A Memory-Bounded Best-First Beam Search and Its Application to Scheduling Halide Programs

Chao Gao,1 Jingwei Chen,1 Tong Mo,1 Tanvir Sajed,1 Shangling Jui,2 Min Qin,2 Laiyuan Gong,2 Wei Lu1

1Huawei Canada Research Center
2Huawei Technologies, China
{chao.gao4, tong.mo, jingwei.chen, tanvir.sajed, jui.shangling}@huawei.com
{gonglaiyuan,qinmin5,robin.luwei}@hisilicon.com

Abstract
Beam search is a popular algorithm for solving real-world problems — especially where search space is an enormously large tree but real-time solutions are most preferred. We present a memory-bounded best-first beam search (MB2FBS), which can be viewed as an improved and generalized version of standard beam search in trees. The algorithm takes three parameters — in contrast to the singular parameter beam size in standard beam search. We discuss how to recover standard beam search and how to realize other search behaviour by setting these three parameters correspondingly. In particular, we show that the principal version of MB2FBS can be thought as an algorithm whose search expense is similar or upper bounded by beam search of certain beam size; however it often finds better solutions as it decides the number of nodes to be searched each depth dynamically with respect cost landscape. We apply our algorithm for tensor program auto-scheduling in Halide, an important industrial problem that uses tree search for optimizing tensor program executions. We show that the principal variants of MB2FBS deliver better empirical results than the highly optimized beam search counterpart. Most importantly, it finds superior schedules while no more computation cost is used for search, which is highly desirable for real-time program compilation and optimization.

Introduction
Beam search Medress et al. is a heuristic search paradigm that has been extensively used in solving problems from many areas of artificial intelligence, such as vision (Roy and Todorovic 2014), job scheduling (Birgin, Ferreira, and Ronconi 2015), planning (Karbowska-Chilinska et al. 2019) and natural language processing (Meister, Vieira, and Cotterell 2020) — in these domains, beam search is preferred presumably because the computer memory is typically insufficient for searching the whole state space of a problem instance, meanwhile a quick and sub-optimal solution is more desirable than an optimal but slow one.

Essentially, beam search achieves fast solving by searching only on a promising subset of nodes from the whole state-space , assuming there is an external input for heuristic pruning. Indeed, there have been two popular ways of characterizing beam search (Bisiani 1992): the general form describes beam search as an abstract algorithm where search only has to be conducted on a subset of states (called beam) induced by a set of heuristic pruning rule — the search strategy used to explore can either be depth-first, breadth-first or best-first; the another simplified and more specific form of beam search regards the algorithm as a truncated version of breadth-first search — given a beam size parameter and a priority function for comparing states, the search needs only to consider top nodes each layer. Evidently, the second form can be seen as a specific realization of the first version. However, in practice, the simplified form is more popularly used such that it is often referred as the standard version of beam search (Zhou and Hansen 2005).

Figure 1 illustrates general and standard breadth-first based beam search.

Figure 1: Illustration of general and standard beam search.

The principal drawback of beam search is that the algorithm by design does not guarantee completeness. To address this issue while maintaining its advantage of being able to deliver solutions using short time and small computer memory, subsequent algorithms have been developed to embed beam search into another systematic searching such that completeness can be achieved. To name a few, (Zhang...
1998) proposed any time complete beam search that iteratively calls beam search with relaxed pruning rules so that convergence to optimal can be assured; (Zhou and Hansen 2005) proposed beam-stack search that conducts a sequence of standard beam search using depth-first search; (Furcy and Koenig 2005) devised limited discrepancy beam search that performs limited discrepancy search (Harvey and Ginsberg 1995; Korf 1996) upon standard beam search.

On the other hand, standard beam search might be viewed as an extension of greedy search. Like greedy search, beam search selects successors to expand consecutively until a solution is reached; unlike greedy search, at each time, a set of successor states, rather than a single state, are selected to proceed. This continual pushing forward ensures fast execution of beam search; however, we argue that this scheme is also the source of deficiency of standard beam search because that it neglects the comparison of states across different depths. Instead of implicitly limiting the memory usage through heuristic pruning rules, another line of research has been pursued by giving an explicit memory-bound to the search. The memory-bounded A* (MA*) algorithms (Chakrabarti et al. 1989; Russell 1992; Kaindl and Khorsand 1994; Zhou and Hansen 2002) work similar to A* (Hart, Nilsson, and Raphael 1968) except that it limits the maximum number of searching nodes in OPEN and CLOSED list, and whenever the memory is full, a worst-seeing node in OPEN is released and its parent would be moved from CLOSED to OPEN if necessary. These algorithms guarantee that the search would be able to find an optimal solution as long as the memory is sufficient to store the least sized optimal solution path. In fact, MA* algorithms sit in between IDA* (Korf 1985) / RBFS (Korf 1993) and A*, in the sense that the former uses the least memory possible at the expense of extensive re-expansion, while the latter requires a memory as large as the whole state-space but avoids possible re-expansion at its extreme.

Motivated by developing better incomplete but fast algorithms for scheduling Halide programs (Ragan-Kelley et al. 2013; Adams et al. 2019)1, in this paper, we address the deficiency of beam search using best-first search. We present a generalization of standard beam search from the perspective of memory-bounded and k—best-first search (KBFS) (Felner, Kraus, and Korf 2003). We illustrate the usefulness of the new search algorithm with a synthetic tree environment whose cost landscape can be manipulated for specific interests of investigation. Finally, experiments in Halide demonstrate that our new algorithm surpasses the state-of-the-art beam-search auto-scheduler without incurring more computation budgets.

### Preliminaries

To ease presentation and analysis, we assume the state-space \( \mathcal{S} \) to be searched can be regarded as a tree, such that the following terminologies and assumptions are used:

- There is a singular root node called start.
- \( \forall s \in \mathcal{S}, \text{ch}(s) \subset \mathcal{S} \) represents the set of children below \( s \), while \( s \neq \text{start}, \text{pa}(s) \in \mathcal{S} \) stands for the parent of \( s \).
- A node \( s \in \mathcal{S} \) is called terminal if \( \text{ch}(s) = \emptyset \).
- Following A* algorithms, \( \forall s \in \mathcal{S}, g(s) \) is the path cost \( \text{start} \rightarrow s \), while \( h(s) \) estimates the cost from \( s \) to goal. \( h(s) \) is called admissible if \( h(s) \leq h^*(s) \), i.e., \( h(s) \) underestimates the optimal cost from \( s \) to goal. \( f(s) \triangleq g(s) + h(s) \).
- All costs are non-negative.

Starting from the root node, best-first search algorithms construct a search tree \( \mathcal{T} \) iteratively through a sequence of node expansions. Formally, \( \forall s \in \mathcal{T}, s \) is called closed if \( \text{ch}(s) \subset \mathcal{T} \) otherwise open, and suppose they are respectively stored in sets CLOSED and OPEN, then \( \mathcal{T} = \text{CLOSED} \cup \text{OPEN} \). At each iteration of best-first search, a node with the lowest (thus best) \( f \) cost in OPEN is selected for expansion, and when \( h \) is admissible everywhere, it is known that the best-first search becomes A* (Pearl 1984). A best-first search is called memory-bounded if there is a maximum storage limit for CLOSED \( \cup \) OPEN such that whenever the limit is reached, node deletion has to be performed on OPEN. In this paper, we do not store CLOSED, since our major concern is not completeness (i.e., backtracking from CLOSED as SMA* (Russell 1992) is not needed).

In standard beam search, the parameter \( b \) is called beam size; for each search depth, states are sorted in ascending order w.r.t \( f \), such that top \( b \) states are selected for searching.

### Unified Framework for Memory-Bounded Best-First Beam Search

In this section, we present our new algorithm framework for memory-bounded best-first beam search.

### A Simple Memory-Bounded Best-First Search

We first discuss a simple algorithm that executes in the same spirit as best-first search except that a bounded memory is used. The algorithm is sketched in Algorithm 1. The priority queue can be implemented using min-max priority queue (Atkinson et al. 1986), such that whenever the \( Q \) is exceeding its bound, a worst state w.r.t \( p \) can be deleted. A natural choice for \( \rho \) is the \( f \) cost, as in A*, and it is clear that when \( Q \) is given a max size larger than the total number of possible states, Algorithm 1 becomes A*, assuming that the heuristic function \( h \) is admissible — in this case, without losing optimality. Algorithm 1 can stop early by returning its first encountered terminal. It is also clear that Algorithm 1 expands \( O(d_{max} M) \) nodes, since at each tree depth, it would never visit more than \( M \) nodes due to the memory bound.

#### Batched Expansion Equals Beam Search

We see that Algorithm 1 updates the bounded memory after every node expansion. We now consider the case that the insertion to \( Q \) is delayed until a batch of nodes have been expanded — we refer this scheme as batched expansion. That is, in Algorithm 2, we modify the \( s \leftarrow Q . \text{pop}() \Rightarrow S_k \leftarrow Q . \text{pop}(k) \) where \( k \) is batch size, and \( C \leftarrow \text{expand}(s) \Rightarrow C \leftarrow \text{expand}(S_k) \).
Figure 2: Memory-bounded best-first search batch 2 expansion vs one-by-one expansion. Each number indicates a $f$-cost, thus lower cost implies higher priority. For batch 1 expansion, first, node 2 is expanded, then $Q \leftarrow \{6, 3\}$, then 3 gets expanded, $Q \leftarrow \{6, 4\}$. For batch 2 expansion, 2 and 3 are expanded at the same time, then $Q \leftarrow \{4, 5\}$.

To see the effect of batched expansion, consider a case illustrated in Figure 2, where if $Q$ has a bounded size of 2, one by one expansion would update $Q \leftarrow \{4, 6\}$ whereas batch of 2 expansion results $Q \leftarrow \{4, 5\}$. Thus, we have the following observation.

**Lemma 1.** If a batch of $k$ nodes expansion is applied, then, in the bounded memory $Q$, there must be $k$ states being pushed forward in the next iteration, i.e., these $k$ states in $Q$ will be replaced by one-depth deeper states.

**Theorem 1.** For simple memory-bounded best-first search, if we limit the priority queue size to $\beta$, and use a batch expansion of $\beta$ states each time, then, the algorithm is equivalent to standard beam search.

**MB2FBS: Mixing Push Forward and Onward**

While Algorithm 1 makes sure that whenever a batch of $k$ nodes is expanded, there is a push forward by $k$ in $Q$, the deficiency of this scheme is that it neglects the other states at lower depth that was not considered — they may become promising again upon the removal of just expanded states.

Consider a case illustrated in Figure 3. Suppose at certain time the $Q$ contains two nodes of costs 2 and 3, and they have a sibling not in $Q$ with a cost of 4. If we expand a batch of two nodes, then update the $Q$ by the newly created child nodes, then $Q \leftarrow \{5, 6\}$; however, if we also re-examine the sibling of 2 and 3, then $Q \leftarrow \{4, 5\}$ — this is arguably a better strategy since nodes with smaller cost estimates are retained. To distinguish, we name this strategy of reconsideration as pushing onward.

It is clear that pushing forward guarantees fast execution to terminal states, while pushing onward makes sure that the most promising states are pursued. In order to subsume the advantages of both strategies, we devise a new search algorithm where the extent of each strategy can be swiftly configured by appropriate parameterization. Specifically, we introduce two parameters, $\beta_1$ and $\beta_2$, respectively represent the number of states being pushed forward and onward at each iteration. The full procedure is depicted in Algorithm 2.

Same as Algorithm 1, Algorithm 2 operates on a bounded memory parameterized by $M$, which we assume should be larger than or equal to $\beta_1 + \beta_2$. In fact, any $M > \beta_1 + \beta_2$ is equivalent to $M = \beta_1 + \beta_2$ — for each iteration in Algorithm 2, sorting the nodes in $Q$ by their priority, then the first top $\beta_1$ nodes from $Q$ are selected for expansion and newly generated child nodes are inserted to $U$, then next top $\beta_2$ nodes from $Q$ are directly moved from $Q$ to $U$. In other words, even $Q$ contains more than $\beta_1 + \beta_2$ nodes, these nodes not in top $\beta_1 + \beta_2$ will simply be ignored. Thus, in practice, setting $M = \beta_1 + \beta_2$ suffices. Assuming $M = \beta_1 + \beta_2$, we can now bound the number of node expansion of Algorithm 2 in Proposition 1.

**Proposition 1.** For Algorithm 2 with parameters $\beta_1 > 0, \beta_2 > 0, \beta = \infty$, and $M = \beta_2 + \beta_1$, then the total number of node expansion is bounded by $O((\beta_1 [\frac{\beta_2}{\beta_1}])^{d_{\text{max}}(d_{\text{max}}+1)})$

**Proof.** The algorithm begins with all states in $Q$ with depth 0, and ends with all states at maximum depth. In other words, let $x$ be the minimum state depth in $Q$, then it starts with $x = 0$, and ends with $x = d_{\text{max}}$. We know that for each iteration of search, $\beta_1$ states are expanded, thus that $x$ either remains unchanged or $x \leftarrow x + 1$. Therefore, the slowest possible growth for $x$ is that the same $\beta_2$ states being retained for the longest time.

Since $M = \beta_1 + \beta_2$ and $\beta_1 \geq \beta_2$, then it takes at most $d_{\text{max}}$ iterations for the same $\beta_2$ states being retained in $Q$. Formally, denoting the time complexity as $T(d_{\text{max}})$, we can express the worst case recursively as follows:

$$T(d_{\text{max}}) \leq \beta_1 d_{\text{max}} + T(d_{\text{max}} - 1)$$

$$= \beta_1 d_{\text{max}} + \beta_1 (d_{\text{max}} - 1) T(d_{\text{max}} - 2)$$

$$= \beta_1 (d_{\text{max}} + (d_{\text{max}} - 1) + \ldots + 1)$$

$$= \beta_1 d_{\text{max}}(d_{\text{max}} + 1)/2$$

For the other case that $\beta_1 < \beta_2$ ($M = \beta_1 + \beta_2$), only a
factor is needed for the recursion:

\[
T(d_{\text{max}}) \leq \beta_1 d_{\text{max}} \left(\frac{\beta_2}{\beta_1}\right) + T(d_{\text{max}} - 1) \\
= \beta_1 \left(\frac{\beta_2}{\beta_1}\right)d_{\text{max}} (d_{\text{max}} + 1)/2 \\
= \mathcal{O}(\beta_1 \frac{\beta_2}{\beta_1} d_{\text{max}}^2 + 1) \\
\]

Thus, the claim holds.

We can draw the following observation from Proposition 1: for a given search depth, multiple batches of \(\beta_1\) states might be eventually expanded — in contrast to standard beam search, where a constant of \(b\) states will be expanded. To address this discrepancy and let Algorithm 2 matches the efficiency of beam search, we then introduce counting scheme to control the number of states being expanded at each depth — we can force the algorithm not to expand more than \(\beta (\beta < \infty)\) nodes. We name Algorithm 2 with \(\beta < \infty\) as the controlled version of MB2FBS. We note that, for the controlled version, \(M\) has to be at least \(\beta d_{\text{max}}\), otherwise the algorithm might be unable to return a solution. That is, if \(Q\) and \(U\) are with too small capacity, new nodes created by batch expanding \(\beta_1\) nodes might all get discarded, and if this is repeated multiple rounds till \(\beta_1 + \ldots + \beta_\ell > \beta\), then the algorithm would stop moving forward forever. This phenomenon is also seen in best-first beam search (Meister, Vieira, and Cotterell 2020). In Algorithm 2, we mark how \(\beta\) controls the number of expansion using a vector of \(P\). For the uncontrolled version, these code with \(P\) becomes ineffective because \(\beta = \infty\).

**Remark.** For both uncontrolled and controlled versions of MB2FBS, in practical use, we can skip to set the parameter \(M\) (i.e., let \(Q\) and \(U\) unbounded by assuming \(M = \infty\)), because the memory-usage of \(Q\) and \(U\) are implicitly implied by the setting of \(\beta_1, \beta_2\) and \(\beta\). If one wants to set \(M\), it must be at least either \(\beta_1 + \beta_2\) (uncontrolled) or \(\beta d_{\text{max}}\) (controlled).

It is possible to configure the parameters, i.e., \(\beta_1, \beta_2, \beta\), swiftly for realizing different search behaviors. In Table 1 we summarize how to set these parameters for 8 typical variants.

---

**Algorithm 2: Memory-bounded best-first beam search**

```plaintext
Procedure MB2FBS (start, \(\rho_1, \beta_1, \beta_2, M, \beta = \infty\)):
/* \(\rho\) is priority function such that \(\rho(x) \bigotimes \rho(y)\) implies \(x\) has higher priority than \(y\) */
\(Q \leftarrow \text{priority_queue} < \rho > (\text{max_size} = M)\)
\(POPS \leftarrow \text{vector}(d_{\text{max}})\)
\(r \leftarrow \text{null}\)
\(\text{while not Q.empty()} \) do
\(U \leftarrow \text{priority_queue} < \rho > (\text{max_size} = M)\)
\(\text{for } i \in \{1, 2, \ldots, \beta_1 + \beta_2\} \wedge \text{not Q.empty()} \) do
\(s \leftarrow Q\).pop()\)
\(\text{if } s\).terminal() \) then
\(\text{if } r = \text{null} \text{ or } \rho(s) \preceq \rho(r) \) then
\(r \leftarrow s\)
\(\text{continue} \)
\(\text{if } i > \beta_1 \) then
\(U\).push(s)\)
\(\text{continue} \)
\(\text{if } \text{POPS}[s\).depth] \geq \beta \) then
\(C \leftarrow \text{expand(s)}\)
\(\text{POPS}[s\).depth] \leftarrow \text{POPS}[s\).depth] + 1\)
\(\text{foreach } c \in C \) do
\(U\).push(c)\)
\(Q \leftarrow C\)
\(\text{return } r\)
```

---

- If \(M\) is assumed to be as large as the whole tree, and \(\beta_2 = 0, \beta_1 = \beta = \infty\), then Algorithm 2 becomes an instance of breadth-first search where the visit of states in the same layer is prioritized by priority function \(\rho\). If \(\beta_1 = 1, \beta_2 = \infty\), then the algorithm becomes an equivalent of best-first search — more generally, KBFS (Feller, Kraus, and Korf 2003), if \(\beta_1 > 1\). The best-first search becomes an implementation of A* when the priority function is defined using an admissible heuristic, e.g., \(f = g + h\) where \(h\) is admissible.

- If \(\beta_1 = 1, \beta_2 = 0\), Algorithm 2 becomes greedy search. If \(\beta_1 = 1, \beta_2 = \infty\), then the algorithm becomes best-first search except that \(M\) is bounded, akin to Algorithm 1.

- Suppose \(\beta\) is given, and we set \(\beta_2 = 0, \beta_1 = \beta\), then Algorithm 2 works the same as standard beam search with beam size \(b = \beta\). If \(\beta_1 = 1, \beta_2 = \beta - 1\), then the algorithm becomes an equivalent of the best-first beam-search depicted by (Meister, Vieira, and Cotterell 2020). These two algorithms are guaranteed to find the same solution if \(\rho\) is defined upon admissible heuristic.

- The last row of Table 1 contains the two variants of our new algorithm MB2FBS. The controlled version ensures that it expands no more than \(\beta d_{\text{max}}\) nodes, which is the number of nodes get expanded by beam search with \(b = \beta\).
haustive search is infeasible (Ragan-Kelley 2014; Li et al. 2020). A pipeline usually consists of multiple stages of computation; at each stage, two successive decisions need to be made respectively for intra-stage ordering and cross-stage granularity (Li et al. 2018; Adams et al. 2019). Given a pipeline, a complete low-level implementation is produced upon the decisions at all stages in the corresponding DAG have been fully made, and the process of such decision-making is referred to as Halide scheduling.

Figure 4 illustrate the auto-scheduling process for a toy pipeline — it contains three functional stages, so there are 6 decision depths. In practice, a real-world pipeline usually contains tens of functional stages, and each decision node may contain hundreds of actions (e.g., $100^{20}$), such that exhaustive search is infeasible (Ragan-Kelley 2014; Li et al. 2018). The state-of-the-art X86 CPU auto-scheduler finds schedules using a variants of beam search along with a cost model (Adams et al. 2019).

The major innovations of auto-scheduler (Adams et al. 2019) can be summarized as follows. First, it uses a pre-trained cost model to guide the search. The cost model hybridizes symbolic analysis and a neural network (LeCun, Bengio, and Hinton 2015) — for each node $s$ in search tree, suppose $N(s)$ decisions have been made in $s$, then it calculates a vector of schedule features of size $N(s)$, then estimates a cost. It is generally true that those costs at depth $d+1$ are greater than depth $d$. However, this monotonic increasing property is not truly guaranteed because of the use of neural network, implying that for heuristic search, the available heuristic is not admissible. These properties further imply that algorithms that utilize admissible pruning, e.g., IDA* (Korf 1985), are infeasible for the domain. Indeed, to enable large space search, an incomplete beam search is used in (Adams et al. 2019) ². More specifically, it develops a multi-pass coarse-to-fine beam search. That is, the auto-scheduler runs beam search multiple times, where each run uses a different hash function for pruning actions; thus, each pass defines a different coarse region for searching, and beam search pass $i+1$ uses a hash set from pass $i$ to further refine the region being searched. We note that, although local search algorithms, e.g., simulated annealing (Chen et al. 2018b) and genetic algorithms (Gao et al. 2021; Zheng et al. 2020), have also been used by other tensor compiler optimization frameworks (Chen et al. 2018a). These algorithms are not suited for the Halide compiler because currently it does not have direct and structured representation of a schedule object. The search space is a decision-tree, where one can only construct a schedule by sequentially making decisions. Therefore, schedules only occur at the last depth of the decision-tree. In other words, the schedules are just terminal states — no explicit structures are exposed for the

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$β_1 &gt; 0, β_2 &gt; 0$</td>
<td>$β \leftarrow \infty, β_1 &gt; 0, β_2 &gt; 0$</td>
<td>$β_1 \leftarrow β, β_2 \leftarrow β - 1$</td>
<td>$β_1 \leftarrow 1, β_2 \leftarrow \infty, β \leftarrow 1$</td>
<td>$β_1 \leftarrow 1, β_2 \leftarrow \infty$</td>
<td>$β_1 \leftarrow \infty, β_2 \leftarrow \infty$</td>
<td>$β_1 \leftarrow 1, β_2 \leftarrow \infty, β \leftarrow \infty$</td>
<td>$M = m_{max}d_{max}$</td>
</tr>
<tr>
<td>$M = d_{max}^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M = m_{max}d_{max}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M = b_{max}d_{max}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Possible configuration of MB2FBS for other well-known heuristic search algorithms.

Experiments

We now proceed to deploy MB2FBS for auto-scheduling in Halide (Ragan-Kelley et al. 2013). We begin by introducing the problem, how the scheduling can be modelled as a tree, and what search strategies are used by the state-of-the-art auto-scheduler for this domain.

Tensor Program Scheduling in Halide

Halide is a domain specific language for high-level specifying computer vision pipelines (e.g., Gaussian blur) as well as a tool that can automatically find high-performance low-level hardware-dependent implementations of the pipeline during compilation. It represents the pipelines as directed acyclic graphs (DAG), where each graph node is called a stage. A pipeline usually consists of multiple stages of computation; at each stage, two successive decisions need to be made respectively for intra-stage ordering and cross-stage granularity (Li et al. 2018; Adams et al. 2019). Given a pipeline, a complete low-level implementation is produced upon the decisions at all stages in the corresponding DAG have been fully made, and the process of such decision-making is referred to as Halide scheduling.

Figure 4 illustrates the auto-scheduling process for a toy pipeline — it contains three functional stages, so there are 6 decision depths. In practice, a real-world pipeline usually contains tens of functional stages, and each decision node may contain hundreds of actions (e.g., $100^{20}$), such that exhaustive search is infeasible (Ragan-Kelley 2014; Li et al. 2020). The state-of-the-art X86 CPU auto-scheduler finds schedules using a variants of beam search along with a cost model (Adams et al. 2019).

The major innovations of auto-scheduler (Adams et al. 2019) can be summarized as follows. First, it uses a pre-trained cost model to guide the search. The cost model hybridizes symbolic analysis and a neural network (LeCun, Bengio, and Hinton 2015) — for each node $s$ in search tree, suppose $N(s)$ decisions have been made in $s$, then it calculates a vector of schedule features of size $N(s)$, then estimates a cost. It is generally true that those costs at depth $d+1$ are greater than depth $d$. However, this monotonic increasing property is not truly guaranteed because of the use of neural network, implying that for heuristic search, the available heuristic is not admissible. These properties further imply that algorithms that utilize admissible pruning, e.g., IDA* (Korf 1985), are infeasible for the domain. Indeed, to enable large space search, an incomplete beam search is used in (Adams et al. 2019) ². More specifically, it develops a multi-pass coarse-to-fine beam search. That is, the auto-scheduler runs beam search multiple times, where each run uses a different hash function for pruning actions; thus, each pass defines a different coarse region for searching, and beam search pass $i+1$ uses a hash set from pass $i$ to further refine the region being searched. We note that, although local search algorithms, e.g., simulated annealing (Chen et al. 2018b) and genetic algorithms (Gao et al. 2021; Zheng et al. 2020), have also been used by other tensor compiler optimization frameworks (Chen et al. 2018a). These algorithms are not suited for the Halide compiler because currently it does not have direct and structured representation of a schedule object. The search space is a decision-tree, where one can only construct a schedule by sequentially making decisions. Therefore, schedules only occur at the last depth of the decision-tree. In other words, the schedules are just terminal states — no explicit structures are exposed for the
search algorithms to define concepts such as neighboring schedules and schedule transformations.

**MB2FBS for Halide**

To have a fair comparison, we implement MB2FBS directly on the codebase of (Adams et al. 2019)\(^1\). We use the same pre-trained cost model for X86 CPU throughout our experimentation. In addition, since the cost model in Halide tends to give less comparable cost estimates for states in different depths, we define a new priority function that combines depth and cost.

For a given pipeline, the maximum search depth \(d_{\text{max}}\) is known prior to search, and suppose the smallest known solution cost is \(X\). For state \(s\), let \(s_{\text{cost}}\) be the cost model estimate for \(s\). To balance the influence of \(s_{\text{depth}}\) and \(s_{\text{cost}}\), we compute the priority of \(s\) as follows:

\[
\rho(s) \triangleq \frac{X - s_{\text{cost}}}{D_{\text{max}} - s_{\text{depth}}}.
\]

To embed MB2FBS into the multi-pass search framework as (Adams et al. 2019), we set \(D\) and \(X\) as follows:

- We set \(D_{\text{max}} \leftarrow i \cdot d_{\text{max}} + 1\), where \(i\) is pass index starting from 1.
- In first run \(X \leftarrow \infty\)\(^4\), in other runs, \(X\) set to the smallest known solution cost.

Figure 5 demonstrates how \((D_{\text{max}}, X)\) modifies the priorities of states, resulting in a more favourable cross-depth comparison of states in Halide. By varying \((D_{\text{max}}, X)\), MB2FBS is able to search different regions from pass to pass. Note that this priority function has no effect on beam search, because it only compares states in the same depth.

As recommended in Halide, the maximum pass number is set to 5 for both MB2FBS and (Adams et al. 2019). For beam search, the recommended beam size is 32, but we also include results from beam size 64. For MB2FBS, we test two variants: the \(\beta\)-controlled variant with \(\beta_1 = 28, \beta_2 = 4, \beta = 32\), and uncontrolled variant \(\beta_1 = 30, \beta_2 = 2\). We set memory bound to \(\infty\) for all scenarios since we have seen this is safe choice for both controlled and uncontrolled MB2FBS.

Table 2 contains the results of multi-pass beam search and MB2FBS on 15 standard test pipelines in Halide. Among the 15 test cases, the uncontrolled version of MB2FBS is able to find 9 better costs than the best of beam-32 and beam-64, while having 3 cases with slightly larger costs. The results of the controlled version of MB2FBS are similar. Both versions achieved an overall geometric mean smaller than the beam search counterparts. We note that the costs in Table 2 are best terminal costs from cost model, but those costs are typically consistent with real schedule qualities — when translating the terminal states into Halide schedules, after benchmarking, we find that two MB2FBS variants achieved averaged time-costs respectively 21587\(\mu\)s and 21561\(\mu\)s, while the beam-search-32 achieved 27821\(\mu\)s, i.e., a 22% speedup is observed. Note that this improvement is non-trivial in the sense once a pipeline is compiled, it might be used by millions of users everyday, cumulatively saving considerable time and energy.

We further list the node expansions of all algorithms in Table 3. The uncontrolled version used slightly more node expansion than beam search with beam size 32, while for the controlled version, the node expansion is strictly smaller or equal. These results in Table 3 certify that, for the MB2FBS algorithms, the reason that they achieved better performance is mainly due to their incorporation of best-first seeking strategies, not more computation clocks.

**Effectiveness of Modified Priority Function**

To verify the effectiveness of the new priority function, we run additional experiments with the new priority function

---

\(^1\)https://github.com/halide/Halide

\(^4\)In implementation, we use \(10^9\) to represent \(\infty\)
node expansion increased from being unused, i.e., it only prioritizes states by cost estimates.

Table 3: Each node expansion number is averaged over the 5 passes being used for each algorithm. Results obtained on same Intel CPU server.

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Adams2019-b</th>
<th>MB2FBS</th>
<th>MB2FBS-($\beta_1, \beta_2, \beta$) sol. and expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>64</td>
<td>(30, 2, $\infty$) (28, 4, 32)</td>
</tr>
<tr>
<td>bilateral grid</td>
<td>514.0</td>
<td>998.0</td>
<td>570.4</td>
</tr>
<tr>
<td>local laplacian</td>
<td>6594.0</td>
<td>13176.0</td>
<td>6722.2</td>
</tr>
<tr>
<td>nl means</td>
<td>834.0</td>
<td>1660.0</td>
<td>1066.8</td>
</tr>
<tr>
<td>lens blur</td>
<td>4802.0</td>
<td>9572.0</td>
<td>5113.2</td>
</tr>
<tr>
<td>camera pipe</td>
<td>2370.0</td>
<td>4723.0</td>
<td>2627.0</td>
</tr>
<tr>
<td>stencil chain</td>
<td>2242.0</td>
<td>4451.0</td>
<td>2959.0</td>
</tr>
<tr>
<td>harris</td>
<td>770.0</td>
<td>1510.0</td>
<td>923.2</td>
</tr>
<tr>
<td>hist</td>
<td>578.0</td>
<td>1136.0</td>
<td>729.0</td>
</tr>
<tr>
<td>max filter</td>
<td>578.0</td>
<td>1148.0</td>
<td>635.0</td>
</tr>
<tr>
<td>unsharp</td>
<td>578.0</td>
<td>1148.0</td>
<td>639.2</td>
</tr>
<tr>
<td>interpolate</td>
<td>3266.0</td>
<td>6524.0</td>
<td>3365.2</td>
</tr>
<tr>
<td>conv layer</td>
<td>258.0</td>
<td>514.0</td>
<td>286.6</td>
</tr>
<tr>
<td>mat mul</td>
<td>194.0</td>
<td>358.0</td>
<td>223.8</td>
</tr>
<tr>
<td>iir blur</td>
<td>258.0</td>
<td>508.0</td>
<td>357.2</td>
</tr>
<tr>
<td>bgu</td>
<td>1282.0</td>
<td>2556.0</td>
<td>1292.8</td>
</tr>
<tr>
<td>Average</td>
<td>1674.5</td>
<td>3332.1</td>
<td>1834.0</td>
</tr>
<tr>
<td>Avg.Time(s)</td>
<td>106</td>
<td>164</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 4: Minimum solution cost found and node expansion by MB2FBS on 15 standard test cases without using the new priority function.

Table 3: Each node expansion number is averaged over the 5 passes being used for each algorithm. Results obtained on same Intel CPU server.

being unused, i.e., it only prioritizes states by cost estimates.

Table 4 shows the results. Referring to the results in Tables 2, we see that for the uncontrolled version, the overall averaged node expansion increased from 1834.0 to 1992.87, while the geometric mean cost reduced from 12.963 to 12.61. For the controlled version, the averaged node expansion remained almost unchanged, while the geometric mean increased from 12.575 to 12.70. From these results, it is clear that the new priority function helped the controlled version of MB2FBS, while for the uncontrolled version, its practice merit was unnoticed. This is unsurprising because that the major change brought by the new priority function is that it drives the search to visit deeper states preferably — this is a desirable property for the controlled version as it can be viewed as a truncated and early-stopped modification of uncontrolled MB2FBS.

Analysis of the Effect of Mixing Forward and Onward

While the results from previous sections indicate MB2FBS is superior to beam search for solving real-world Halide problems, because of the large problem sizes, it is not easy to conduct extensive ablation studies to fully show what cases MB2FBS is more preferable and what cases not. To clearly show how mixing pushing forward and onward affect search performance, we develop a synthetic tree environment. This is for mimicking Halide problems but the instances can be randomly generated with parameterization. The transparency nature of this environment allows us to conduct experiments for quantifying key strengths of MB2FBS. We can sample instances of different characteristics and scale by appropriate parameterization — for these problems, optimal solutions can also be retrieved and used as a golden reference.

Specifically, we randomly sample synthetic tree as a $t$-ary tree environment for empirically studying different search strategies. Each tree instance is randomly generated such that each internal node has exactly $b_{\text{max}}$ children, and each node $s$ has a randomly sampled cost drawn from $\text{Uniform}(0, \text{depth}(s))$, where $\text{depth}(s)$ is the depth of $s$. The tree has maximum depth of $d_{\text{max}}$. To increase solution difficulty, the cost for each leaf node $l$ be sampled from $\text{Uniform}[\text{depth}(l) + \delta, \text{depth}(l) + \delta^2]$, where $\delta$ is a parameter. The path cost for any leaf node $l$ is defined as the cumulative cost of all nodes along the path from $\text{start}$ to $l$.

Typically, smaller $\delta$ value results random instances easier to solve for greedy-biased strategies. Figure 6 illustrates how $\delta$ affects search performances — when $\delta = 0$, pure greedy algorithm can easily find near optimal or optimal solution cost. Thus, the performances of all algorithms become less distinguishable, and the incorporation of pushing onward does not exhibit visible benefit. However, as $\delta$ increases, the advantage of MB2FBS becomes noticeable — for $\delta = 100$, MB2FBS obtains clearly better accuracy than the other algorithms including beam search with beam size of 32. To ensure the computation used by MB2FBS is comparable to beam search, we set $\beta_1 \leftarrow 30, \beta_2 \leftarrow 2$. We use maximum tree-depth $d_{\text{max}} = 8$ and branching factor $b_{\text{max}} = 4$ because we find this setting produces neither too large nor too trivial instances for experimentation and analysis.

Fixing $\delta = 100$, we then experimentally compare MB2FBS and beam search from various configurations. Figure 7 show the comparison of MB2FBS and beam search with varied beam size $b = \{8, 16, 32, 64, 128, 256\}$. In terms of accuracy, both MB2FBS variants ($b$-controlled and uncontrolled) perform better than beam search overall. For all algorithms, the node expansion grows linearly w.r.t $b$. For the uncontrolled MB2FBS $\beta_1 = b - 2, \beta_2 = 2$, the node expansion of MB2FBS is slightly larger than beam search. For the controlled version, MB2FBS never expands
more nodes than beam search on all cases. The better results of MB2FBS are due to the addition of beam search with a small amount of pushing onward behaviour. When beam size becomes large, i.e., \( b = 256 \), all algorithms achieve the same accuracy of 0.939, presumably because \( \beta_2 = 2 \) is insufficient to bring an effect to MB2FBS. To verify this, we run additional experiments by varying \( \beta_2 = \{2, 8, 16, 32, 64, 128\} \). Table 5 shows the results — e.g., if we increase \( \beta_2 = 32 \), both controlled and uncontrolled MB2FBS are able to achieve an averaged accuracy better than beam search (0.952 and 0.945, respectively), meanwhile the node expansion of controlled MB2FBS (1073.8) is less than beam search.

### Conclusions

We have described a memory-bounded best-first search beam algorithm framework where different search behaviours can be generated by correspondingly setting its hyper-parameters. We have provided an analysis on its properties and demonstrated the usefulness using artificially created tree problems. Most importantly, the advantage of the new algorithm is manifested in its successful deployment to auto-scheduling Halide programs; MB2FBS yielded better solutions without incurring extra computation budgets. While it is indisputable that no search strategies would be universally superior to others in all cases, a practically desirable solution might most likely be the one that is able to strike a balance between the extremes. From this perspective, the extra parameters introduced in MB2FBS should not be viewed as a deficiency, rather, MB2FBS provides a new opportunity for practitioner to swiftly configure these parameters for problem instances of different characteristics.

Machine learning compilers have drawn a considerable attention recently (Li et al. 2020). The auto-scheduling algorithms typically combine search and cost model learning, e.g. (Chen et al. 2018b). Our results in Halide certifies that advanced heuristic search can be an valuable part for compiler optimization — we expect that more AI planning algorithms will be applied for these lines of development.

---

**Figure 6:** Solution accuracy decreases as \( \delta \) increases. For each experiment \( \delta \), 10 random trees are sampled and then ran by each algorithm.

**Figure 7:** For beam search, the x-axis \( b \) represents beam size. For MB2FBS, \( b \) means \( \beta_1 + \beta_2 \), and a constant value of 2 is given for \( \beta_2 \). 10 random trees are generated and tried for each experiment.
References


