Computer Science and Information Systems 00(0):0000–0000 https://doi.org/10.2298/CSIS123456789X

Energy-efficiency of software and hardware algorithms* [*] [⋆]

Maja H. Kirkeby, Thomas Krabben, Mathias Larsen,
Maria B. Mikkelsen, Mads Rosendahl, Martin Sundman
Department of People and Technology
Roskilde University
Roskilde, Denmark
majaht@ruc.dk, krabben@ruc.dk, mamaar@ruc.dk,
mariabm@ruc.dk, madsr@ruc.dk, sundman@ruc.dk
and
Tjark Petersen, Martin Schoeberl
DTU Compute
Technical University of Denmark
Lyngby, Denmark
s186083@student.dtu.dk, masca@dtu.dk

Abstract. In this article, we compare the energy efficiency of hardware and software implementations of Heapsort and Dijkstra's algorithm for route finding. The software implementations are written in C for Raspberry Pi, and the hardware implementations are crafted in Chisel for an FPGA. Our objective is to examine how we can fairly compare energy efficiency between hardware and software. This study seeks to identify circumstances where time and energy efficiency diverge, providing preliminary insights that inform hardware selection. Our findings serve as a step towards understanding the complex trade-offs in algorithm performance across different computational platforms.

Keywords: energy efficiency, performance, FPGA, CPU, algorithms

1. Introduction

Improving software's time and energy efficiency is essential [1, 5]. Implementing the algorithms in Field-Programmable Gate Arrays (FPGAs) demonstrates substantial performance gains in highly parallelizable tasks, e.g., [16, 26]. However, only a few have considered the energy efficiency of hardware implementations [12, 15]. Previous studies on improving algorithms using FPGAs have employed FPGAs in different ways. One approach requiring highly specialized knowledge is implementing the algorithm in hardware-specifying languages such as Verilog or VHDL [24,26]. Alternative approaches,

⋆ This work is supported by the Innovation Fund Denmark for the project DIREC (9142-00001B), by CERCIRAS Cost Action CA19135, and by the Independent Research Fund Denmark Project no. 2102-00281B.

oriented toward software developers, use FPGAs as accelerators for, e.g., C/C++ programs [12] or, as in this study, where use the high-level programming language Chisel [2, 23]. This research extends the initial studies published at the 2022 Workshop on Resource Awareness of Systems and Society [16]. The initial study indicated an exciting upper bound for performance improvement when comparing a highly parallelizable program, Conway's Game of Life [3], to software implementations. While the upper bound is interesting, it does not provide any insights into the gain from converting ordinary software implementations into hardware implementations for ordinary programs. In this study, we will focus on this gap.

How do time and energy consumption compare for hardware and software implementations of ordinary software algorithms?

In our choice of algorithms, it is our aim that it should not provide any particular advantage for either hardware or software versions. It is possible to find highly parallelizable algorithms where the hardware implementation will have an obvious advantage. In our experiments, we use Heapsort and Dijkstra's algorithm to find the shortest path in a graph. Both are widely used and have well-known and studied implementations. Both algorithms are not easy to parallelize [11] [8] but can still contain a fair amount of parallelism at the local level.

The main contributions of this article are:

1. *Comparative Analysis:* The paper compares the energy efficiency between software and hardware implementations of two well-known algorithms: Heapsort and Dijkstra's algorithm. This comparison is new in that it focuses on both energy consumption and performance across software and hardware implementations and, thus, provides the first insights into the energy and performance trade-offs
2. *Methodological Innovation:* It introduces a methodological framework that ensures fair comparisons between software and hardware implementations. This includes a thorough discussion on the choice of measurement techniques.

In the following, we provide the basis for a fair comparison (Section 2) and, in Section 3, we introduce the specifications and implementations of the algorithms. Section 4 describes our experimental setup and Section 5 describes our results. Afterwards, we discuss the closest related literature (Section 6), and contextualize our results. Section 9 concludes the paper and provides a summary of future work.

2. A Fair Comparison

In this section, we discuss a methodological framework to ensure that comparisons between software and hardware implementations are both fair and insightful. By explaining the choice of hardware and measurement protocols, this section highlights the practical aspects of the comparison and lays the foundation before we dive into the detailed descriptions of specific implementations

2.1. Choice of Measurement

Previous work on energy consumption of software implementations has employed energy estimations using Intel's Running Average Power Limit (RAPL) [9], as it has been reported as having negligible overhead and providing precise results [7, 21]. e.g., [20]. However, while RAPL is precise and highly correlated (a value of 0.99) [14]) with the actual power dissipation, it does not provide accurate results, i.e., the RAPL measurements are not close to the true energy consumption. Thus, instead, we employ external measurements that provide each device's ground truth energy consumption. This choice also has drawbacks. For instance, it introduces more noise since it measures the energy consumption of the entire device compared to only the CPU. It also introduces an imprecision in the synchronization between the time in the measuring unit obtaining the energy consumption and the time in the measured device that executes the algorithm. While both examples introduce more noise in the measurements, this methodology will support a fair comparison across platforms.

2.2. Choice of Hardware

One factor that influences the energy consumption of both hardware and software implementations is the choice of hardware. Since the hardware is very different in the two cases, we will use the same requirement for choosing the hardware, namely, to use cheap and available hardware.

There are many cheap and available computers for software implementations, e.g., Orange Pi, Asus Tinker, Nvidia Jetson Nano, or Raspberry Pi. However, in this study, we have chosen a standard Raspberry Pi 4 computer Model B with 4GB RAM and a 1.5 GHz 64-bit quad-core ARM Cortex-A72 processor running the standard Raspberry Pi OS, a Linux version.

There are two types of FPGA units: pure FPGA boards and system-on-chip FPGA boards. Choosing the pure FPGA board will allow us to measure the exact energy consumption of the FPGA unit alone without interference from other processors, which would be the case with System on Chip FPGA boards.

For the hardware implementations, we have chosen a Digilent FPGA board Cmod A7 (version Cmod A7-35T) with an XC7A35T-1CPG236C FPGA unit, an MSPS On-chip ADC, 20800 Look-up Tables (LUTs), 41600 Flip-Flops, 225 KB Block RAM and 5 Clock Management Tiles.

2.3. Choice of Implementations

In addition, previous studies, e.g., [5, 6, 20] highlight that whole-systems energy consumption provides insights into the varying energy consumption across different implementations, illustrating how choices in software development impact overall energy use. Thus, for instance the choice of programming language [20], implementation style [6], and compiler flags [22] influence the energy consumption and execution time of the programs. For the hardware implementation, energy factors typically include the number of logic elements and routing resources, see, e.g., [25].

We aim to compare the typical behavior of the implemented algorithm on the given form of hardware. For the software solutions, we base it on standard C-implementations, e.g., following the descriptions in [4], using standard settings of optimizations in the GNU CC compiler.

The Hardware solutions are hand-written versions in Chisel with common approaches to optimize the design for improved performance. How we have realized the algorithms in hardware is discussed in the next section

While the structures of the implementations will differ, we will align their work so that they follow the same requirements. The implementations will:

1. carry out Dijkstra's algorithm and Heapsort, respectively.
2. avoids external communication by including the algorithm inputs in the implementation instead of reading the input from external files.
3. the program will return the smallest subset of the result to ensure computation of the results while reducing the read/write accesses.

3. Algorithms

This section provides detailed descriptions of Heapsort and Dijkstra's implementations. With a focus on implementation specifics, the next Section 4 allows us to finalize the experimental design before providing the results in Section 5.

3.1. Heapsort: Software

The implementation of Heapsort uses a standard implementation from the Rosetta repository[1], see Figure 1 (page 5). It uses a max-heap data structure, storing values in a balanced tree where a node is bigger than its children. The tree is represented as an array. In a binary tree, the children at array index i can be found at array index $i * 2 + 1$ and $i * 2 + 2$. The implementation uses a k-heap with slightly better complexity measures since the tree will have a smaller depth for the same number of store values.

The k-heap structure is initially established by moving values down the tree structure if they are smaller than their children. In the second phase, the biggest value is repeatedly moved to the back of the array, and the heap structure is reestablished.

3.2. Heapsort: Hardware

The hardware solution of Heapsort uses higher order k-max-heaps to increase parallelism. In the k-heap, $k + 1$ elements must be compared in each step while re-establishing the heap order. In hardware, this comparison can be done in parallel, thus theoretically allowing for the heap order to be established in $\log_k(n)$ clock cycles. Practically, fetching all required values from memory, finding the largest of them, and swapping the parent and the largest child if the heap order is violated has to be spread over multiple clock cycles to allow the circuit to be operated at high clock frequencies. An architectural diagram of the heap module is shown in Figure 2 (page 6). The circuit can not easily be pipelined since deciding which child should be the next parent while traversing the heap downwards is always delayed, resulting in a pipelined implementation having to stall most of the time.

[1] `http://www.rosettacode.org/`

```
// heapsort start
int max (int *a, int n, int parent) {
  int largest = parent;
  for(int child = (K*parent)+1; child < (K*parent)+K+1; child++)
    if(child < n && a[child] > a[largest])
      largest = child;
  return largest;
}
void downheap (int *a, int n, int i) {
  while (1) {
    int j = max(a, n, i);
    if (j == i) break;
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
    i = j;
  }
}
void heapsort (int *a, int n) {
  int i;
  for (i = (n - 2) / K; i >= 0; i--)
    downheap(a, n, i);
  for (i = 0; i < n; i++) {
    int t = a[n - i - 1];
    a[n - i - 1] = a[0];
    a[0] = t;
    downheap(a, n - i - 1, 0);
  }
}
// heapsort end
```

Fig. 1. The software implementation of Heapsort in C, where `a` is the input array to be sorted and `n` is its size.

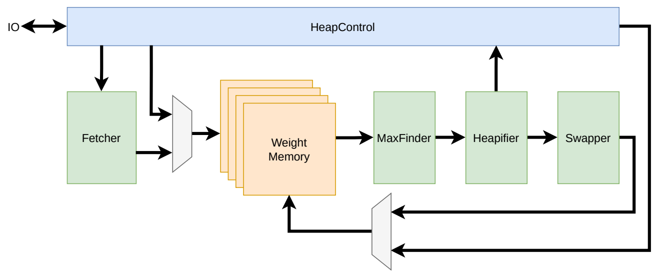

Fig. 2. The architecture of the hardware implementation for Heapsort.

Nonetheless, parallelism can be exploited in multiple instances, giving the hardware implementation an edge over a software implementation in clock cycles per iteration. All k children of a given node can be fetched simultaneously using wide memories, which store k concatenated values per address. Thus, only two load operations are needed per iteration. The maximum between the parent and all k children nodes can be found using a tree of blocks, where the maximum of two values is selected. The total delay of this circuit is $\log_2(k)$ times the delay of a single block. For larger values of k, the tree has to be divided into multiple levels separated by registers to allow for operation at high clock frequencies. The write operations to the memory associated with a swap can be overlapped with fetching the next parent or children, depending on the direction of the traversal.

The described heap module for sorting is paired with a small circuit that inserts values from memory into the heap and then extracts the ordered sequence from the heap by continuously removing the root until the heap is empty.

3.3. Dijkstra: Software

The implementation of Dijkstra's algorithm represents the graph as an adjacency list, where all edges from a vertex are grouped together. The algorithm computes the shortest route from a given vertex (here, the vertex at index 0) to all other vertices. For each vertex, it will find the previous vertex in the shortest route from the start to that vertex and the weight of that route. The actual route can be found by following the previous vertex indices through the graph until the start vertex is reached. The implementation can be seen in Figure 3.

3.4. Dijkstra: Hardware

A priority queue-based system allows for an easy determination of the next node to visit. Using the hardware heap of the heap sort experiment, a hardware priority queue could be constructed that accepts a new value worst case every $\log_k(n)$ clock cycles, where n is the number of nodes in the graph and k is the degree of parallelism in the heap. This results in $n(n-1)d\log_k(n)$ clock cycles to execute Dijkstra's algorithm, where d is the density of the graph with $d = 1$ representing a fully connected graph. This solution is not easily parallelizable since it would require a priority queue that can insert multiple values simultaneously.

```
#include <stdio.h>
// the graph is stored as an adjacency list
// where edges are grouped by start vertex
// input to Dijkstra's algorithm
int n;
#define n 6 // m: number of vertices
#define m 9 // m: number of edges
// start vertex of edge, grouped by vertex
int node1[]={ 0, 0, 0, 1, 1, 2, 2, 3, 4};
// end vertex of edge
int node2[]={1, 2, 5, 2, 3, 3, 5, 4, 5};
// weight of edge
int dist[]={7, 9, 14, 10, 15, 11, 2, 6, 9};
// vertex to edge index link
int edge[]={0, 3, 5, 7, 8, 0};
// Data structures for the algorithm
int done[n], prev[n], wght[n];
const int MAX=1000000;
void main(){
  int start = 0;
  // dijkstra start
  for(int i=0;i<n;i++){
    done[i]=0; prev[i]=-1; wght[i]=MAX;}
  wght[start]=0;
  int cur=-1, w = 0;
  for(int k=0;k<n;k++) {
    cur = -1; w = MAX;
    for (int i = 0; i < n; i++) {
      if (done[i]==0 && wght[i] < w) {
          cur = i;
          w = wght[i];}
    }
    if(cur<0)break;
    int j = edge[cur];
    while (j < m) {
      int n1 = node1[j];
      int n2 = node2[j];
      int d = dist[j];
      if (n1 != cur) break;
      int d2 = wght[n2], d3 = w + d;
      if (d2 > d3) {
        prev[n2] = cur;
        wght[n2] = d3;}
      j++;
    }
    done[cur] = 1;
  }
  cur = n-1;
}  // dijkstra end
```

Fig. 3. Dijkstra's shortest path algorithm; it finds the path from vertex 0.

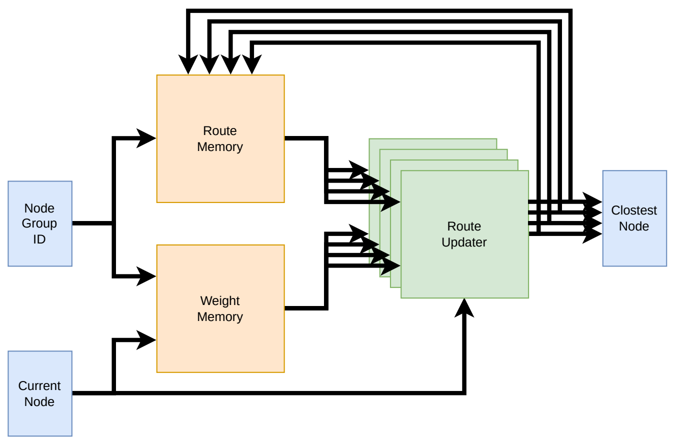

Fig. 4. The architecture of the hardware implementation of Dijkstra's algorithm.

An alternative implementation uses a linear search through all nodes to determine the next node to visit. This proves to be an advantage since the route updates and search for the next node to visit can be conducted in parallel in hardware. The length of an alternative path through the currently visited node is calculated for each node. If the alternative path is shorter, it is written to the route table instead of the previously known distance from the start. At the same time, a state element holding the closest unvisited node yet to be encountered is updated if this node is closer to the start. After all nodes have been visited, the state element holds the next node to visit. This results in $n(n-1)$ clock cycles to execute Dijkstra's algorithm. This solution can easily be parallelized by working on multiple path updates simultaneously. This requires multiple read ports on the weight memory and multiple read and write ports to the route table. Furthermore, not only one route has to be compared to the closest unvisited not yet to be encountered but multiple at the same time.

The second solution is chosen over the first since it is easily parallelizable while requiring significantly fewer hardware resources. This translates into lower power dissipation and only worse performance on graphs of density below $1/log_k(n)$. Considering the total energy of the execution of Dijkstra's algorithm, the density at which both solutions perform equally well is offset even more in favor of the second solution since its lower power consumption compensates for the longer run time.

An architectural diagram of the hardware implementation of Dijkstra's algorithm is shown in Fig. 4. A group of k nodes reads their preliminary shortest path, their visited status from the route memory, and their weight to the currently visited node from the weight memory. For each node in the group, the distance of a new route through the currently visited node is calculated and compared to the old shortest path. The shorter of the two routes is selected and written to the route memory. Furthermore, the state element holding the closest unvisited node to the start is updated if the distances sent to the route memory are shorter. In the hardware implementation, the design is pipelined with the two memory modules separating the circuit into two stages.

To support single-cycle access to the weights without using $O(n2)$ memory, a buffering solution that exploits the known direction of traversal of the weights is employed. This solution stores weights as packed, unaligned adjacency lists, thus only requiring $O(e)$ memory, where e is the number of directed edges in the graph.

4. Experimental Setup

The hardware implementations were executed on a Digilent FPGA board Cmod A7 (version Cmod A7-35T), and the software implementations were compiled with gcc (Raspbian 8.3.0-6+rpi1) 8.3.0 with flag `-O2` and executed on a Raspberry Pi 4 computer Model B 4GB RAM. There is one program per input array, i.e., one file per software and hardware implementation and input array.

4.1. Input spaces

In Heapsort, the runtime and, therefore, perhaps also the energy consumption depend on the number of input elements and their order. A previous study [15] showed that while the order matters for runtime, the comparability arises from using the same order in all experiments. In this study, we (1) always use the same order, namely ordered input, which causes the highest number of comparisons, and (2) vary the number of elements: 4096, 6144, 8192, 10240, 12288, 14336, and 16384.

For Dijkstra's shortest path algorithm, the execution time depends on the type of graph, i.e., sparse or dense and directed or undirected, and the start node for which the path is calculated. In the sorting algorithm, the focus was on growth, and for the shortest path algorithm, we varied only the form and the starting node. The implementation will calculate routes switching between all the possible start vertices.

4.2. Pre-study: Configuration of Heapsort

An exploratory study [15] found that the optimal degree of parallelism in hardware programs differs when considering energy and time. The kind of parallelism in hardware algorithms and software algorithms differs, as described in Section 3, and, thus, the optimal degree of parallelism differs for software and hardware implementations and differs for each algorithm. A fair comparison allows the optimal degree of parallelism, and therefore, we have run a pre-study to find the optimal degree of parallelism. For both hardware and software implementations, we have optimized for energy. Both Dijkstra and Heapsort can have different degrees of parallelism in the hardware implementation, while for the software implementation, this is only true for Heapsort.

Hardware The test-run graph obtained directly from the Siglent can be seen in Figure 5[2]. It shows the current over time (in minutes) for Heapsort using different k-values, i.e., $k \in \{2, 4, 8, 16, 32, 64\}$ on FPGA inputs of size 4096. The k values are not tested in order and are annotated above the associated execution. In the graph, we also see error-prone executions without annotations. The $k = 16$ provides the energy-optimal execution since it has the least area under the curve, i.e., the shortest execution time and the lowest current (the voltage is fixed).

[2] The original experimental measure data was lost, and we cannot provide precise energy consumption.

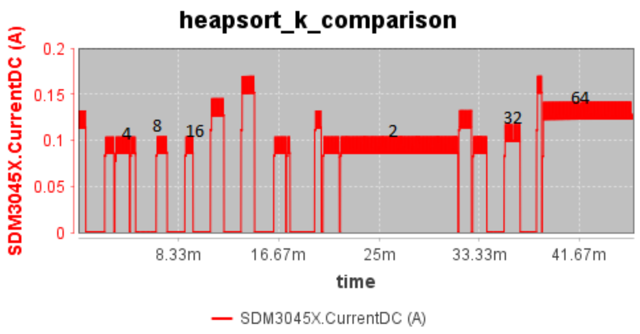

Fig. 5. The current over time for Heapsort using different $k \in \{2, 4, 8, 16, 32, 64\}$ on FPGA inputs of size 4096. The k-values are not tested in order; the k-values are annotated above the associated execution. In the graph, we also see error-prone executions without annotations. The $k = 16$ provides the energy optimal execution.

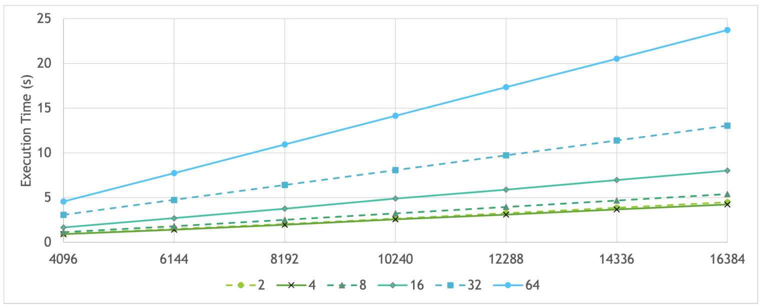

Fig. 6. The average execution time in seconds for Heapsort using a k-heap on Raspberry Pi over inputs from 4096 to 16384. The $k = 4$ provides the fastest executions.

Software The growth rate is almost linear, as expected from an algorithm that runs at $n \log_k n$. Interestingly, the constant factor does matter since the fastest time is with $k = 4$. For bigger k, finding the subtree with the biggest root note becomes the main bottleneck in the execution.

4.3. Measuring and Adjustment of Repetitions

Description of what we have included in our measurements which hardware and software are we using.

```
...
const int MAX=1000000;

#define mx 500000

int main(){
  int mx1=mx/n;
  int lng=0;
  for(int n1=0;n1<n;n1++){
    for(int k1=0;k1<mx1;k1++){

    int start=n1;
    // dijkstra start
    ...
    // dijkstra end
    while(cur!=start){
      lng+=wght[cur];
      cur = prev[cur];
    }
    }
  }
  printf("done %d\n",lng);
  return 0;
}
```

Fig. 7. This Dijkstra's shortest path algorithm will calculate 500.000 routes switching between the n possible start vertices.

We measure the energy consumption for the entire Raspberry Pi using a programmable power supply Siglent SPD3303X-E Linear DC 3CH in connection with a Siglent SDM3045X Digital Multimeter is a 4 1/2 digit (66,000 count) multimeter. This setup allows for the required high-precision readings of the current. The equipment spans the current of both the FPGA and the Raspberry Pi and allows us to measure the power dissipation each 100ms. To ensure a large enough sample size of the current, we adjust the total execution time to be at least 3 seconds. The adjustments occur within both the software and hardware implementation. For the adjustments within the software implementations, see Figure 8 for the setup used in Heapsort and Figure 7 for the setup in Dijkstra. See Table 1 for a precise

```
#define REPETITIONS = 3000
// heapsort start
...
// heapsort end
int main () {
  for (int i = 0; i < REPETITIONS; i++) {
    for (int j = 0; j < N; j++) a[j] = j;
    heapsort(a, N);
  }
  return 0;
}
```

Fig. 8. Adjustment of repetitions in the Heapsort software application.

number of iterations used in the hardware and software implementations. In addition, this adjustment will reduce the synchronization imprecision between the equipment and the hardware.

Mode	Algorithm	input size/type	Repetitions
Software	Heapsort	all	3000
Hardware	Heapsort	4096	1550
Hardware	Heapsort	6144	1033
Hardware	Heapsort	8192	775
Hardware	Heapsort	10240	620
Hardware	Heapsort	12288	516
Hardware	Heapsort	14336	442
Hardware	Heapsort	16384	387
Software	Dijkstra	all	500000
Hardware	Dijkstra	all	9000

Table 1. Overview of the number of in-algorithm repetitions that ensure a least execution time of 3 seconds.

5. Results

Our results demonstrate significant and contrasting differences in energy efficiency and performance between Heapsort and Dijkstra's software and hardware implementations. These findings may necessitate more nuanced insights into the optimal hardware selection based on the algorithmic demands.

5.1. Heapsort

Comparative data on Heapsort's time and energy consumption in an FPGA and the Raspberry Pi is shown in Figure 9 and Table 2, with a focus on the energy efficiency achieved

	Raspberry Pi			FPGA		
Input size	time pr. ite. (ms)	power (W)	Ener. pr. ite. (mJ)	time pr. ite (ms)	power (W)	Ener. pr. ite. (mJ)
4096	1.541	3.043	4.689	2.102	0.431	0.906
6144	2.383	3.050	7.269	3.614	0.571	2.062
8192	3.296	3.050	10.055	4.959	0.572	2.837
10240	4.259	3.032	12.913	6.513	0.573	3.731
12288	5.166	3.047	15.741	7.922	0.572	4.533
14336	6.150	3.064	18.847	9.944	0.569	5.656
16384	7.066	3.080	21.764	11.213	0.378	4.241

Table 2. Heapsort's energy consumption within FPGA (k=16) and Raspberry Pi (k=4)

through hardware implementation. These results demonstrate energy savings and time costs when employing FPGAs over traditional CPUs. It is worth noticing that while the power dissipation by software and hardware implementations are different, they seem constant for the individual implementation.

The energy consumption of the Heapsort implementation on the Raspberry Pi is highly correlated to the execution time. The time consumption follows the expected $n(\log n)$ time complexity, and there is no significant increase in energy consumption when larger parts of memory are used during execution.

The total execution times are greater than 3 seconds, see Table 3; the total executions times are not directly comparable because the number of repetitions differ. This figure also shows a 2-second difference between the Siglent and the Raspberry time measurements. This is because we have chosen to count an experiment as when the Raspberry current increases beyond a certain threshold; in this case, when the current reaches 0.55A and more. This methodology cuts the execution time short on both ends. The energy consumption is calculated based on the logged times from the Raspberry Pi and the average power dissipation. While the execution time becomes correct, this method increases the average power dissipation slightly, and thus, we report a slightly larger energy consumption for the Raspberry Pi.

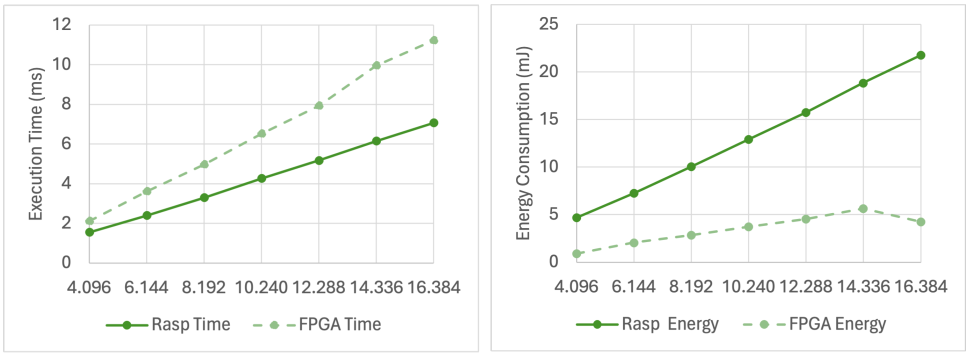

Fig. 9. Heapsort's time and energy consumption within FPGA (k=16) and Raspberry Pi (k=4).

Experiment	Raspberry Pi		FPGA
	(Sig.)	(Raspb.)	
4096	2.536	4.623	3.258
6144	5.017	7.15	3.733
8192	7.712	9.888	3.843
10240	11.388	12.777	4.038
12288	13.672	15.499	4.088
14336	16.495	18.450	4.395
16384	19.095	21.198	4.34

Table 3. Heapsort's total execution time per input. They are not directly comparable because the number of repetitions differ.

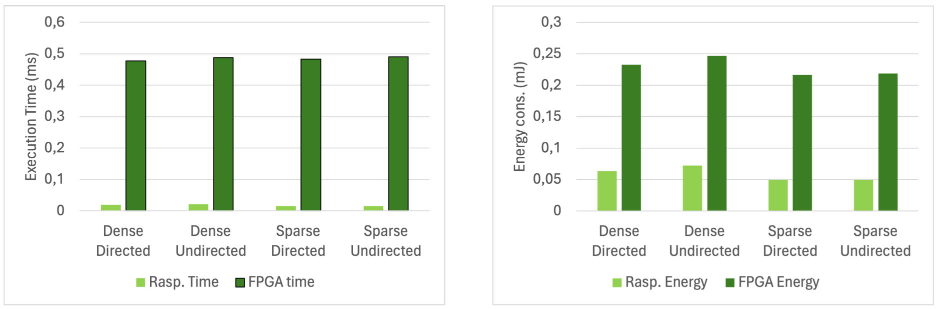

Fig. 10. Dijkstra: Avg. Time and Energy consumption within FPGA and Raspberry Pi.

5.2. Dijkstra

The results show that Dijkstra's shortest path algorithm does not necessarily benefit from hardware implementations when considering performance or energy, see Figure 10. The FPGA platform demonstrates higher average times and energy consumption per iteration than the Raspberry Pi across various graph configurations. For example, in scenarios like dense directed graphs, it consumed 0.232 mJ per iteration compared to 0.064 mJ by the Raspberry Pi, demonstrating a significantly higher energy usage, see Table 4.

These results suggest that while FPGAs are typically considered for their potential to enhance performance through parallelism, their energy efficiency for Dijkstra's algorithm is less effective than traditional processing on a Raspberry Pi. This revelation is particularly crucial for applications where energy consumption is as critical as processing speed, such as in portable or embedded devices where power conservation is essential.

6. Related Work

There are many studies on hardware implementations of algorithms, e.g., [10, 12, 13, 16–19, 24, 26], however, only a few focuses or touches upon energy consumption of implementations [13, 18] or compare across platforms [12, 16].

The first study by Jmaa et al. [12] focused on the acceleration of various sorting algorithms using FPGAs through high-level synthesis, comparing FPGA performance to other

Experiment	Raspberry Pi time pr. ite. (ms)	Raspberry Pi Power (W)	Raspberry Pi Energy pr. ite. (mJ)	FPGA time pr. it. ((ms)	FPGA Power (W)	FPGA Energy pr. ite. (mJ)
Dense Directed	0.019	3.32	0.064	0.477	0.488	0.232
Dense Undirected	0.021	3.377	0.072	0.488	0.505	0.246
Sparse Directed	0.015	3.242	0.049	0.483	0.448	0.216
Sparce Undirected	0.015	3.260	0.049	0.49	0.446	0.218

Table 4. Dijkstra's average energy consumption within FPGA and Raspberry Pi.

platforms like CPUs. Their objective is to demonstrate the benefits of FPGA acceleration in terms of execution time and standard deviation of execution times. While the purpose is different, our studies overlap in the focus on sorting algorithms but differ in the usage of FPGA, evaluation metrics, and devices.

The exploratory study by Kirkeby and Schoeberl [16] compares the performance of hardware and software implementations and indicates possible energy consumption. The study is limited in that it focuses on performance, but it aligns well with our study in that it compares hardware and software implementations.

The studies by Jmaa et al. [13] evaluate sorting algorithms implemented into FPGAs. It compares various sorting algorithms, including InsertionSort, QuickSort, HeapSort, ShellSort, MergeSort, and TimSort solely on a software platform (ARM Cortex A9 processor part of the Zynq Zedboard). The study primarily measures computational time, energy consumption, and stability to find the most efficient algorithm for embedded systems applications. This study is close in content to our study, but they do not compare the hardware implementations with other implementations. Instead, they compare the time and energy usage of their FPGA implementations. Equivalent to our observation, they found that the FPGA's power dissipation is approximately constant when evaluating their hardware implementations, and thus, we confirm their high correlation between energy and time.

7. Discussion of Results

Our results reveal new trade-off considerations between algorithm efficiency and performance. This discussion contextualizes our findings, leading directly to Section 9 where we summarize the key insights and summarize avenues for further research.

7.1. Algorithm Characteristics and Hardware Suitability

Heapsort and Dijkstra's algorithms have different levels of inherent parallelism and complexity. Heapsort may benefit more from native parallelization in hardware due to its ability to efficiently parallelize the comparison and swapping elements, especially when sorting larger datasets. In contrast, our implementation of Dijkstra's algorithm may not fully leverage FPGA capabilities due to its sequential dependencies, leading to less impressive gains or even inefficiencies on FPGAs.

While our results are less promising for Dijkstra, previous studies show that FPGA implementations of Dijkstra can provide considerable performance optimization compared to software due to their different growth rates: "The average execution time of the FPGA-based version grew only linearly, whereas the average execution time of the microprocessor-based version displayed quadratic growth [24], see Table 5. Thus, it may be that increasing the graph size can improve the results, but perhaps our result is a consequence of our implementation. Future work would include reimplementations and energy evaluations of previously successful hardware implementations.

Vertices	Logic Elements	Memory bits	Execution time(FPGA-based)	Execution time (μP-based)	Average speedup factor
8	834	632	10.6μs	250μs	23.58
16	1536	2116	13.4μs	434μs	32.39
32	2744	8287	17.2μs	802μs	46.63
64	5100	32894	21.6μs	1456μs	67.41

Table 5. Dijkstra performance improvements by Tommiska et al. [24].

7.2. Energy Efficiency versus Performance

Both algorithms highlight the trade-offs between performance gains and energy efficiency. In both algorithms, the FPGAs increased execution time. Energy and time are highly correlated. However, comparing the execution time across software and hardware does not necessarily indicate a similar pattern for their energy consumption.

The benefit of the FPGA is that the power dissipation is 5 to 8 times lower than the Raspberry Pi's power dissipation. Therefore, a good rule of thumb would be that FPGA is a good choice when the execution time of the FPGA is 5 to 8 times faster than the Raspberry Pi. In our study, this is the case for Heapsort but not for Dijkstra. In addition, this factor may change with a different choice of hardware. Future work would include evaluating the energy and time trade-off factors for various FPGAs and computers.

7.3. System Architecture Design

Insights from both algorithms can guide system architects in designing more efficient systems by choosing the right combination of hardware and software based on the specific algorithms they expect to run most frequently. There is no clear case for always employing FPGAs to provide energy reductions equivalently to performance. However, in some cases, the developer may be able to exploit native parallelism and ensure that hardware implementation has a slower execution time growth rate for increasing input sizes. It would be beneficial to evaluate the energy consumption of high-performing hardware implementations and to identify trade-off trends on the architectural design level.

7.4. Hypotheses on Algorithmic Performance

Our results demonstrate that Heapsort, with its potential for parallel processing, capitalized on the FPGA's architecture to yield significant energy savings. One hypothesis for this outcome is that the FPGA's ability to conduct multiple comparisons in parallel, particularly through the efficient use of LUTs and parallel memory access, minimizes both time and energy when compared to the CPU's more serial processing. For instance, as the input size increased, the FPGA continued to scale effectively, likely due to its architectural suitability for handling concurrent operations. This suggests that algorithms with similar local parallelism would exhibit comparable performance benefits.

In contrast, Dijkstra's algorithm, with its inherent sequential dependencies, struggled to take advantage of FPGA's strengths. We hypothesize that this inefficiency stems from the algorithm's need to update path lengths iteratively, which bottlenecks performance and limits potential energy gains. Future research could investigate whether alternate graph traversal algorithms with fewer sequential dependencies, or different graph configurations (such as sparse versus dense graphs), might better exploit hardware acceleration. Additionally, hardware design optimizations targeting these sequential steps could mitigate some of these limitations.

8. The Limits of Speedup

Building on the hypothesis that algorithms with local parallelism would exhibit similar performance benefits, we explored the upper bounds of FPGA speedup using the highly parallelizable Conway's Game of Life [3]. This experiment serves to demonstrate how extreme parallelism can push the limits of performance and energy efficiency on FPGA, further supporting the idea that parallel algorithms are well-suited for hardware acceleration. Conway's Game of Life is a zero-player game defined on cellular automata. The cellular automata are 2D grids called worlds, where each grid cell has eight neighbors, and each cell can have one of two states: dead or alive. For each step, the cell states are updated according to their state and the states of their neighboring cells in the previous step.

1. Any live cell with two or three live neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

The initial state is given as input to the program. Because each cell depends only on nearby cells, Game of Life is highly parallelizable.

Table 6 shows the average execution time for a single time step and the speedup provided by the FPGA implementation compared to the Java software implementation executed on a MacBook Pro and the Raspberry Pi. These results show that the gain in performance increases with the measured world sizes.

Speedup factors range from 25 for a 10x10 world to 1500 for a 100x100 world. The speedup scales linearly with the problem size. While the Game of Life is an artificial workload, its parallelizable nature made it an ideal candidate for indicating an upper bound for speedups when moving algorithms from software into an FPGA.

8.1. Resource Utilization

We have implemented the Game of Life of different sizes in a Cyclon IV FPGA found on the DE2-115 evaluation board. We report the design size in logic elements (LEs) and

Table 6. The execution time in us and speed-up of FPGA over software executions.

		Execution time per step (us)			FPGA Speedup	
World	Cells	Mac	Rasperry	FPGA	Mac	Rasperry
10x10	100	0.10	1.783	0.0040	25	445
20x20	400	0.33	5.137	0.0040	82	1284
30x30	900	0.70	9.965	0.0041	170	2430
40x40	1600	1.21	17.212	0.0040	302	4302
50x50	2500	1.81	25.204	0.0044	411	5728
60x60	3600	2.76	37.822	0.0045	613	8404
70x70	4900	3.54	57.665	0.0040	884	14416
80x80	6400	4.81	64.396	0.0047	1023	13701
90x90	8100	6.50	81.309	0.0045	1444	18068
100x100	10000	7.51	109.964	0.0048	1564	22909

registers. An LE represents one 4-bit lookup table. For synthesis, we used the Quartus 19.1.0 Lite Edition.

Table 7 shows the FPGA implementation's resource consumption for different world sizes. We can see that the size grows linear. The maximum frequency of the circuit is reported between 209 MHz and 250 MHz. Therefore, when we assume running it at 200 MHz we can compute one iteration in 5 ns.

Table 7. The resource utilization and minimum iteration time of different sized Game of Life worlds in an FPGA.

Size	LEs	Registers	min. Clock Period
10 x 10	804	104	4.0 ns
20 x 20	3539	404	4.0 ns
30 x 30	7995	904	4.1 ns
40 x 40	14463	1604	4.0 ns
50 x 50	23439	2504	4.4 ns
60 x 60	34414	3604	4.5 ns
70 x 70	45119	4904	4.0 ns
80 x 80	59136	6404	4.7 ns
90 x 90	75102	8104	4.5 ns
100 x 100	97871	10004	4.8 ns

As expected, we use one register per cell. However, the number of LEs per cell is surprisingly high, an average of around 9 LEs per cell. We assume that the Chisel Pop-Count method has some room for improvement. However, as we aim for a technique that enables software developers to describe their algorithms in hardware, we are avoiding optimization tricks.

8.2. Estimated Energy Consumption

We did not measure power or energy consumption of the FPGA implementation. However, the DE2-115 FPGA board comes with a power supply of 24 W. Therefore, this is the upper bound of power consumption of the whole FPGA board, including peripheral devices and external memories.

If we assume 24 W as an upper bound on the power consumption and an operating frequency of 200MHz, then one iteration of a 100 x 100 Game of Life world consumes 96nJ. In comparison, the Raspberry Pi has been reported to consume an average of 6.4 W when all four cores are busy[3] and one iteration of a 100x100 world takes 0.109964 ms. Thus, a conservative energy consumption estimate for one iteration of a 100x100 world is 0.703769 mJ. From these conservative estimates, the hardware implementation can significantly improve energy consumption compared to the Raspberry Pi 4.

9. Conclusion

This study compared the energy efficiency and performance of software and hardware implementations of Heapsort and Dijkstra's algorithms. Our findings reveal both the advantages and limitations of using FPGAs compared to traditional software implementations on a Raspberry Pi.

For Heapsort, the hardware implementation on an FPGA demonstrated a clear advantage in terms of energy efficiency, confirming the potential of FPGAs for algorithms where some operations can be done in parallel, The results showed that the energy consumption for Heapsort on FPGA was consistently lower than on the Raspberry Pi, particularly as input sizes increased. This suggests that FPGAs can effectively reduce energy consumption for tasks where some parallel processing can be exploited.

In contrast, Dijkstra's algorithm did not exhibit the same level of energy efficiency on FPGA. Despite FPGAs' inherent capabilities for handling parallel tasks, the complex dependencies and sequential nature of Dijkstra's algorithm limited the expected gains. The study highlighted that traditional CPU implementations might still hold an advantage in terms of both performance and energy consumption for algorithms with significant sequential operations.

The comparison also underscored the importance of choosing hardware or software solutions based on the specific requirements and characteristics of the algorithm. While FPGAs offer considerable reductions in power dissipation, they are not universally superior for all computational tasks. Our findings suggest that the decision to use FPGA over CPU should be guided by a more detailed knowledge of the algorithm's structure and the potential for parallelism.

Additionally, this study contributes to the ongoing discussion about the trade-offs between computational speed and energy efficiency. It provides a first step towards a nuanced perspective that can aid system architects and developers in making informed decisions about the hardware-software configurations that best meet their performance and efficiency goals.

While this study focuses on Heapsort and Dijkstra's algorithm, both of which offer limited parallelism, future work will explore more parallelizable algorithms, such as

[3] `https://www.pidramble.com/wiki/benchmarks/power-consumption`

quicksort or matrix operations, to better demonstrate FPGA's energy and performance potential. Our preliminary results with Conway's Game of Life (Section 8) show that substantial speedups can be achieved through parallel computation, and similar gains are expected from a wider range of algorithms. This suggests that algorithms with similar characteristics can significantly reduce energy consumption by distributing computational tasks evenly across FPGA's processing elements. We hypothesize that applying this approach to a wider range of highly parallelizable algorithms will result in similar improvements in both energy efficiency and performance, as distributing tasks across FPGA's processing elements can significantly reduce energy consumption. These extensions will provide further insights into how parallelism can be optimized to enhance both computational speed and energy savings across a variety of workloads.

Future work will focus on optimizing hardware implementations and expanding the range of algorithms tested to further explore their energy and performance potential. Additionally, we plan to evaluate the scalability and applicability of our findings across a wider variety of hardware platforms, including more powerful multicore processors, GPUs, and different FPGA models and configurations. This broader assessment will help establish more generalized guidelines for selecting between software and hardware implementations, particularly in terms of energy efficiency and performance metrics.

References

1. Anders S G Andrae. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letter*, 3:19–31, 2020.
2. Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012.
3. Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays. Vol. 2.* Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1982. Games in particular.
4. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms. third. *New York*, 2:187–195, 2009.
5. Kerstin Eder, John P. Gallagher, Pedro López-García, Henk Muller, Zorana Banković, Kyriakos Georgiou, Rémy Haemmerlé, Manuel V. Hermenegildo, Bishoksan Kafle, Steve Kerrison, Maja Kirkeby, Maximiliano Klemen, Xueliang Li, Umer Liqat, Jeremy Morse, Morten Rhiger, and Mads Rosendahl. Entra: Whole-systems energy transparency. *Microprocessors and Microsystems*, 47:278–286, 2016.
6. Hayden Field, Glen Anderson, and Kerstin Eder. Eacof: A framework for providing energy transparency to enable energy-aware software development. *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1194–1199, 2014.
7. Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan 2012.
8. Hiroaki Hirata and Atsushi Nunome. A modified parallel heapsort algorithm. *International Journal of Software Innovation*, 8(3):1–18, 2020.
9. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2023. Accessed: 2024-01-29.
10. G. R. Jagadeesh, T. Srikanthan, and C. M. Lim. Field programmable gate array-based acceleration of shortest-path computation. *IET computers & digital techniques*, 5(4):231–237, 2011.

11. Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1811–1815, 2012.
12. Yomna Ben Jmaa, Rabie Ben Atitallah, David Duvivier, and Maher Ben Jemaa. A comparative study of sorting algorithms with fpga acceleration by high level synthesis. *Computacion y Sistemas*, 23:213–230, 2019. Comparison: FPGA vs. CPU, very clear!¡br/¿Are they doing the same thing in 2021 paper?
13. Yomna Ben Jmaa, David Duvivier, and Mohamed Abid. Sorting algorithms on arm cortex a9 processor. In Leonard Barolli, Isaac Woungang, and Tomoya Enokido, editors, *International Conference on Advanced Information Networking and Applications*, volume 227, pages 355–366. Springer International Publishing, 2021.
14. Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 3(2):1–26, jun 2018.
15. Maja H. Kirkeby, Thomas Krabben, Mathias Larsen, Maria B. Mikkelsen, Tjark Petersen, Mads Rosendahl, Martin Schoeberl, and Martin Sundman. Energy consumption and performance of heapsort in hardware and software, 2022.
16. Maja H. Kirkeby and Martin Schoeberl. Towards comparing performance of algorithms in hardware and software, 2022.
17. Dirk Koch and Jim Torresen. Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 45–54, New York, NY, USA, 2011. Association for Computing Machinery.
18. Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. An fpga implementation for solving the large single-source-shortest-path problem. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(5):473–477, 2016.
19. Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, aug 2009.
20. Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Sci. Comput. Program.*, 205:102609, 2021.
21. Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.
22. Bernardo Santos, João Paulo Fernandes, Maja H. Kirkeby, and Alberto Pardo. Compiling haskell for energy efficiency: Empirical analysis of individual transformations. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain*. Association for Computing MachineryNew YorkNYUnited States, 2023.
23. Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. available at `https://github.com/schoeberl/chisel-book`.
24. Matti Tommiska and Jorma Skyttä. Dijkstra's shortest path routing algorithm in reconfigurable hardware. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2147:653–657, 2001.
25. Xilinx. Vivado design suite user. guide power analysis and optimization, October 2021. UG907 (v2021.2).
26. Yuzhi Zhou, Xi Jin, and Tianqi Wang. FPGA implementation of a_* algorithm for real-time path planning. *Int. J. Reconfigurable Comput.*, 2020:8896386:1–8896386:11, 2020.