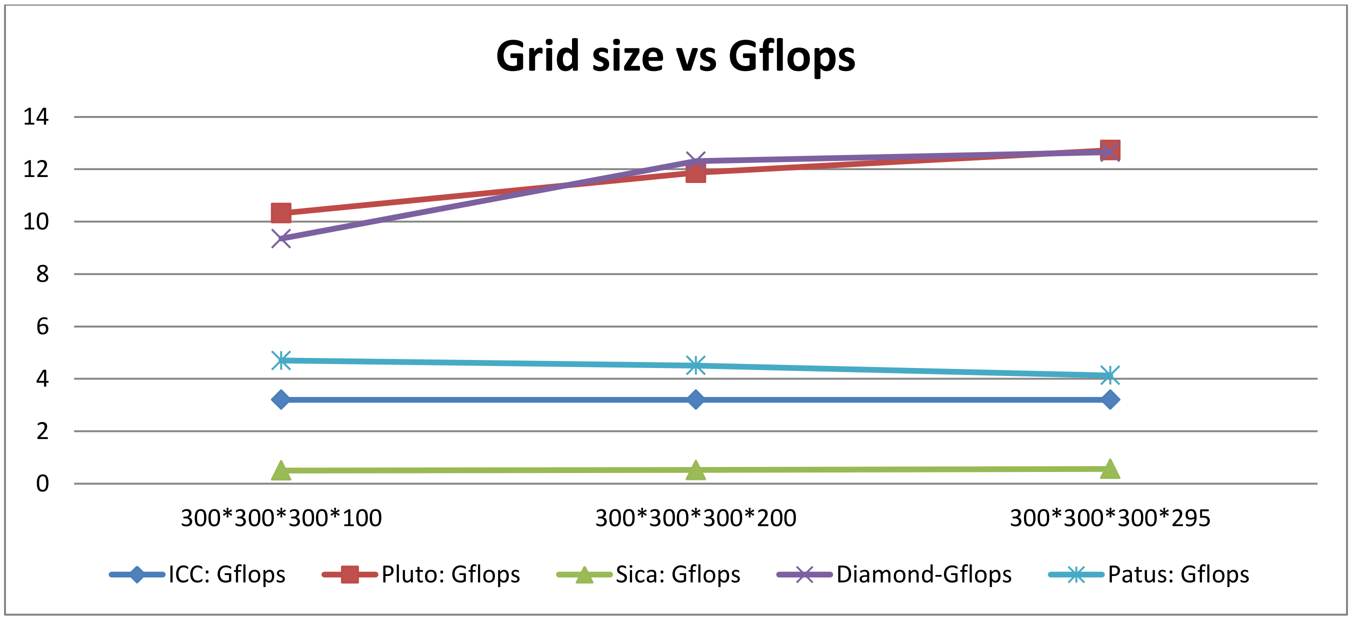


FIGURE 5.32. Grid size versus GFLOPS

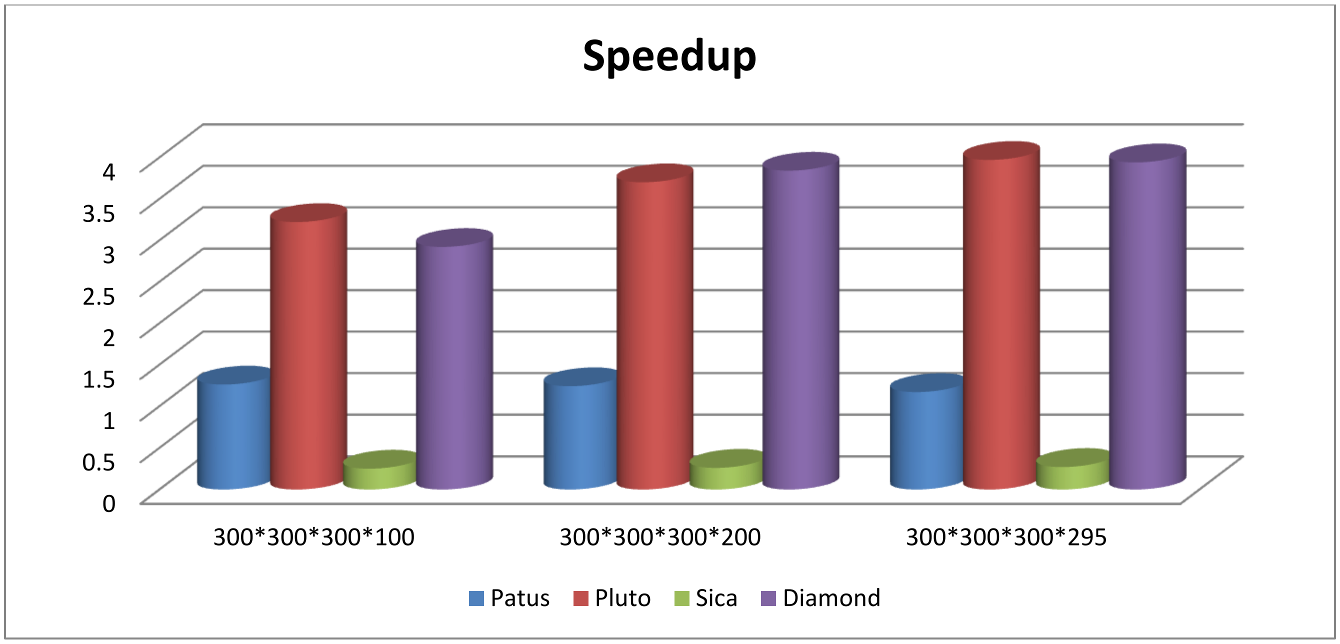


FIGURE 5.33. Performance speedup

Observation and comments:

We can observe from the above performance graphs of stencils that patus is performing considerably slower than pluto's pipelined code. This is because, in contrast to Jacobi style of coding, seidel style of coding has dependencies in subsequent iterations and patus, after tiling the grid sizes is not able to vectorize the inner loop by loop unrolling. One might ask, how pluto was able to generate a good optimized code? And the answer for this is, pluto uses a transformation technique called skewing, which skews the loops to make a certain level of the loop parallel and all dependencies are carried by other levels of the loops. Hence, in seidel style of coding, pluto generated an optimized code in which the outer loops were made parallel for tiling purpose and inner loops carried all the dependencies. This is how pluto was able to generate parallel pipelined executable tiles. Thus, patus was slower with respect to pluto because of its lack of implementing skewing technique.

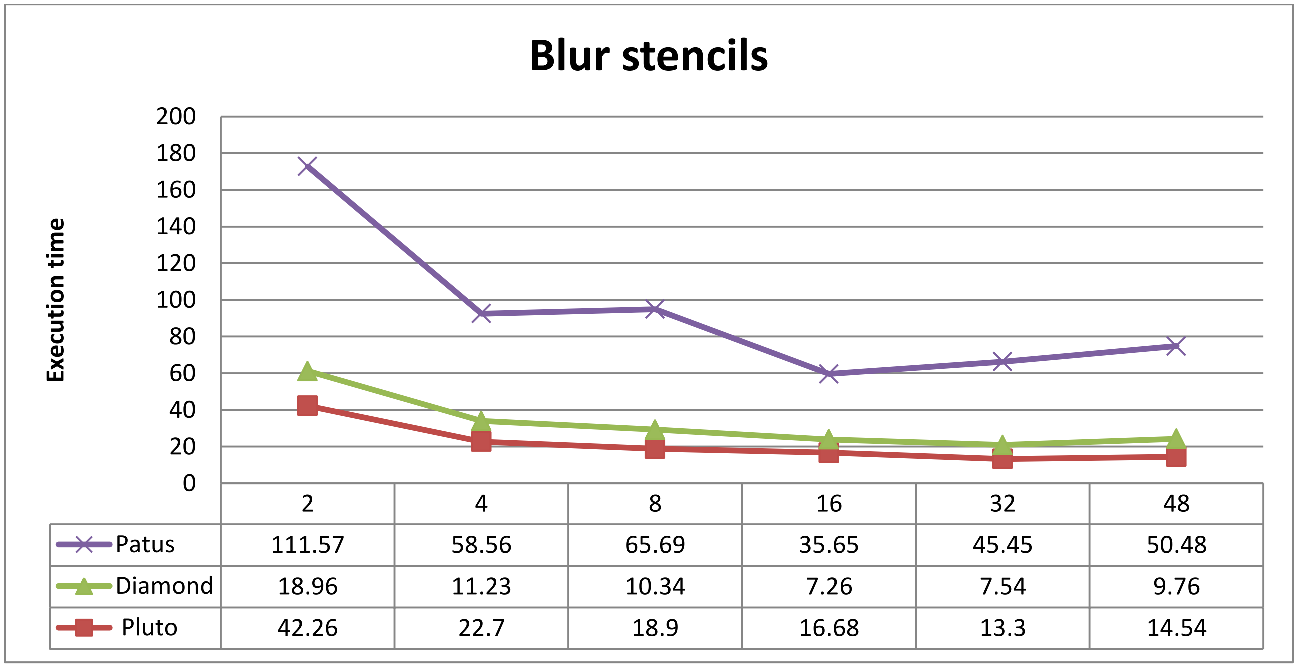
5.4 Execution time vs number of threads

This section provides the information on execution performance of each stencil against the number of threads used to execute the code. We can see that, for all the stencils, the best performance in terms of execution time is achieved for thread counts of 16 or 32.

- **Blur stencil**

TABLE 5.15. Execution time for relative number of threads for blur stencil

Number of threads	Pluto	Diamond	Patus
2	42.26	18.96	111.57
4	22.70	11.23	58.56
8	18.90	10.34	65.69
16	16.68	7.26	35.65
32	13.30	7.54	45.45
48	14.54	9.76	50.48



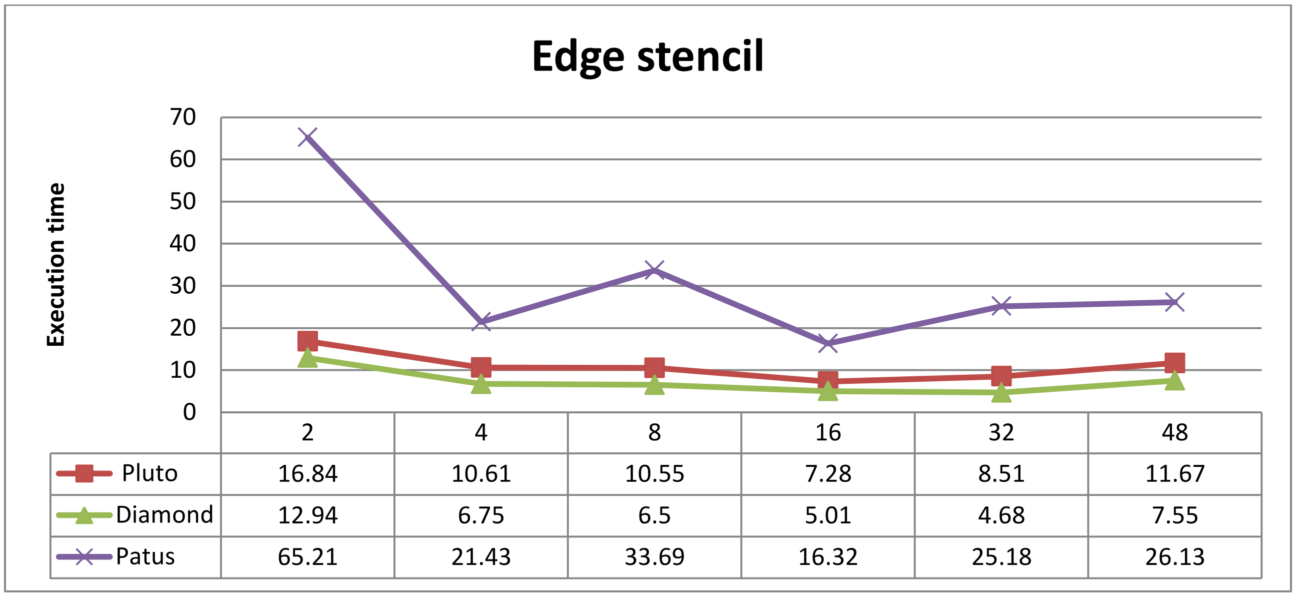
	2	4	8	16	32	48
Patus	111.57	58.56	65.69	35.65	45.45	50.48
Diamond	18.96	11.23	10.34	7.26	7.54	9.76
Pluto	42.26	22.7	18.9	16.68	13.3	14.54

FIGURE 5.34. Number of threads versus Execution time

- **Edge stencil**

TABLE 5.16. Execution time for relative number of threads for edge stencil

Number of threads	Pluto	Diamond	Patus
2	16.84	12.94	65.21
4	10.61	6.75	21.43
8	10.55	6.50	33.69
16	7.28	5.01	16.32
32	8.51	4.68	25.18
48	11.67	7.55	26.13



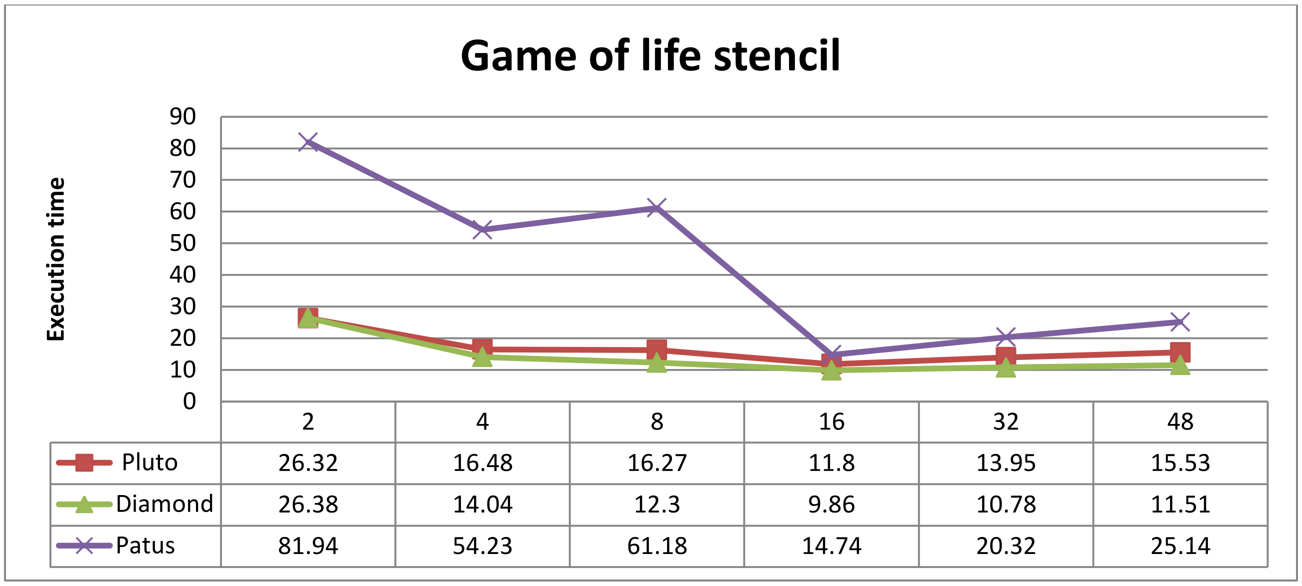
	2	4	8	16	32	48
Pluto	16.84	10.61	10.55	7.28	8.51	11.67
Diamond	12.94	6.75	6.5	5.01	4.68	7.55
Patus	65.21	21.43	33.69	16.32	25.18	26.13

FIGURE 5.35. Execution time versus number of threads

- **Game of life**

TABLE 5.17. Execution time for relative number of threads for game of life stencil

Number of threads	Pluto	Diamond	Patus
2	26.32	26.38	81.94
4	16.48	14.04	54.23
8	16.27	12.30	61.18
16	11.80	9.86	14.74
32	13.95	10.78	20.32
48	15.53	11.51	25.14



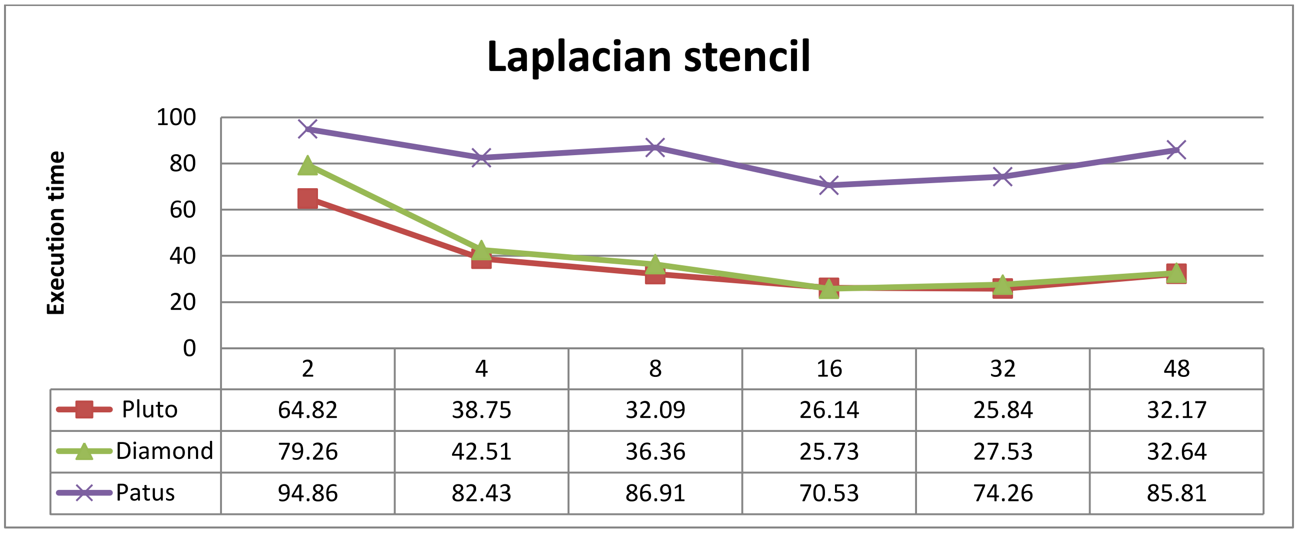
	2	4	8	16	32	48
🟥 Pluto	26.32	16.48	16.27	11.8	13.95	15.53
▲ Diamond	26.38	14.04	12.3	9.86	10.78	11.51
✕ Patus	81.94	54.23	61.18	14.74	20.32	25.14

FIGURE 5.36. Execution time versus number of threads

- **Laplacian**

TABLE 5.18. Execution time for to number of threads for laplacian stencil

Number of threads	Pluto	Diamond	Patus
2	64.82	79.26	94.86
4	38.75	42.51	82.43
8	32.09	36.36	86.91
16	26.14	25.73	70.53
32	25.84	27.53	74.26
48	32.17	32.64	85.81



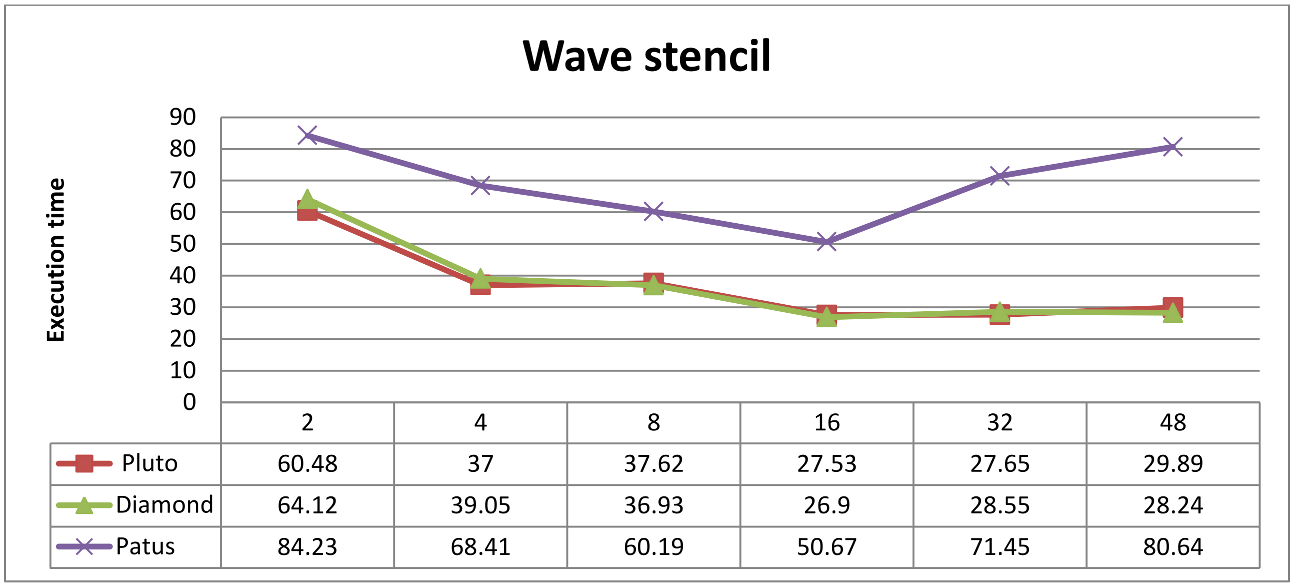
	2	4	8	16	32	48
🟥 Pluto	64.82	38.75	32.09	26.14	25.84	32.17
▲ Diamond	79.26	42.51	36.36	25.73	27.53	32.64
✕ Patus	94.86	82.43	86.91	70.53	74.26	85.81

FIGURE 5.37. Execution time versus number of threads

- **Wave**

TABLE 5.19. Execution time for number of threads for wave stencil

Number of threads	Pluto	Diamond	Patus
2	60.48	64.12	84.23
4	37.00	39.05	68.41
8	37.62	36.93	60.19
16	27.53	26.90	50.67
32	27.65	28.55	71.45
48	29.89	28.24	80.64



	2	4	8	16	32	48
Pluto	60.48	37	37.62	27.53	27.65	29.89
Diamond	64.12	39.05	36.93	26.9	28.55	28.24
Patus	84.23	68.41	60.19	50.67	71.45	80.64

FIGURE 5.38. Execution time versus number of threads

Chapter 6 Conclusion

In conclusion, this report provides the performance of both patus and pluto compilers on stencils using two different styles of coding i.e. Jacobi style and Seidel style. In Jacobi style of coding, pluto compiler was able to perform better than patus compiler because patus compiler lacks time blocking (tiling) and only blocks space (grid sizes) whereas pluto tiles both time and space. In Seidel style of coding, we observed that patus compiler is performing much slower than pluto compiler because of the given nature of dependencies in Seidel style of coding. Pluto compiler uses skewing transformation technique to make certain loop parallel and hence it is able to perform well with Seidel style of coding. Patus compiler lacks the use of skewing technique in its implementation to make parallel loop and hence patus compiler is not suited for the codes with Seidel style. In all of our experiments, pluto compiler, with some fine tuning for finding the appropriate tile sizes and the use of different tile shapes, has outperformed patus compiler on every stencil. Therefore, for optimizing stencils and to be able to get a good optimized code in terms of data locality and parallelism, the use of pluto compiler is a good choice than patus compiler.

References

Bondhugula, U., Baskaran, M. M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., & Sadayappan, P. (2008). Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. doi:10.1007/978-3-540-78791-4_9.

Bondhugula, U., Hartono, A., Ramanujam, J., & Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer.

Bondhugula, U., Pananilath, I., & Bandishti, V. (2012). Tiling stencil computations to maximize parallelism.

Bondhugula, U., Ramanujam, J., & Sadayappan, P. (2008). PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System.

Christen, M., Schenk, O., & Burkhart, H. (2011). PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures.

Dustin, F., Thomas, S., Michael, J., & Sven, M. (2013). Facilitate simd-code-generation in the polyhedral model by hardware-aware automatic code-transformation. doi: urn:nbn:de:0011-n-2347222.

Vita

Pratik Prabhu Hanagodimath was born in Haveri, India, to Prabhu Hanagodimath and Premalata Hanagodimath. Pratik studied electrical and electronics engineering at SDM College of Engineering and Technology, Visvesvaraya Technological University, India, from 2006 through 2010 towards obtaining his Bachelor of Engineering degree. He worked for six months in a company called JSW steel ltd., India, from January 2011 to June 2011. He is currently a Masters student in the Department of Electrical & Computer Engineering at Louisiana State University, Baton Rouge, where he has been a graduate student since fall 2011. During this time he has been working as a research assistant in the field of compiler optimization under the guidance of Dr J Ramanujam.