On the Composition and Exponentiation of Stencil Kernels – Improving Performance and Energy Consumption

Daniel Rubio Bonilla
High Performance Computing
Center Stuttgart (HLRS)
University of Stuttgart
Stuttgart, Germany
rubio@hlrs.de

Manuel Carro Liñares
IMDEA Software
Madrid, Spain
manuel.carro@imdea.org

Colin W. Glass
High Performance Computing
Center Stuttgart (HLRS)
University of Stuttgart
Stuttgart, Germany
glass@hlrs.de

ABSTRACT

TODO: a native speaker should check the whole document.

Many High-Performance Computing (HPC) programs perform array or n-dimensional matrix computations that involve a class of kernels known as stencils that characterize for updating every element according to some fixed pattern. Stencils are commonly used in solving partial differential equations (used in scientific problems), image processing and geometric modeling.

In this paper we present a method to combine stencil kernel(s) that are executed iteratively, or subsequently, into one kernel to reduce the number of iterations or number of stencil kernels applied. Our empirical results show that in most cases and for different hardware families, ranging from embedded systems to HPC this strategy improves performance and reduces energy consumption.

TODO: check for the right categories

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

HPC, stencil, optimization, energy, performance, algorithms

1. INTRODUCTION

Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern, called stencil[4]. In numerical analysis a stencil is a geometric arrangement of a nodal group relative to the point of interest used to offer a numerical solution of partial differential equations or apply filters to image among other uses. For this reason they are often found in computer scientific and engineering simulation code, *e.g.* for computational fluid dynamics.

Stencil codes perform a sequence of repetitive steps through a given array or matrix that generally is a 2- or 3-dimensional regular grid[3]. In each step the stencil code updates all array elements, called cells, with a new value that is computed using for the calculation the neighbor array elements in a fixed pattern[10]. In most cases boundary values are left unchanged, but in some cases those need to be adjusted during the course of the computation as well. Since the stencil computation is the same for each element, the pattern of data accesses is repeated[5]; computationally a stencil represent a pattern of accessing to memory, as for calculating the next step value of an element in a grid, it will need its current value and the current value of a set of neighbors defined by its distance to them.

It is important to increase the performance and reduce the energy consumption of the stencil kernels, not only because they are very relevant and commonly used in image processing, and scientific and engineering codes, but also because they often result in a high percentage of the total computational resources of these applications.

In this paper we describe a new technique that is based on mathematical understanding and manipulation of the stencil mathematical expressions, that as shown by empirical results it often leads to shorter execution times and lower energy usage. In Section 3 we describe the process of stencil exponentiation and composition then in Section 4 we present a use case and apply this technique to 1-, 2- and 3-dimensional stencils and analyze their theoretical computational cost. In Section 5 the results of the benchmarks are exposed and analyzed. And finally, in Section 6 the usability and improvements on this technique are discussed.

2. RELATED WORK

extend and detailed it more?

Due to the high relevance of stencil computations many approaches have been develop to accelerate and improve the performance of these algorithms. Most approaches are based on improving the data locality [7], such as blocking data in cache so that it can be reused [11], or and changing the computational order to allow vectorization. Both of these approaches are commonly used like, for example, by the PLUTO tool, an automatic parallelization tool based on the polyhedral model, which tunes the code to exploit data locality and vectorization, introduce memory management optimizations and perform automatic parallelization [1] amont others.

Up to now the most common ways of optimizing stencil codes played with how the code is executed. The approach described in this work is manipulating the mathematical formulation of the stencil. This implies that the technique here described can be also complemented with the common optimization strategies for stencils.

Move pluto benchmarks to results?

To analyze the feasibility of combining both approaches we took a 1-D stencil given as part of the PLUTO examples and we composed it. Then to both approaches we used PLUTO to optimize data locality, vectorize the code and parallelize it, resulting in four different codes; the original code, the composed code, the original code after PLUTO and the composed code after PLUTO. Figures 1 and 2 show the execution time and energy consumed to execute the different versions of the stencil on different hardware.

correct pictures

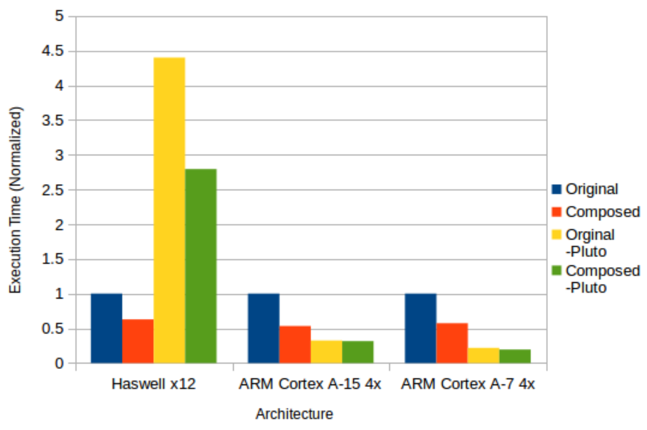

Figure 1: Pluto and Stencil Composition - Performance (Normalized)

Maybe remove this note?

We used the 1-D stencil example given by PLUTO as we failed to generate correct code from our more complex stencil codes that were used for the other benchmarks.

3. STENCIL COMPOSITION

TODO: improve the mathematical formalization of the problem and solution

Function Composition, in computer science, is a way to

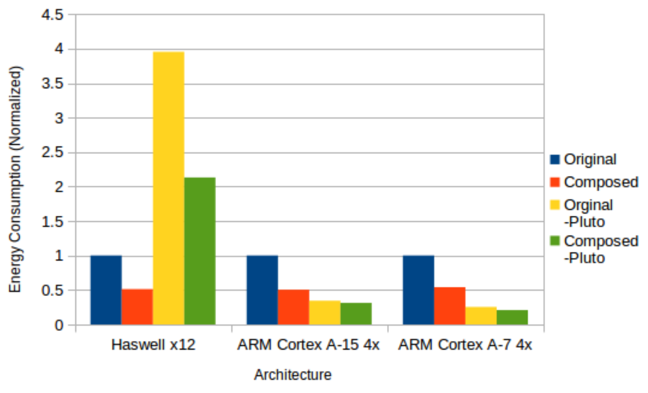

Figure 2: Pluto and Stencil Composition - Energy (Normalized)

combine simple functions[1] to build more complex functions, where the result of each function is passed as the argument of the next function, and the result of the last one is the result of the whole[9]. Often developers directly apply the result of one function as the argument(s) of another function a feature that is allowed by most programming languages because the ability to easily compose functions encourages factoring functions for maintainability and code reuse. In the next C code `z1` and `z2` are equivalent.

```
y  = g(x);
z1 = f(y);

z2 = f(g(x));
```

In functional languages (Haskell is used for this example), the function composition can be expressed as:

```
z = f . g
```

using the built-in operator (.), which can be read as *f after g* or *g composed with f*. Where one of the possible definitions of the (.) operator is:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

In our context, a stencil code is a function that is usually applied to all the elements of a n-dimensional array, taking as parameters a fixed pattern of neighbors elements, and the result is used to update the value of the element to which it is applied. In many applications different stencil functions are applied subsequently or the same stencil function is applied iteratively. Composing the stencils functions result on a new equivalent function with a different (broader) geometrical pattern over the input data, but reducing the number of iterations over the data array.

3.1 Composition of Different Stencils

Given an application that uses two different stencil functions $g()$ and $f()$ that are applied to all elements of an n-dimensional array $V[]$ with VN elements so that $V'[] =$

[1]Real-valued functions over a subset of the real-line that are measurable

$g(V[])$ and $V''[] = f(V'[]) = f(g(V[]))$ then we can define the function $z()$ so that $V''[] = z(V[])$.

Being $a_i[]$ the set of elements of $V[]$ that are the input parameters for $g()$ to obtain the value of an specific element of $V'[]$, $V'[i]$, so that $V'[i] = g(a_i[])$ and being $b_j[]$ the set of elements of $V'[]$ that are the input parameters for $f()$ to obtain the value of an specific element of $V''[]$, $V''[j]$, so that $V''[j] = f(b_j[])$. Then we can replace each element of $b_j[]$ by the pair of function $f()$ and input parameters of the array $V[]$, $a_{ij}[]$ that give each element of $b_j[]$. Replacing them in $g()$ allows us to obtain $z()$.

In summary, the composed stencil function $z()$ is obtained by replacing the parameters of $g()$ with their dependence on function $f()$ on the original elements of the array. Lets see if with an example. If
$V'[i] = f(V[i-1], V[i], V[i+1]) = V[i-1] + V[i+1] - V[i]$
and
$V''[i] = g(V'[i-1], V'[i], V'[i+1]) = V'[i] - V'[i-1] - V'[i+1]$ then each element $V'[i-1]$, $V'[i]$ and $V'[i+1]$ can be replaced with their equivalents on depending on $f()$:
$g(V'[i-1], V'[i], V'[i+1]) = V[i-1] + V[i+1] - V[i] - V[i-2] - V[i] + V[i-1] - V[i] - V[i+2] + V[i+1] = z(V[i-2], V[i-1], V[i], V[i+1], V[i+2])$

3.2 Stencil Exponentiation

Stencil Exponentiation is a particular case of stencil composition. Using the same approach, instead of composing two different functions, we compose the same function with itself. Given the stencil function f that is applied iteratively over a particular data-set, it can be composed with itself, $f^2 = f.f$, resulting in a powered stencil function that has to be applied half of the times to obtain the same result.

4. STENCIL COMPOSITION APLIED TO A USE CASE

The heat equation is a parabolic partial differential equation that describes the variation in temperature in a given region over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \qquad (1)$$

where u is the temperature at a certain location, t is the time variable, α is the thermal diffusivity (a positive constant of the material) and ∇ denotes the Laplace operator[8].

In this work we examine the performance of the explicit heat equation –in 1-, 2- and 3-dimensions–, before and after composing it with itself (*i.e.* exponentiation of the computational stencil kernel). First we will demonstrate on how to compose the equation from the algorithmic point of view, then we will analyze the computational cost of the different versions and to finalize, in Section 5 we will show the performance and energy benchmarks on different platforms (different HPC and embedded systems).

4.1 Generation of the composed stencil code

The discretization of the partial differential equation (1) for the different n-dimensional cases that we are going to analyze is done using the Finite Difference Method (FDM), which is one of the dominant approaches to obtain numerical solutions of partial differential equations[2].

For this work We considered a *phantom halo* with null values around the original data-set so that the same stencil function, conventional or composed, can be applied to all the elements of original data. This way the need to create special functions for the borders and corners of the n-dimensional matrices is avoided. The concrete details of the tests are exposed later in Section 5.

4.1.1 1-D Composed Stencil

The discretization of the equation (1) for a 1-Dimensional space can be expressed as

$$b[i] = a[i] + D(a[i-1] - 2a[i] + a[i+1]) \qquad (2)$$

where D is the thermal diffusivity, the vector $a[]$ holds the input data and the vector $b[]$ stores the result of the application of the stencil over the elements of $a[]$. Figure 3 shows the accessed data-pattern for reading (black and red dots in the $a[]$ data array) and data written (red dot in the $b[]$ data array). It is not possible to read and write directly to the array $a[]$ because the computation $a[i+1]$ depends on the data in $a[i]$ before applying the stencil computation to $a[i]$ (*i.e.* there is a Read After Write –RAW– data dependency between all elements of $a[]$).

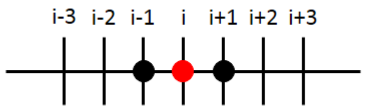

Figure 3: 1D Stencil – read from black and red dot positions, write to red dot position

If we define the function $s()$ as the function that applied to $a[]$ returns $b[]$ so that $b[] = s(a[])$, then it can also define t as $s^2()$ so that

$$c[] = t(a[]) = s^2(a[]) = s(s(a[])) = s(b[]) \qquad (3)$$

i.e. $c[]$ is the array that stores the results of applying twice the original stencil kernel to all the elements of the original data-set $a[]$. To generate the function t and knowing that it is equivalent to $s(b[])$ as seen in equation (3), then we can express it for a particular element of $a[]$ as

$$t(a[i]) = s(b[i]) = b[i] + D(b[i-1] - 2b[i] + b[i+1]) \qquad (4)$$

and knowing that

$$b[i-1] = a[i-1] + D(a[i-2] - 2a[i-1] + a[i]) \qquad (5)$$

$$b[i+1] = a[i+1] + D(a[i] - 2a[i+1] + a[i+2]) \qquad (6)$$

with their relative data access pattern depicted in Figure 4 for $b[i-1]$ and for $b[i+1]$,

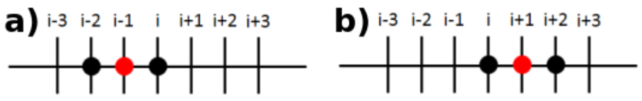

Figure 4: 1D Stencil – relative data access pattern for $b[i-1]$ and $b[i+1]$

we can apply the substitutions of $b[i-1]$, $b[i]$ and $b[i+1]$ in equation (4) resulting in

$$\begin{aligned} t(a[i]) &= c[i] \\ c[i] &= a[i] + D(a[i-1] - 2a[i] + a[i+1]) \\ &+ D(a[i-1] + D(a[i-2] - 2a[i-1] + a[i]) \\ &- 2(a[i] + D(a[i-1] - 2a[i] + a[i+1])) \\ &+ a[i+1] + D(a[i] - 2a[i+1] + a[i+2])) \end{aligned} \tag{7}$$

After the direct substitution of the terms, eliminating the intermediary step $b[]$, the equation (7) can be simplified to the equation (8).

$$\begin{aligned} c[i] &= D^2(a[i-2] + a[i+2]) \\ &+ (2D - 4D^2)(a[i-1] + a[i+1]) \\ &+ (1 - 4D + 6D^2)a[i]; \end{aligned} \tag{8}$$

The new data access pattern of equation (8) is graphically represented in Figure 5. It can be observed that the geometrical pattern of input data has broadened to include all the elements of the access pattern needed to compute $b[i-1]$, $b[i]$, $b[i+1]$.

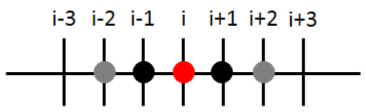

Figure 5: 1D Stencil – Composed Heat Equation Stencil data access pattern

4.1.2 2-D Composed Stencil

The discretization of the equation (1) for a 2-Dimensional space can be expressed as

$$\begin{aligned} b[j,i] &= a[j,i] + D(a[j-1,i] + a[j,i-1] - 4a[j,i] \\ &+ a[j,i+1] + a[j+1,i]) \end{aligned} \tag{9}$$

where D is the thermal diffusivity, the vector $a[]$ holds the input data and the vector $b[]$ stores the result of the application of the stencil over the elements of $a[]$. Figure 7 shows the accessed data for reading (black and red dots in the $a[]$ data array) and data written (red dot in the $b[]$ data array).

As we have seen in Section 4.1.1, it is not possible to read and write directly to the array $a[]$ because of the algorithmic RAW data dependencies between the elements of $a[]$.

Similarly as in the 1-Dimensional case we can define the function $s()$ as the function that applied to $a[]$ returns $b[]$ so that $b[] = s(a[])$, then it can also define t as $s^2()$ so that

$$c[] = t(a[]) = s^2(a[]) = s(s(a[])) = s(b[]) \tag{10}$$

where $c[]$ is the array that stores the results of applying twice the original stencil kernel to all the elements of the original data-set $a[]$. As previously seen, to generate the function t and knowing that it is equivalent to $s(b[])$ as seen in equation (16), then we can express it for a specific element of $a[]$ as

$$\begin{aligned} t(a[j,i]) &= s(b[j,i]) = b[j,i] + D(b[j-1,i] \\ &+ b[j,i-1] - 4b[j,i] + b[j,i+1] + b[j+1,i]) \end{aligned} \tag{11}$$

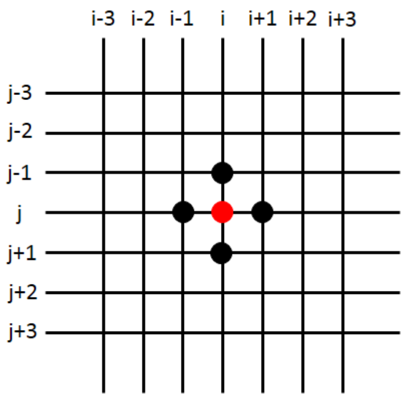

Figure 6: 2D Stencil – read from black and red dot positions, write to red dot position

and knowing that

$$\begin{aligned} b[j-1,i] &= a[j-1,i] + D(a[j-2,i] + a[j-1,i-1] \\ &- 4a[j-1,i] + a[j-1,i+1] + a[j,i]) \end{aligned} \tag{12}$$

$$\begin{aligned} b[j,i-1] &= a[j,i-1] + D(a[j-1,i-1] + a[j,i-2] \\ &- 4a[j,i-1] + a[j,i] + a[j+1,i-1]) \end{aligned} \tag{13}$$

$$\begin{aligned} b[j,i+1] &= a[j,i+1] + D(a[j-1,i+1] + a[j,i] \\ &- 4a[j,i+1] + a[j,i+2] + a[j+1,i+1]) \end{aligned} \tag{14}$$

$$\begin{aligned} b[j+1,i] &= a[j+1,i] + D(a[j,i] + a[j+1,i-1] \\ &- 4a[j+1,i] + a[j+1,i+1] + a[j+2,i]) \end{aligned} \tag{15}$$

with their relative data access pattern depicted in Figure 7 the substitutions (equations (9), (12), (13), (14) and (15))

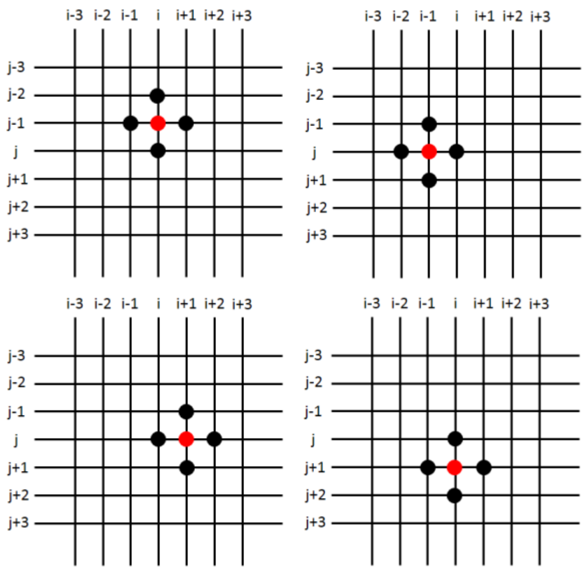

Figure 7: 2D Stencil – read from black and red dot positions, write to red dot position

can be directly applied to equation (11) resulting in

$$
\begin{aligned}
t(a[j,i]) &= c[j,i] \\
c[j,i] &= a[j,i] + D(a[j-1,i] + a[j,i-1] - 4a[j,i] \\
&\quad + a[j,i+1] + a[j+1,i]) + D(a[j-1,i] \\
&\quad + D(a[j-2,i] + a[j-1,i-1] - 4a[j-1,i] \\
&\quad + a[j-1,i+1] + a[j,i]) + a[j,i-1] \\
&\quad + D(a[j-1,i-1] + a[j,i-2] - 4a[j,i-1] \\
&\quad + a[j,i] + a[j+1,i-1]) - 4(a[j,i] \\
&\quad + D(a[j-1,i] + a[j,i-1] - 4a[j,i] \\
&\quad + a[j,i+1] + a[j+1,i])) + a[j,i+1] \\
&\quad + D(a[j-1,i+1] + a[j,i] - 4a[j,i+1] \\
&\quad + a[j,i+2] + a[j+1,i+1]) + a[j+1,i] \\
&\quad + D(a[j,i] + a[j+1,i-1] - 4a[j+1,i] \\
&\quad + a[j+1,i+1] + a[j+2,i]))
\end{aligned} \tag{16}
$$

The resulting equation after the direct substitution of the terms, eliminating the intermediary step $b[]$, the equation (16) can be simplified to the equation (17).

$$
\begin{aligned}
c[j,i] &= D^2(a[j-2,i] + a[j,i-2] + a[j,i+2] \\
&\quad + a[j+2,i]) + (2D^2)(a[j-1,i-1] \\
&\quad + a[j-1,i+1] + a[j+1,i-1] + a[j+1,i+1]) \\
&\quad + (2D - 8D^2)(a[j-1,i] + a[j,i-1] + a[j,i+1] \\
&\quad + a[j+1,i]) + (1 - 8D + 20D^2)(a[j,i])
\end{aligned} \tag{17}
$$

The new data access geometrical pattern of equation (17) is graphically represented in Figure 8.

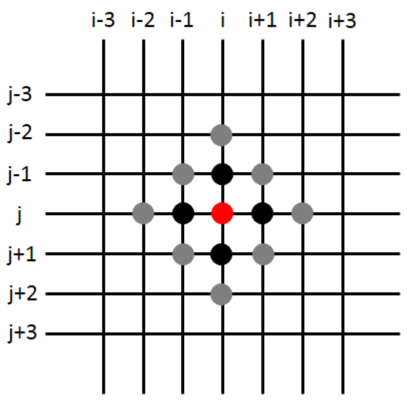

Figure 8: 2D Stencil – Composed Heat Equation Stencil data access pattern

4.1.3 3-D Composed Stencil

The discretization of the equation (1) for a 3-Dimensional space can be expressed as

$$
\begin{aligned}
b[k,j,i] &= a[k,j,i] + D(a[k-1,j,i] \\
&\quad + a[k,j-1,i] + a[k,j,i-1] - 6a[k,j,i] \\
&\quad + a[k,j,i+1] + a[k,j+1,i] + a[k+1,j,i])
\end{aligned} \tag{18}
$$

where D is the thermal diffusivity, the vector $a[]$ holds the input data and the vector $b[]$ stores the result of the application of the stencil over the elements of $a[]$. Figure 9 shows the accessed data for reading (black and red dots in the $a[]$ data array) and data written (red dot in the $b[]$ data array). In this case, similarly to the 1-D and 2-D versions of this problem (Sections 4.1.1 and 4.1.2), it is not possible to read and write directly to the array $a[]$ because of the algorithmic RAW data dependencies between the elements of $a[]$.

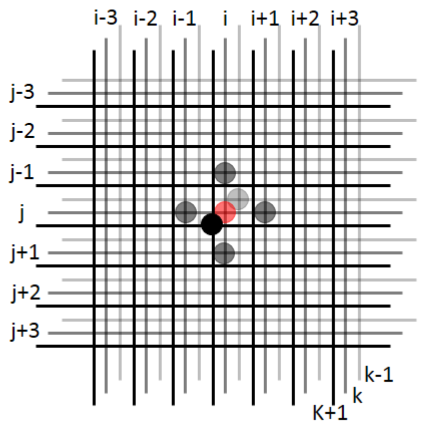

Figure 9: 3D Stencil – read from black and red dot positions, write to red dot position

As in the previous 1-D and 2-D cases, we can define the function $s()$ as the function that applied to $a[]$ returns $b[]$ so that $b[] = s(a[])$, then it can also define t as $s^2()$ so that

$$
c[] = t(a[]) = s^2(a[]) = s(s(a[])) = s(b[]) \tag{19}
$$

where $c[]$ is the array that stores the results of applying twice the original stencil kernel to all the elements of the original data-set $a[]$. As previously seen, to generate the function t and knowing that it is equivalent to $s(b[])$ as seen in equation (19), then we can express it for a specific element of $a[]$ as

$$
\begin{aligned}
t(a[k,j,i]) &= s(b[k,j,i]) = b[k,j,i] + D(b[k-1,j,i] \\
&\quad + b[k,j-1,i] + b[k,j,i-1] \\
&\quad - 6b[k,j,i] + b[k,j,i+1] \\
&\quad + b[k,j+1,i] + b[k+1,j,i])
\end{aligned} \tag{20}
$$

As shown in the previous sections, if the specific elements of $b[]$ that appear in the equation (20) are replaced by their equivalents expressed as just depending on elements of $a[]$ so that all the elements of $b[]$ disappear from the equation we obtain, already simplified the equation(21).

$$\begin{aligned}c[k,j,i] = {} & (2D - 6D^2)(a[k,j-1,i] + a[k,j,i-1] \\ & + a[k,j,i+1] + a[k,j+1,i] + a[k-1,j,i] \\ & + a[k+1,j,i]) + (2D^2)(a[k,j-1,i-1] \\ & + a[k,j-1,i+1] + a[k-1,j-1,i] \\ & + a[k+1,j-1,i] + a[k,j+1,i-1] \\ & + a[k-1,j,i-1] + a[k+1,j,i-1] \\ & + a[k,j+1,i+1] + a[k-1,j,i+1] \\ & + a[k+1,j,i+1] + a[k-1,j+1,i] \\ & + a[k+1,j+1,i]) + (D^2)(a[k,j-2,i] \\ & + a[k,j,i-2] + a[k,j,i+2] + a[k,j+2,i] \\ & + a[k-2,j,i] + a[k+2,j,i]) \\ & + (1 - 12D + 42D^2)a[k,j,i]\end{aligned} \tag{21}$$

In this case the data access geometrical pattern has also grown to include all the necessary *points* from $a[]$ to compute at once an element of $c[]$ without the need of an intermediary grid $b[]$.

4.2 Computational Analysis

In this section we will analyze the number of arithmetic and memory operations needed by each version of the stencil kernels. These figures can be indicative of the benefits or detriments caused by the stencil composition, although to really answer the question on how it affects execution time and energy used, the only possibility is to execute representative application and measure it. The reason is that it is almost impossible to predict detailed behaviors on modern CPUs due to the different optimization strategies that these currently implement, such as complex memory hierarchies (*i.e.* different cache levels with some levels shared across different cores), data pre-fetching, multiple execution pipelines, etc. As stencil computations are very data intensive, to determine the impact from the memory subsystem, two implementations of every formulation were developed, one with single precision and another with double precision floating point data types. Double precision uses twice the memory for each data element, 8 bytes, while single precision uses 4 bytes. The cost of computing the same operation in single or double precision might be similar, or not, depending on the hardware implementation.

TODO: make tables look nicer

4.2.1 1-D Computational Analysis

An equivalent formulation of equation (2) is given in equation (22). This formulation is characterized for offering a better performance, even when it performs and extra arithmetic operation, because the number of memory operations is reduced, which is the main bottleneck as will be seen later in Section 5. The computational cost of these are compared in Table 1.

$$b[i] = (1 - 2D)a[i] + D(a[i-1] + a[i+1]) \tag{22}$$

Table 2 compares the theoretical computational cost of executing twice equation (22), left column, with the execution of the composed stencil as formulated in equation (8), right column, as these are mathematically equivalent.

	Eq. (2)	Eq. 22
ADDs	3	2
MULs	1	2
READs	4	3
WRITEs	1	1

Table 1: Computational analysis of equivalent 1-D stencil formulations

	2x Eq. (22)	Eq. (8)
ADDs	4	4
MULs	4	3
READs	6	5
WRITEs	2	1

Table 2: Computational analysis of applying twice the original 1-D stencil kernel and once the equivalent composed stencil kernel

The composed 1-D stencil reduces some of the arithmetic and memory operations.

4.2.2 2-D Computational Analysis

Similarly to the 1-D version, equation (9) can be formulated as in equation (23) that is equivalent but offers better performance due to the reduction of memory operations.

$$\begin{aligned}b[j,i] = {} & (1 - 4D)a[i,j] + D(a[j-1,i] + a[j,i-1] \\ & + a[j,i+1] + a[j+1,i]\end{aligned} \tag{23}$$

The computational cost of these are compared in Table 3.

	Eq (9)	Eq 23
ADDs	5	4
MULs	1	2
READs	6	5
WRITEs	1	1

Table 3: Computational analysis of equivalent 2-D stencil formulations

Table 4 compares the theoretical computational cost of executing twice equation (23), left column, with the execution of the composed stencil as formulated in equation (17), right column, as these are mathematically equivalent. Arithmetic and memory read operations have slightly increased, but write to memory operations remain lower in the composed stencil.

4.2.3 3-D Computational Analysis

As with the previous versions, equation (18) can be formulated as in equation (24) that is equivalent but offers better performance due to the reduction of memory operations.

	2x Eq (23)	Eq (17)
ADDs	8	12
MULs	4	4
READs	10	13
WRITEs	2	1

Table 4: Computational analysis of applying twice the original 2-D stencil kernel and once the equivalent composed stencil kernel

The computational cost of these are compared in Table 5.

$$\begin{aligned} b[k,j,i] &= (1-6D)a[k,j,i] + D(a[k-1,j,i] \\ &+ a[k,j-1,i] + a[k,j,i-1] + a[k,j,i+1] \\ &+ a[k,j+1,i] + a[k+1,j,i]) \end{aligned} \quad (24)$$

	Eq (18)	Eq 24
ADDs	7	6
MULs	1	2
READs	8	7
WRITEs	1	1

Table 5: Computational analysis of equivalent 3-D stencil formulations

Table 6 compares the theoretical computational cost of executing twice equation (24), left column, with the execution of the composed stencil as formulated in equation (21), right column, as these are mathematically equivalent.

	2x Eq (24)	Eq (21)
ADDs	12	24
MULs	4	4
READs	14	25
WRITEs	2	1

Table 6: Computational analysis of applying twice the original 3-D stencil kernel and once the equivalent composed stencil kernel

In this case it can be observed that most of the parameters got sensibly worse. ADD and memory READ operations are almost doubled, and only the WRITE to memory operation remains lower. For this reasons we might be tempted to think that, obviously, the execution time has to be worse. But as we will see in Chapter 5 this is not always true, as it heavily depends on the hardware platform on which the code is executing.

5. EMPIRICAL PERFORMANCE ANALYSIS

The composed stencils for 1-, 2- and 3-dimensions (equations (8), (17) and (21) respectively) have been directly benchmarked against theire standard versions and formulated in equations (22), (23) and (24). The equations were directly coded in C, *e.g.* equation (22) was implemented as:

```
#define D 0.05f //Some value < 0.5
...
b[i]= (1.0f-2.0f*D)*a[i]+D*(a[i-1]+a[i+1]);
```

when `a[]` and `b[]` were arrays storing single precission floating point elements and

```
#define D 0.05 //Some value < 0.5
...
b[i]= (1.0-2.0*D)*a[i]+D*(a[i-1]+a[i+1]);
```

when `a[]` and `b[]` were storing double precision elements. All tests were executed on different hardware, ranging from low-power embedded Cortex-A7 cores to bigger and *power-hungry* Cortex-A15 cores to modern HPC Intel *Haswell* cores, in single and multiple threads. These platforms are detailed in the next Section 5.1.

The test applications applied the stencil a determined number of iterations N ($N/2$ for the composed stencil, so that the final computation is equivalent) and used pointer-swapping double buffering to store the intermediate results between iterations. Table 7 describes the size of the problem, the memory used –in single precision (SP) and double precision (DP)–, and the number of iterations the stencils were applied.

	Odroid-XU3	Hornet
1-Dimension		
Grid Size	96000000	2097152000
Bytes (SP/DP)	(1.43/0.72) GiB	(16.0/32.0) GiB
Iterations (Si/Co)	(200/100)	(30/15)
2-Dimensions		
Grid Size	9600^2	25600^2
Bytes (SP/DP)	(0.68/1.37) GiB	(5.0/10.0) GiB
Iterations (Si/Co)	(40/20)	(60/30)
3-Dimensions		
Grid Size	400^3	1280^3
Bytes (SP/DP)	(0.5/1.0) GiB	(16.0/32.0) GiB
Iterations (Si/Co)	(40/20)	(20/10)

Table 7: Test size description for each platform

5.1 Hardware

The **Odroid-XU3** is a High Performance Embedded Board featuring 8 CPU cores in two 4-core cluster. The first cluster is formed by ARM Cortex A-7 cores (little cores – configuration A, Table 8) and the second cluster is formed by ARM Cortex A-15 cores (big cores – configuration B, Table 9). The board contains 4 current and voltage sensors to measure the power consumption of the Cortex A-15 cores (big core cluster), the Cortex A-7 cores (little core cluster), GPU and DRAM individually. The underlying Operating System installed on this board is Ubuntu 14.04 LTS GNU/Linux.

The **Cray XC40** is a massively parallel multiprocessor HPC system manufactured by Cray. It is based on x86-64 Intel Haswell Xeon processors and the Cray Aries network. Each computing node has the mechanism to measure and expose trough the file system the total power used from the start up till the moment of the request but unfortunately it does

CPU	Samsung Exynos 5422 – Configuration A
Cores	4x Cortex A-7 @ 1.4 Ghz
Cache	0.5 MiB L2 (shared)
RAM	2GiB LPDDR3 @ 933MHz (14.9 GiB/s)

Table 8: Odroid-XU3 – Configuration A – Hardware description

CPU	Samsung Exynos 5422 – Configuration B
Cores	4x Cortex A-15 @ 2.0 Ghz
Cache	2 MiB L2 (shared)
RAM	2GiB LPDDR3 @ 933MHz (14.9 GiB/s)

Table 9: Odroid-XU3 – Configuration B – Hardware description

not report how the energy was used. Although the test machine (the Hornet system at HLRS) consists of 3944 compute nodes, only one was used for the tests as using more nodes would not contribute to the aim of this paper. The system is running a version of GNU/Linux customized by Cray and designed for this particular cluster. The tests in this platform were executed in single and double precision and using one or 24 cores. The Hardware of a Hornet node are detailed in Table 10.

CPU	2x Intel Xeon E5-2680v3
Cores	24x Haswell Cores @ 2.5 Ghz
Cache L2	24x 0.25 MiB L2 (for each core + L1)
Cache L3	2x 30 MiB LLC (one per CPU)
RAM	128GiB DDR4 @ 2100MHz (136 GiB/s)

Table 10: Hornet Node – Hardware description

5.2 Methodology

The compiler GCC 5.1 was used to generate the binaries for the Odroid-XU3 board with the options `-O3 -ffast-math`[2] as this combination of compilation flags offered the same or best performance as other flags, even better than adding `-mtune=native`, `-mtune=cortex-a15` or `-mtune=cortex-a7`. Other compilers such as GCC 4.9.2 and LLVM 3.6 were also tested but the performance of their binaries was always worse than GCC 5.1. For both different types of cores in this CPU, the tests were executed in single and double precision and using one or 4 cores (*i.e.* using all the cores of each cluster). This board integrates accurate current and voltage sensors on the PCB that allow to measure the energy consumption of the big cores (Cortex-A15 cluster), little cores (Cortex-A7 cluster), GPU and DRAM individually. Unfortunately there is no sensor for the whole system as some devices, such as the network NIC or the flash drive, are not reported. But the power needed by these devices is small compared to the CPU in active usage.

[2]`-ffast-math` allows the compiler to re-order floating point operations, which numerically might lead to not exactly the same results due to the representation limits and rounding in IEEE 754 technical standard [6].

For the Cray XC40 the Cray Compiling Environment (CCE 8.3.14) was used as this specialized compiler generates binaries with better performance than other generic compilers available in the service nodes of this system. The tests in this platform were also executed in single and double precision and using one or 24 cores (*i.e.* the total amount of cores in a node). Each node contains a sensor that measures the energy consumed by the whole node, without specifying how the energy was used, for this reason we will not be able to analyze how the composed stencil behaves from the energy point of view of the CPU or memory individually, but just the node as a whole.

5.3 Results

In the charts shown in this section (from Figures 10 to 18) the pairs of bars show the execution time and energy consumed to execute the same problem where the left blue bar represents the time in milliseconds and the right orange bar represents the energy used in joules; lower values of each bar mean better results.

The labels of the charts are composed by 3 groups of 2 characters. The first group can be `Si` for "simple" stencil (the conventional one) or `Co` for "composed" stencil, the one generated with the approach described in this paper. The second group can be `SP` for "Single Precision" or `DP` "Double Precision" floating point data types. The latest group can be `1T`, meaning that the test was running using only one thread (core), or `4T` running on 4 threads (on the Odroid-XU3 board) or `24T` running on 24 threads (on the Hornet HPC system).

TODO: Improve charts
a) not blured
b) improve colors for B/W printing

TODO: Make Energy Tables nicer

1-D Composed Stencil Results

From the computational analysis from Section 4.2 we can expect certain benefit of the composed stencil. This is confirmed in the empirical results where, in many cases, both execution time and energy used is halved when dual precision data types and multi-threading is used for all the kinds of hardware tested.

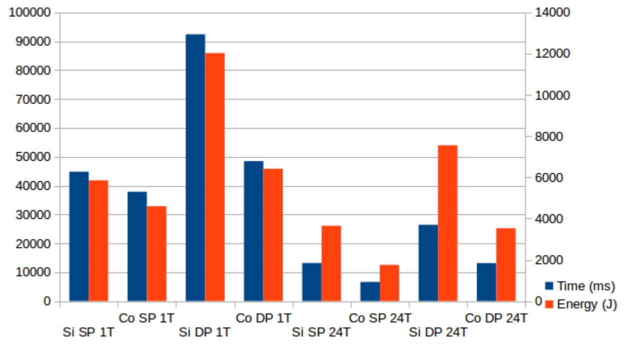

Figure 10: Hornet 1-D Stencil benchmark

Table 11 shows that the not-used elements of the CPU (the

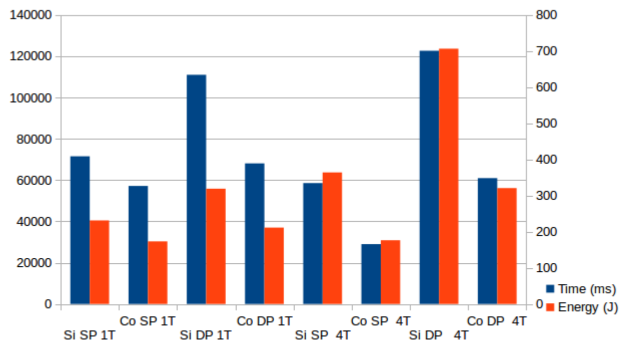

Figure 11: Cortex-A15 1-D Stencil benchmark

Cortex-A7 cluster and the GPU) and active elements (the Cortex-A15 cluster and the memory) used energy proportional to the execution time.

	Total	A15	A7	GPU	Memory
Si SP 1T	231.42	206.87	1.97	8.61	13.96
Co SP 1T	173.29	157.26	1.41	6.21	8.39
Si DP 1T	319.10	281.42	2.73	11.14	23.77
Co DP 1T	211.43	188.76	1.82	7.27	13.56
Si SP 4T	364.17	336.11	2.03	8.15	17.85
Co SP 4T	176.51	163.07	1.05	3.94	8.44
Si DP 4T	706.63	646.77	4.33	17.73	37.77
Co DP 4T	320.78	292.09	2.10	8.35	18.22

Table 11: 1-D Stencil – Energy (J) decomposition using the Cortex-A15 cluster

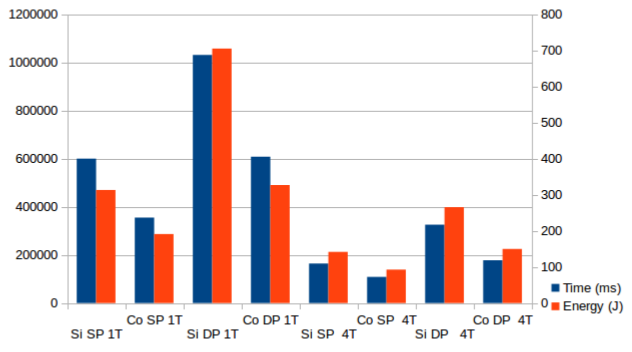

Figure 12: Cortex-A7 1-Dimension Stencil

As with the Cortex-A15 cluster, Table 12 shows that the not-used elements of the CPU (the Cortex-A15 cluster and the GPU) and active elements (the Cortex-A7 cluster and the memory) used energy proportional to the execution time.

2-D Composed Stencil Results

More interesting results come fro the 2-D composed stencil as the computational analysis shows that the computational and memory access cost is worse than in the conventional stencil. But the benchmarks show that although the improvement in both execution time and energy is not as good as in the 1-D case, benefits are significant (except for Hornet

	Total	A15	A7	GPU	Memory
Si SP 1T	313.34	57.25	144.92	57.28	53.87
Co SP 1T	191.23	36.54	88.54	36.54	29.60
Si DP 1T	705.46	263.87	245.43	105.50	90.65
Co DP 1T	327.14	64.79	145.07	63.34	53.92
Si SP 4T	141.80	16.98	76.11	15.99	32.71
Co SP 4T	92.75	11.81	51.35	11.24	18.33
Si DP 4T	265.61	38.12	134.05	35.53	57.89
Co DP 4T	149.85	21.33	76.16	20.11	32.23

Table 12: 1-D Stencil – Energy (J) decomposition using the Cortex-A7 cluster

using single precision and one thread) specially when using multi-threading and double precision.

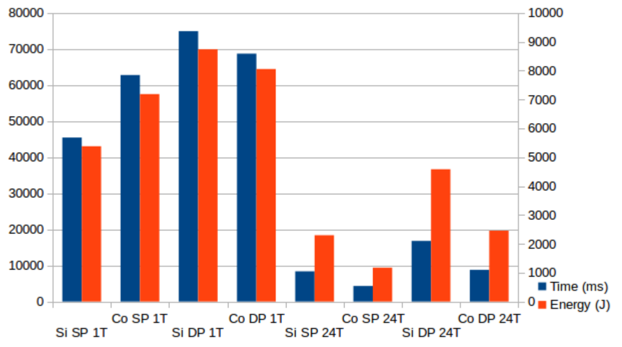

Figure 13: Hornet 2-Dimension Stencil

Figure 13 depicts the results on Hornet. Only in the case were single precision and one thread were used the performance and energy were worse (around 33%), and all the other cases, both performance and energy used, improved (merely an 8% for double precision using one thread, and more than a 45% when multi-threading was used for single and double precision).

Figures 14 and 15 show that for both ARM Cortex types there was a benefit in all cases, but these were less than for Hornet, ranging around 12 to 35% less execution time and 15 to 25% reduction in energy used.

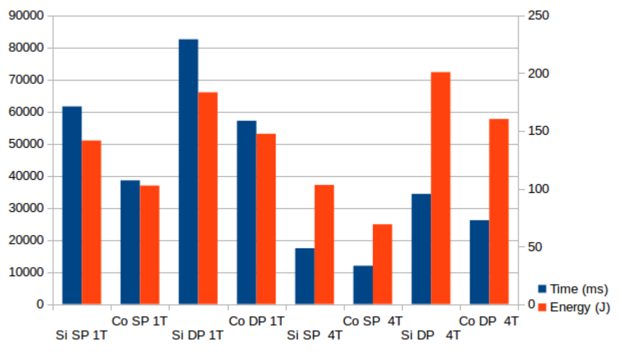

Figure 14: Cortex-A15 2-Dimension Stencil

The energy decomposition for the Cortex-A15 test, Table 13, shows an increment of the power (energy per unit of time) used by the active cores (5 to 20%), while the power needed by the memory is reduced (up to a 15%). The total power needed is bigger but due to the lower running time the total energy used is reduced. For this case the Cortex-A7 test shows very similar results (*c.f.* Table 13).

	Total	A15	A7	GPU	Memory
Si SP 1T	141.64	128.47	1.30	6.42	5.43
Co SP 1T	102.64	94.77	0.77	3.81	3.27
Si DP 1T	183.42	163.66	1.83	8.16	9.75
Co DP 1T	147.52	134.42	1.19	5.58	6.31
Si SP 4T	103.18	96.00	0.64	2.43	4.10
Co SP 4T	69.08	63.33	1.49	1.90	2.34
Si DP 4T	200.94	184.02	1.36	5.48	10.07
Co DP 4T	160.32	147.48	1.40	4.24	7.19

Table 13: 2-D Stencil – Energy (J) decomposition using the Cortex-A15 cluster

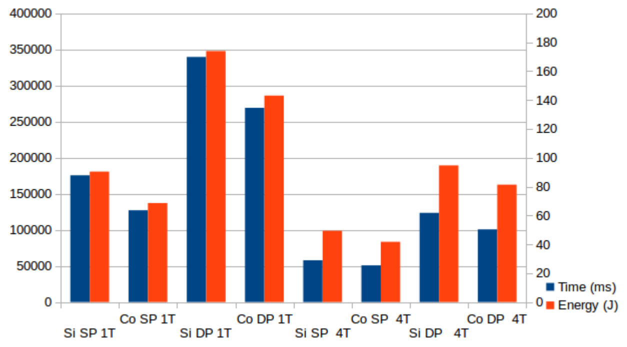

Figure 15: Cortex-A7 2-Dimension Stencil

	Total	A15	A7	GPU	Memory
Si SP 1T	90.38	16.73	43.13	16.73	13.77
Co SP 1T	68.61	13.71	32.94	12.61	9.33
Si DP 1T	173.95	35.70	78.79	34.69	24.76
Co DP 1T	143.07	28.39	64.69	28.19	21.78
Si SP 4T	49.38	6.41	27.16	5.97	9.83
Co SP 4T	41.73	5.46	23.26	5.14	7.85
Si DP 4T	94.70	14.10	47.72	12.98	19.89
Co DP 4T	81.37	12.88	40.16	10.86	17.46

Table 14: 2-D Stencil – Energy (J) decomposition using the Cortex-A7 cluster

3-D Composed Stencil Results

The benchmarks performed for the 3-D composed stencil and depicted in Figure 16 show that the performance and energy used by the Hornet node is improved (up to almost 30% less execution time and 20% less energy if multi-threading is used) but that both performance and energy may sensible deteriorate in single-threaded scenarios.

Similarly, *c.f.* Figure 17, the Cortex-A15 deteriorates its performance and energy consumption in case of composed 3-D stencil in single threaded tests. In multi-threading and

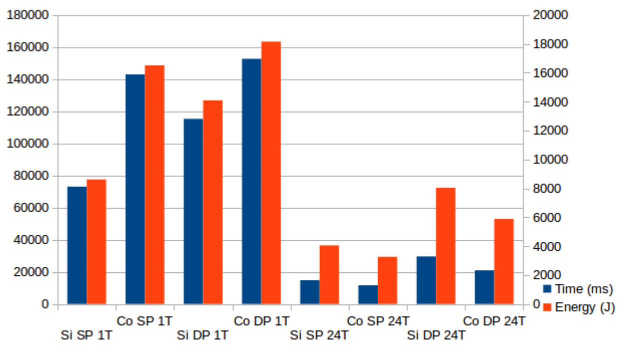

Figure 16: Hornet 3-Dimension Stencil

single precision the performance is reduced (execution time increases a 32%) but energy consumption is also reduce almost a 18%, *i.e.* it takes more time to compute but it uses less energy in total. For the multi-threading and double precision case performance improves a 10% and energy consumption does not vary (it is reduced less than a 1%).

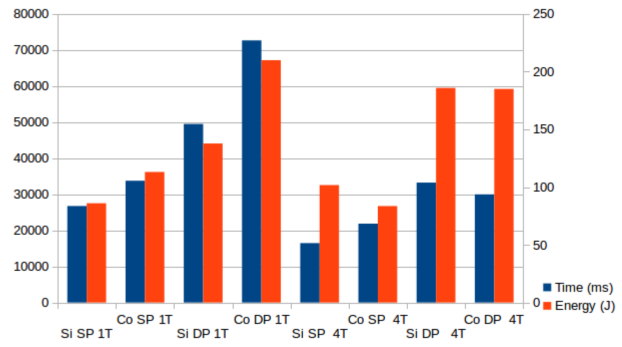

Figure 17: Cortex-A15 3-Dimension Stencil

From Table 15 can be observed that for both single-threaded cases the memory power was less and that the active cores needed more power, having a longer run time the total energy used significantly increased. In the multi-threaded single precision case the power used by the active cores and the memory was greatly reduced and compensated the higher total running time leading to a total energy reduction compared to the single-thread case. In the multi-threaded double precision case the power used by the memory was decreased but the cores needed more power making the system use more power, the total running time was less so the total energy used was slightly less.

Figure 18 clearly shows that for the Cortex-A7 cores the composed 3-D stencil reduces performance and increases energy consumption.

Table 16 clarifies that even if the power needed by the memory is decreased the active cores increase their power needs, making the system more energy hungry. As the computation time also increases for all the tests, the energy used in every test is also increased.

6. CONCLUSIONS

	Total	A15	A7	GPU	Memory
Si SP 1T	86.03	77.42	0.71	3.24	4.65405
Co SP 1T	113.10	103.16	0.84	3.94	5.14199
Si DP 1T	137.75	121.42	1.15	5.21	9.9518
Co DP 1T	209.91	189.65	1.58	7.36	11.3083
Si SP 4T	101.78	93.20	0.76	2.34	5.47308
Co SP 4T	83.60	74.51	1.41	3.16	4.51311
Si DP 4T	185.90	168.55	1.24	4.74	11.3577
Co DP 4T	185.02	170.13	1.34	4.76	8.78189

Table 15: 3-D Stencil – Energy (J) decomposition using the Cortex-A15 cluster

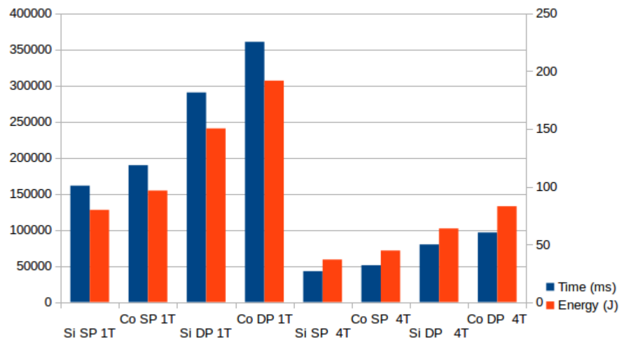

Figure 18: Cortex-A7 3-Dimension Stencil

Stencil kernels are relevant for science, engineering and image processing, and often take a high percentage of the computation in the applications of these fields. In this work we have exposed a way to compose iterative stencils or combine different ones into a single kernel, reducing the total number of iterations over the original data-grid. With this approach we have demonstrated that it is possible to achieve reduction of the execution time and energy used of over a 40% for 2-dimensional stencils and 30% for 3-dimensional stencils in HPC machines. For systems with a more limited memory subsystem, such as many ARM-based systems, the improvements for 2-dimensional stencils are also noticeable but might not be for 3-dimensional stencils. The best results were achieved from 1-dimensional stencils, but these have very little applicability in real-life problems. We have also seen that the combination of stencil kernels is a mechanical process, thus it should be possible to automatize it in future.

In will be interesting to analyze how the combination of stencils affects hardware accelerator, in particular FPGAs. We can hypothesize that this approach can benefit them as they can take advantage from the reduced amount of times that the data has to be streamed to them.

7. REFERENCES

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[2] M. S. Christian Grossmann, Hans-G. Roos. *Numerical Treatment of Partial Differential Equations*. Springer Science and Business Media, 2007.

[3] F. Dietmar. *Grid-Computing: Eine Basistechnologie für Computational Science*. Springer, 2010.

[4] G. R. et al. Compiling stencils in high performance fortran. In *Proceedings of SC'97*. High Performance Networking and Computing, 1997.

[5] M. G. Fan Chan, Jiannong Cao. *High-Performance Computing – Paradigm and Infrastructure*. Wiley Interscience, 2005.

[6] I. O. for Standardization (ISO), the Institute of Electrical, and E. E. (IEEE). *IEEE Standard for Floating-Point Arithmetic (IEEE 754) – ISO/IEC/IEEE 60559:2011*. International Organization for Standardization (ISO) and the Institute of Electrical and Electronics Engineers (IEEE), 2011.

[7] T. Gysi, T. Grosser, and T. Hoefler. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In *Proceedings of the 29th International Conference on Supercomputing (ICS'15)*, pages 177–186. ACM, Jun. 2015.

[8] P. N. J. Crank. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Proceedings of the Cambridge Philosophical Society*, 43(1):50–67, 1947.

[9] L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 1(5):73–132, 1993.

[10] C. Leopold. Tight bounds on capacity misses for 3d stencil codes. *Computational Science – ICCS 2002*, 1(2329):843–852, April 2002.

[11] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. *2014 IEEE 38th Annual Computer Software and Applications Conference*, 1:579–586, 2009.

	Total	A15	A7	GPU	Memory
Si SP 1T	79.83	14.48	37.50	14.16	13.67
Co SP 1T	96.62	17.85	46.98	18.00	13.78
Si DP 1T	150.43	30.24	67.20	29.17	23.82
Co DP 1T	191.82	37.32	89.56	37.51	27.42
Si SP 4T	36.83	5.12	19.84	4.45	7.39
Co SP 4T	44.66	5.40	27.00	5.10	7.14
Si DP 4T	63.73	8.86	32.57	8.25	14.03
Co DP 4T	82.99	10.88	46.92	10.42	14.76

Table 16: 3-D Stencil – Energy (J) decomposition using the Cortex-A7 cluster