Scheduling Methods for OpenVX Programs on Heterogeneous Multi-core Systems

Tzu-Hsiang Lin, Cheng-Yen Lin, and Jenq-Kuen Lee
Department of Computer Science, National Tsing-Hua University, Taiwan
Email: thlin@pllab.cs.nthu.edu.tw, kennylin@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw

Abstract—Heterogeneous multi-core architectures are playing an important role in improving the overall performance of computer systems. To program such systems, OpenVX [1] promises to provide a standard programming framework for computer vision processing. OpenVX is with a graph-based execution model to describe the computation behavior and data flow relationship. Each computation node in the graph can be dispatched to a different target, such as multicore CPUs with C, OpenMP runtime, OpenCL on GPUs, or a dedicated hardware. Therefore, how to efficiently schedule all the computation nodes to those different targets opens up the optimization opportunities.

In this paper, we propose a method to schedule OpenVX task graph by considering both memory locality and system throughput. The proposed two phase scheduling method first perform coarsen schemes to cluster nodes together, and then in the second phase a scheduling method is employed to schedule nodes into different targets. Preliminary experiments show that our scheme works well in scheduling OpenVX programs on heterogeneous environments.

Keywords: OpenVX, scheduling, coarsen, heterogeneous systems, computer vision

1. Introduction

In recent years, heterogeneous multi-core architectures are playing an important role in improving the overall performance of computer systems. Many chips are designed with different types of cores (such as GPU, DSP, hardware accelerator, etc.) integration. It challenges the programmer on programming, tasks scheduling and resource management. To ease the challenge of programming such systems, OpenVX [1] promises to provide a standard programming framework for computer vision processing. OpenVX is an open, royalty-free standard proposed by Khronos Group. It is an emerging programming framework for computer vision processing. As the OpenVX specification only defines the functional requirements, implementers can accelerate their implementation by applying a variety of optimization techniques for different optimization objectives. For example, different scheduling policies can be devised to meet the performance requirements or power constraints for mobile devices. Also, one can try to reduce the data transfer overhead by considering memory layout and utilizing the local memory. OpenVX is also with a graph-based execution model to describe the computation behavior and data flow relationship. Each computation node in the graph can be dispatched to a different target, such as multicore CPUs with C, OpenMP runtime, OpenCL on GPUs, or a dedicated hardware. Therefore, how to efficiently schedule all the computation nodes to those different targets opens up the optimization opportunities.

In this work, we address the scheduling issues of OpenVX task graph by considering both memory locality and system throughput. Our proposed two phase scheduling method first perform coarsen schemes to cluster nodes together, and then in the second phase, a scheduling method is employed to schedule nodes into different targets. In our work, we reference the work in [4] for static task-scheduling algorithms classification and the notion of graph attribute definitions such as upward and downward ranking, comparison metrics and the concept of HEFT, and CPOP algorithm. In addition, the task clustering problem is an NP-hard problem and is shown in [5]. Task time estimation model and load blancing mechanism are introduced in [6]. The program features with profiling can be seen in [7]. Moreover, many different optimization strategies can be found in the work [8]. Preliminary experiments show that our scheme works well in scheduling OpenVX programs on heterogeneous environments.

The remainder of this paper is organized as follows. Section 2 overviews the OpenVX programming. Section 3 presents our scheduling methods. Next, Section 4 presents the preliminary experimental results and Section 5 concludes this paper.

2. Background with OpenVX Programming

For the sake of completeness, we introduce OpenVX programming in this section. OpenVX is an open standard programming framework for accelerating portable computer vision applications on different computation targets. The optimized implementations of OpenVX framework are provided by hardware vendors, while software developers will find a unified abstraction interface to utilize accelerated implementation either in software or hardware. With the APIs, one hopes both functional and performance portabilities across diverse devices and platforms can be enable for
OpenVX applications.

In OpenVX framework, it uses the graph execution model to describe the computation behavior and data flow relationship by composing the computation nodes and data objects. A computation node is an instance of a computer vision kernel that combines with reference parameters, target affinity for execution, and associated graph. In other words, a computer vision function can be implemented as a kernel with parameter signatures, parent graph reference, and target options if the node has corresponding implementation on that targets. A target means an executor that can execute the OpenVX kernels, such as a multicore CPU with C or OpenMP runtime, general purpose computing GPUs with OpenCL or a dedicated hardware. It decouples the client code from a specific vendor’s configurations or technologies and leaves the flexibility for vendor’s implementations. Currently, the official sample implementation supports only C model, OpenMP, and OpenCL targets. All data objects in OpenVX framework are opaqueness. To avoid facing hardware-specific memory structures, the knowledge of memory location and data layout is controlled by the OpenVX framework implementation. The edges between computation nodes and data objects indicate the data dependency and data objects between nodes in a graph can be defined as virtual data objects to reveal optimization opportunities. In summary, the graph execution model is a directed acyclic graph (DAG) that determines the computation process of an OpenVX application.

The programming flow for an OpenVX application is summarized in Figure 2. There are six major phases in the programming flow. First, we need to create a context for the OpenVX framework to manage the reference counts on all objects. This is a necessary parameter for any manipulation with this framework. Using the context, we can construct the graph in the second phase. In this phase, we define needed data objects and connect them as a graph by distributing the computation nodes and the input and output reference of data objects for nodes. Third, the graph needs to be verified once as the structure of graph has not been verified or has been modified. Fourth, it starts the computation process by issuing the graph to OpenVX framework. We can execute this graph multiple times if needed. Fifth, it destructs the graph by calling the API to release nodes, data objects, and graph resources. Finally, it releases the OpenVX context and exits the program.

Figure 1 illustrates an OpenVX task graph of a Haar face detection application. The corresponding code segment is shown in Figure 3. We use oval shape with broad stroke to represent computation nodes while the rectangle shapes represents data objects. To focus on the processing flow of the OpenVX graph, we use function call to represent the routines about image reading (ReadInputImage), result drawing (DrawResult), and resource releasing (ReleaseResource). Moreover, we also use the function call (PublishCustomNodes) to omit the codes manipulate user-defined nodes API and for the process about publishing our custom nodes (vxIntegralColumn, vxIntegralRow, vxIntegralSquare and vxViolaJones) to OpenVX framework. The input image(src) will be first converted into gray image(d[1]) by

```
vx_context c = vxCreateContext();
vx_graph g = vxCreateGraph(c);
vx_uint32 w=640, h=480, resSize=100;
PublishCustomNodes(&c);
vx_image src = vxCreateImage(c,w,h,VX_DF_IMAGE_RGB);
vx_array results = vxCreateArray(c,VX_TYPE_COORDINATES2D, resSize);
vx_image d[] = {
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_IYUV),
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_U8),
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_U8),
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_U8),
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_U8),
  vxCreateVirtualImage(g, w, h, VX_DF_IMAGE_U8),
};
vx_node n[] = {
  vxColorConvertNode(g,src,d[0]),
  vxChannelExtractNode(g,d[0], VX_CHANNEL_Y,d[1]),
  vxIntegralColumn(g,d[1],d[2]),
  vxIntegralColumn(g,d[2],d[3]),
  vxIntegralRow(g,d[2],d[4]),
  vxIntegralSquare(g,d[3],d[5]),
  vxViolaJones(g,d[4],d[5], results);
};
if (vxVerifyGraph(g)==VX_SUCCESS) {
  ReadInputImage(w,h,&src);
  ProcessGraph(g);
  DrawResult(src, results);
  ReleaseResource(&c,g,d,n,&src,&results);
}
```

![Fig. 1: An OpenVX graph of Haar face detection application](image1.png)

![Fig. 2: OpenVX code flow](image2.png)

![Fig. 3: Code snippet of OpenVX Haar face detection](image3.png)
color-convert(n[0]) and channel-extract nodes(n[1]). Then this gray image is fed into two independent nodes to do the integral computation (node n[2] to node n[5]). After that, two integral images (d[4], d[5]) are merged in nodes(n[6]) doing cascade features computation using Viola Jones algorithm. Finally, the output rectangles can be obtained in an array (results). We can observe from the above process that some nodes on the paths (n[2],n[4] and path n[3],n[5]) can be parallel processing. In addition, virtual data objects(d[0] to [5]) will not be accessed by host, it can be optimized into a local memory residing on compute devices. For example, in OpenCL, we can make these memory objects as buffers on GPU devices, and read/write operations only occur when it is with host access or one needs to transfer data to different targets.

3. Coarsen-Scheduling algorithm

The Coarsen-Scheduling algorithm aims to improve the targets utilization and the memory locality in OpenVX framework. Computation nodes in the graph are dispatched to a specific target after processing by our two phase algorithm. The input of this algorithm are an OpenVX graph and a sequence of OpenVX target denoted as $P = \{p_0, p_1, \ldots, p_n\}$. An OpenVX graph is not only a weighted DAG as we mentioned above, but also a bipartite graph that can be denoted as $G = (N, D, E)$. By using this bipartite representation, we divide the graph into two sets and one edge set. One of the node sets is the computation nodes set $n_0 \ldots n_i$, and the other is the data objects set $d_0 \ldots d_j$. Each edge in the edges set $E$ represents the read or write relationship between a computation node in the computation nodes set $N$ and a data object in the data objects set $D$. In this algorithm, we process an OpenVX graph in two phases: the node coarsen phase and the node scheduling phase. We now detail the process of each of the two phases.

3.1 Node coarsen algorithm

The node coarsen algorithm groups nodes on the OpenVX graph into clusters. With the input graph and target information, the output of this algorithm is a cluster set, $C$. Nodes in the same cluster will be forced to dispatch to the same target in the scheduling phase, so we only select the nodes which can reduce data transfer time or largely shorten the computation time into the same cluster. The remaining nodes not clustered will be scheduled for the target in the scheduling phase. Because the structure of an OpenVX graph is fixed after the verification stage, we just need to coarse once for a new graph structure. This algorithm will execute in verification phase(vxVerifyGraph(graph)). Before introducing the details of this algorithm, we first give the auxiliary definitions for the clustering steps. The computation time $\mu(n, p_j)$ is the time a node $n \in N$ spent on the OpenVX target $p_j$ doing computation. The data transfer time $\lambda(d, u, v)$ is the time to transfer data $d \in D$ between computation nodes $u \in N$ and $v \in N$. To get the $\mu$ and $\lambda$ value, we need to profile each computation node on each target that OpenVX framework provides at the first time with a fixed data size. Moreover, we assume a linear relationship for data size and computation time, data size and data transfer time. So we will use a quadratic function to evaluate $\mu$ and $\lambda$ value. The $\text{transfer}(d)$ is the maximum time to transfer data $d \in D$ from one writer node $w \in N$ to multiple readers $R \subset N$ defined by

$$\text{transfer}(d) = \max_{r \in R}(\lambda(d, w, r))$$

According to the graph formalism rules defined in the OpenVX specification, every data object $d \in D$ has only a single writer node $n \in N$. So the maximum time to transfer data $d$ must include the output edge from a writer node $w$ connecting to $d$ and one of the edges as input edge connect from $d$ to a reader node. Finally, the improved factor $\text{improve}(n)$ is the maximum difference for computation node $n \in N$ running on target $p_i$ and $p_j$ defined as

$$\text{improve}(n) = \frac{\max_{p_i \in P} \mu(n, p_i)}{\min_{p_j \in P} \mu(n, p_j)}$$

where $p_j$ is the target that require longest computation time to execute computation node $n$.

**Algorithm 1: NodeCoarsen(G, P)**

**Input:** an OpenVX graph: $G = (N, D, E)$, an OpenVX target sequence: $P$

**Output:** nodes for targets $C$

1. Each computation node $n \in N$ assign to a target $p \in P$ that minimize $\mu(n, p)$;
2. Compute the critical path $CP$;
3. $\text{clusterCP}(CP, C)$;
4. $\text{clusterG}(G, C)$;

We now present our node coarsen algorithm in algorithm 1. Initially, we assign each computation node $n \in N$ to a target $p \in P$ that cost minimal computation time in line 1. Then, we compute the critical path in line 2 which will be used in the following cluster procedures. A critical path $CP$ can be computed by traversing the task graph upward while every computation node $n \in N$ choose the lowest computation time target. Starting from the exit node $n_{exit}$, the upward rank of a node $n$ is recursively defined by

$$\text{rank}(n) = \mu(n, p) + \max_{d \in \text{write}(n)} (\text{transfer}(d) + \text{rank}(MTR))$$

where $\text{write}(n)$ is the set of immediate successors of $n$, that is the set of data objects that are written by computation node $n$. In addition, $MTR$ is the reader node that cause maximum
transfer(d) time. Due to the data input of source nodes and data output of exit nodes should back to host target, the rank value of exit node rank(n_{exit}) is \( \mu(n_{exit}, p) \) plus the data transfer time of each output data:

\[
\text{rank}(n_{exit}) = \mu(n_{exit}, p) + \sum_{\text{out} \in \text{output}(n_{exit})} \lambda(\text{out}, n_{exit}, \text{host_target})
\]  

Since rank values are recursively calculated by traversing upward, the largest rank value on the head node plus the data transfer time of each input data is the makespan of the graph. We then divide the following steps of this algorithm into two procedures: \text{clusterCP} in line 3 to cluster nodes on critical path first and \text{clusterG} in line 4 to cluster the other nodes not on the critical path into clusters.

**Procedure clusterCP(CP, C)**

**Input:** Critical path: CP, nodes for targets C  
**Output:** nodes for targets C

1. while there are unvisited data object on CP do
   2. Choose an unvisited data object \( d \in CP \) with maximum \( \lambda(d, w, r) \), where computation nodes \( w \in CP \) and \( r \in CP \);
   3. if \( w.target=target \) then
      4. if isValidWindowSize(w, target, w) and isValidWindowSize(w, target, r) then
         5. \( C(w.target) \leftarrow C(w.target) \cup w \cup r \);
      6. else
         7. \( \text{gain} \leftarrow \lambda(d, w, r) \);
         8. \( \text{changeW} \leftarrow \mu(w, r.target) - \mu(w, w.target) \);
         9. \( \text{changeR} \leftarrow \mu(r, w.target) - \mu(r, r.target) \);
         10. if \( w.visited=false \) then \( \text{changeW} \leftarrow \infty \);
         11. if \( r.visited=false \) then \( \text{changeR} \leftarrow \infty \);
         12. \( \text{cost} \leftarrow \min(\text{changeW}, \text{changeR}) \);
         13. if \( \text{gain} > \text{cost} \) then
            14. if \( \text{cost} = \text{changeW} \) then
               15. \( w.target \leftarrow r.target \);
            else
               16. \( r.target \leftarrow w.target \);
            if isValidWindowSize(w, target, w) and isValidWindowSize(r, target, r) then
               17. \( C(w.target) \leftarrow C(w.target) \cup w \cup r \);
               18. Re-Compute the critical path CP;
            end
      end
   end
22. Mark \( d, w, r \) as visited;
23. end

In the \text{clusterCP} procedure, we aim to shorten the time spent on the critical path, so the makespan of the graph can be shorten. This is a loop process until all data objects on the critical path are visited, excluding input data objects of source nodes and output data objects of exit nodes. We first choose a data object \( d \) on the critical path causing longest transfer time at line 2. Then we know the writer node \( w \) and reader node \( r \) connect to this data object \( d \) because there is only a writer and a reader can match the connection with data object \( d \) and on the critical path conditions. At line 3 to line 6, we try to cluster reader \( r \) and writer \( w \) into same cluster because the \( \mu \) value is minimal and transfer time on same target is minimal too. At line 8 to line 20, reader \( r \) and writer \( w \) are on different targets. We try to change reader or writer target to be the same target if gain earn more than cost. Critical path needs to be recomputed if target changes. At line 21, we mark the \( d, w, r \) as visited so the problem size will decrease. The isValidWindowSize procedure is a mechanism to limit the size of a cluster. We will discuss the impact of the evaluation method used in this procedure in the experiment section.

**Procedure clusterG(G, C)**

**Input:** OpenVX graph: G, nodes for targets C  
**Output:** nodes for targets C

1. while there are unvisited computation nodes in G do
   2. Choose an unvisited computation node \( n \) with maximum improve(n);
   3. \( \text{diff_count} \leftarrow 0 \);
   4. foreach data object \( d \) adjacency to \( n \) do
      5. foreach computation node adjn adjacency to \( d \) with different read/write direction as \( n \) do
         6. if \( \text{adjn.visited=true and adjn.target} \neq n \) target then
            7. \( \text{diff_count} \leftarrow \text{diff_count} + 1 \);
      end
   end
   8. if \( \text{diff_count} = 0 \) then
      9. if isValidWindowSize(n, target, n) then
         10. \( C(n.target) \leftarrow C(n.target) \cup n \);
      else
         11. Choose a data object \( d \) connect with visited computation node \( m \) which will cause maximum \( \lambda(d, n, m) \);
         12. \( \text{gain} \leftarrow \lambda(d, n, m) \);
         13. \( \text{cost} \leftarrow \mu(n, m.target) - \mu(n, n.target) \);
         14. if \( \text{gain} > \text{cost} \) and isValidWindowSize(m, target, n) then
            15. \( n.target \leftarrow m.target \);
            16. \( C(n.target) \leftarrow C(n.target) \cup n \);
         end
   end
   18. Mark \( n \) as visited;
20. end

After nodes on critical path are visited, the \text{clusterG} procedure process the nodes not on the critical path. The aim
of this procedure is to reserve most-earn computation nodes first. We loop until every nodes on the graph are visited. In each round at line 2, we choose the node causing maximum \textit{improve} value. In other words, the nodes take precedence in this clustering process that has largest time difference on the fastest target and slowest target. At line 3 to line 8, we check computation nodes connect with the same data nodes that adjacency to \( n \). If there are visited nodes, it is because they are on critical path and are marked as visited in the previous \textit{clusterCP} procedure. The \textit{diff\_count} will be non-zero if there exist visited nodes with different target. Therefore, at line 9 to line 11 is a simple case. We just try to make the node \( n \) into \( n\text{-target} \) cluster. While a complex case at line 12 to line 18, we find the most expansive data transfer cost edge and try to evaluate the data transfer gain and computation cost of node \( n \). If this trade-off is worth enough, then try to make the node \( n \) select target \( m\text{-target} \) and add into cluster. Finally, computation node \( n \) is mark as visited so the loop will stop.

We use the OpenVX graph shown earlier in Figure 1 as an example to explain how our node coarsen algorithm works. To simplify the analysis process, we assume each computation node has only two targets \( \text{target}_a \) and \( \text{target}_b \). In addition, we assume \( \text{target}_a \) is the OpenVX host target.

The data transfer time between \( \text{target}_a \) and \( \text{target}_b \) is symmetry(e.g. \( \lambda(d,\text{target}_a,\text{node},\text{target}_b \text{ node}) = \lambda(d,\text{target}_b,\text{node},\text{target}_a \text{ node}) \)) and would be zero if on the same target. The computation time \( \mu \) for each node on \( \text{target}_a \) and \( \text{target}_b \) is expressed as a tuple(e.g. (6, 2) for node \( n \) means \( \mu(n,\text{target}_a) = 6 \) and \( \mu(n,\text{target}_b) = 2 \)) and \( \lambda \) is a non-zero value for each data object if data transfer of different targets is needed. Let \( \lambda \) value for \( \lambda[0] \sim \lambda[7] = \{3, 3, 1, 2, 2, 3, 2, 5\} \) and \( \mu \) value for \( \mu[0] \sim \mu[6] = \{6, 2, 4, 1, 6, 5, 4, 6\} \). At first each computation node \( n \) is initially assigned to a target that minimizes \( \mu(n,p) \), so we get the target selection of each node as \( \{B, B, B, B, A, A, A, B\} \). We then compute the critical path, the exit node of this graph is \( n[6] \). The rank of \( n[6] \) is \( \text{rank}(n[6]) = \mu(n[6], B) + \lambda(\text{results}, n[6], A) = 10 + 5 = 15 \). Then we can recursively traverse upward and get the rank of each node is \( \{32 = 40, 1 + 1 + \max(29, 30) = 32, 5 + 2 + 22 = 29, 5 + 2 + 23 = 30, 4 + 3 + 15 = 22, 6 + 2 + 15 = 23, 10 + 5 = 15\} \). The critical path is \( \{n[0], n[1], n[3], n[5], n[6]\} \) and the makespan of this graph is 40.

In \textit{clusterCP} procedure, we pass the critical path into it. The first unvisited data object on the critical path causing maximum \( \lambda \) value is \( d[0] \), because \( \lambda(d[0], n[0], n[1]) = 3 \) is maximum. Writer and reader node of \( d[0] \) are \( n[0] \) and \( n[1] \) with same target choice \( B \), so at line 4 to line 6 of \textit{clusterCP} procedure would try to cluster them into cluster. We assume the window size rule here is adjacency nodes in the same cluster can not more than two nodes. So this windows is valid and we can get \( C(B) = \{n[0], n[1]\} \).

Second loop time, we get \( d[3] \) and \( d[5] \) with same \( \lambda \) value. We choose \( d[3] \) according to the rank non-increasing order. Writer \( n[3] \) and reader \( n[5] \) prefer different target, so at line 8 to line 22 will try to change them to be on same target. The gain is \( \lambda(d[3], n[3], n[5]) = 2 \) and cost is \( \min(6 - 5, 8 - 6) = 1 \). Gain is greater than cost and windows size for writer and reader in cluster \( A \) is valid. We get a new cluster \( C(A) = \{n[3], n[5]\} \). For the last remaining nodes \( d[3] \) and \( d[1] \), because visited nodes appear in writer or reader nodes would bring \( \infty \) cost and for \( d[5] \) change reader cost is 5, gain is smaller than cost. Two clusters are obtained on critical path.

In \textit{clusterG} procedure, we cluster remaining nodes \( n[2] \) and \( n[4] \) on the graph. The \( \text{improve}(n[2]) \) is 1.2 and \( \text{improve}(n[4]) \) is 1.5, so we choose \( n[4] \) as an unvisited computation node first. At line 3 to line 8, the \textit{diff\_count} will count 1 because \( n[6] \) is visited and it is on a different target \( B \) versus \( n[4] \) on target \( A \). So at line 12 to line 18, we try to change the target running \( n[4] \). The gain is \( \lambda(d[4], n[4], n[6]) = 5 \) and the cost is \( \mu(n[4], B) - \mu(n[4], A) = 6 - 4 = 2 \). Gain greater than cost so we will change \( n[4] \) to select target \( B \) and get a cluster \( C(B) = \{n[4], n[6]\} \). For node \( n[2] \), the \textit{diff\_count} count is zero because \( n[1] \) and \( n[4] \) are on the same target(\( B \)) as \( n[2] \) and \( n[3] \) are read edge to \( d[1] \), too. The algorithm finally gets \( C(A) = \{n[3], n[5]\} \) and \( C(B) = \{(n[0], n[1]), (n[4], n[6])\} \) in node coarsen algorithm.

### 3.2 Node scheduling algorithm

The node scheduling algorithm dispatches nodes according to non-increasing order of rank values. This algorithm will execute in graph processing phase(\textit{vxProcessGraph(graph)}) After the node coarsen algorithm clusters some of the computation nodes into clusters, this runtime scheduling algorithm directly dispatch nodes with clusters and makes targets decision for non-clustered nodes by considering the runtime environment. We now present our node scheduling algorithm in Algorithm 2.

Initially, we compute the rank value as algorithm 1 at line 1 to line 3. Then we sort the rank value in non-increasing order and take the first one in each loop cycle. If the node in assign to cluster, we dispatch the node directly at line 6. On the contrary, we try to evaluate the gain and cost on prefer target and most free target. Node can be dispatched to preference target if prefer target is free at line 12. At line 14 to line 23, we try to change node \( n \) to run on most free target. A new rank get at line 16 always higher because of the \( \mu \) cost. But if this cost is worth versus wait time of prefer target, we should change the target of \( n \). Two decision at line 17 to line 18 and at line 20 to line 22 are either change the target of \( n \) and recompute rank or keep original target selection.

We use the result of previous node coarsen algorithm as sample input. Because the cluster relationship.
Algorithm 2: Scheduling(G, P, C)

Input: OpenVX graph: G, an OpenVX target sequence: P, nodes for targets C

Output: computation node n being dispatched to OpenVX target

1. Compute rank value for all nodes by traversing graph upward, starting from sink nodes;
2. Sort the nodes in the graph by non-increasing order of rank values;
3. while there are unscheduled nodes in the graph do
   4. Select the first node n;
   5. if node n ∈ C(p) then
      6. Dispatch all nodes in cluster to target p;
   7. else
      8. preferP ← n.target;
      9. rank on P ← n.rank;
     10. mostFreeP ← the least loading target in P;
     11. if preferP = mostFreeP then
         12. Dispatch the node n to target preferP;
     13. else
         14. Assign n.target to mostFreeP;
         15. Re-Compute rank value for all nodes;
         16. if rank on P − n.rank < wait_time then
             17. Dispatch the node n to target mostFreeP;
             18. Re-Sort the nodes in the graph by non-increasing order of rank values;
         19. else
             20. Assign n.target back to preferP;
             21. Dispatch the node n to target preferP;
             22. Re-Compute rank value for all nodes;
     23. end
   24. end
25. end

\n
\[ n[0], n[1], n[4] \] and \[ n[6] \] directly dispatch to target B. Node \[ n[3] \] and \[ n[5] \] directly dispatch to target A. Node \[ n[2] \] is not in cluster, the original rank of \[ n[2] \] is \[ \mu(n[2], B) + \mu(n[4], B) + \mu(n[6], B) + \lambda(results, B, A) = 5 + 6 + 10 + 5 = 26 \]. If most free target is not B, we try to change node \[ n[2] \] to use target A. A new rank is 29 because the cost of \[ \lambda(d[2], n[2], n[4]) \] and \[ \mu(n[2], A) - \mu(n[2], B) \]. So if wait time for target B is more than 29 = 26 = 3 units, this node \[ n \] would be dispatched to target A.

4. Experimental Results

We implement our coarsen-scheduling algorithm within the OpenVX framework released by Khronos Group. Our experiment environment is on AMD A10-7850k APU with 4 core CPUs and a Spectre GPU inside. The experimental OpenVX framework is built on Ubuntu 14.04.2 LTS with OpenCL runtime support and compile by gcc 4.8.2 compiler with -O3 optimization option. In addition, a profile procedure profiles each node at the first time execution of this framework. The profile information includes the average time a node spent on computation and the average data transfer time for each parameter over 100 execution on fixed size data objects (here is 640*480). Every time the framework initialize its context, these profile informations will be loaded. We exclude the nodes do not support both C and OpenCL target in this experiment and list the profiling results in Table 1.

<table>
<thead>
<tr>
<th>kernel name</th>
<th>target</th>
<th>compute(ms)</th>
<th>transfer(ms)</th>
<th>total(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>table lookup</td>
<td>c-model</td>
<td>2.179</td>
<td>0.085</td>
<td>2.179</td>
</tr>
<tr>
<td>and</td>
<td>c-model</td>
<td>2.957</td>
<td>0.223</td>
<td>2.957</td>
</tr>
<tr>
<td>or</td>
<td>c-model</td>
<td>2.903</td>
<td>0.230</td>
<td>2.903</td>
</tr>
<tr>
<td>xor</td>
<td>c-model</td>
<td>2.980</td>
<td>0.227</td>
<td>2.980</td>
</tr>
<tr>
<td>not</td>
<td>c-model</td>
<td>1.952</td>
<td>0.157</td>
<td>1.952</td>
</tr>
<tr>
<td>histogram</td>
<td>c-model</td>
<td>1.871</td>
<td>0.155</td>
<td>1.871</td>
</tr>
<tr>
<td>box3x3</td>
<td>c-model</td>
<td>11.846</td>
<td>0.198</td>
<td>11.846</td>
</tr>
<tr>
<td>gaussian3x3</td>
<td>c-model</td>
<td>11.855</td>
<td>0.195</td>
<td>11.855</td>
</tr>
</tbody>
</table>

Table 1: Profiling results of OpenVX kernels

As the profiling results shown in the above table, we can observe that all the nodes on OpenCL target (pc.opencl) consume lower processing time than the nodes compute on CPU target (c-model) (even take account of both computation time and data transfer time). Even the slowest kernel on GPU (the 'or' kernel with 0.280 ms) is 6.68 times faster than the fastest kernel on CPU (the 'histogram' kernel with 1.871 ms). However, there will be more and more accelerating targets and complex applications adopting the OpenVX framework. So the nodes with smaller time difference on different targets will appear in the near future. In the case that different accelerators with similar computation ability, nodes may have less different processing time on different targets for individual execution. For example, a complex computation algorithm may consume similar processing time on GPU and DSP. In this case, our proposed algorithm can schedule these nodes for a good arrangement.

In this experimental framework, the node characteristics make our coarsen-scheduling algorithm have similar results as the target priority design in the original sample implementation. The target priority design dispatch nodes to OpenCL target first, then OpenMP and C model. Nodes have OpenCL target support will choose it and get good performance result because most of the nodes have good result on OpenCL target. But there still exist some cases that will benefit from
Table 2: Time results with and without memory locality

<table>
<thead>
<tr>
<th>node</th>
<th>memory locality</th>
<th>compute(ms)</th>
<th>transfer(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>YES</td>
<td>1.169</td>
<td>0.538</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>1.156</td>
<td>0.557</td>
</tr>
<tr>
<td>xor</td>
<td>YES</td>
<td>0.445</td>
<td>0.038</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>0.578</td>
<td>0.025</td>
</tr>
<tr>
<td>or</td>
<td>YES</td>
<td>0.403</td>
<td>0.080</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>0.428</td>
<td>0.015</td>
</tr>
<tr>
<td>and</td>
<td>YES</td>
<td>0.173</td>
<td>0.008</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>0.191</td>
<td>0.023</td>
</tr>
</tbody>
</table>

5. Conclusions

In this paper, we proposed a two phase scheduling method to address the scheduling issues of OpenVX task graph. The proposed two phase scheduling method considered both memory locality and system throughput to schedule OpenVX Programs on heterogeneous multi-core systems. The first phase of our algorithm performs coarsen schemes to cluster nodes together and the second phase schedule nodes into different targets. The result of our experiment shows that our algorithm works well in scheduling OpenVX programs on heterogeneous environments and analyzed some cases this algorithm will be benefit.

Acknowledgment

This work is conducted under the "The core technologies of smart handheld devices project" of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China and is also in part by the MOST (Ministry of Science and Technology) project.

References