Dynamic Programming for Optimal Model Partitioning in Pipeline-Parallel LLM Training

Nicholas Wise Saragih Sumbayak - 13524037
Department of Informatics Engineering
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
nicholasaragih@gmail.com — 13524037@std.stei.itb.ac.id

***Abstract*—Training large language models (LLMs) with billions of parameters requires distributing the work across many accelerators. Pipeline parallelism partitions the chain of transformer layers into K sequential stages, each placed on a different device, and pipelines the execution of a mini-batch split into smaller micro-batches. The makespan of such a pipeline is dominated by its slowest stage, so the way layers are assigned to stages directly determines end-to-end training time. This paper formulates the model-partitioning problem as a dynamic-programming problem and shows that the optimal partition can be computed exactly in $O(KL^2)$ time using the Bellman-style recurrence. A reference Python implementation is described and applied to fifteen synthetic LLM shapes ranging from a 12-layer GPT-2-style stack to a 96-layer 16-stage pipeline. Across every configuration the DP-optimal partition meets or beats a uniform-partition baseline, and the gap widens substantially as layer-cost heterogeneity grows, yielding a 14–16% makespan reduction on the most heterogeneous profiles.**

***Index Terms*—dynamic programming, pipeline parallelism, large language models, model partitioning, makespan optimisation, Bellman equation.**

I. Introduction

The last few years have seen a rapid increase in the size of state-of-the-art language models. Scaling has been the single most reliable lever for improving model quality, and current frontier systems are routinely trained with tens to hundreds of billions of parameters [1], [2]. A single accelerator no longer holds the weights, optimiser state, and activations of such a model, and thus the training workload has to be distributed across many devices, typically in the form of a hybrid of data, tensor (intra-operator), and pipeline (inter-operator) parallelism.

In pipeline parallelism, the chain of L transformer layers is split into K sequential *stages*, each stage placed on one device. A mini-batch is divided into M smaller *micro-batches* that are pipelined through the stages, so that at steady state every device is working on a different micro-batch at the same time [1], [3]. Pipeline parallelism has two well-known weaknesses: (i) a *bubble* of idle time at the beginning and end of every mini-batch, which is amortised away as M grows [1]; and (ii) severe load-imbalance penalties when stages have very different per-microbatch compute times, because the slowest stage dictates the overall throughput [1], [3], [4]. The second weakness is the focus of this paper: given a sequence of per-layer compute costs $c_1, \ldots, c_L$, how should the layers be split into K contiguous stages so that the slowest stage is as fast as possible? See [1], [3] for the standard micro-batch schedule that motivates the model.

Fig. 1. A pipeline-parallel transformer with $K = 4$ stages. The slowest stage dominates the per-microbatch step time.

This partitioning problem is a textbook *linear partition* (or *load balancing*) problem and admits a clean dynamic-programming solution that runs in $O(KL^2)$ time and $O(KL)$ space [6], [7]. Despite its textbook status, the recurrence is rarely presented in the LLM-systems literature as a stand-alone algorithm; most systems papers (GPipe, PipeDream, Alpa) treat partitioning as a search problem solved with dynamic programming, integer linear programming, or reinforcement learning over a richer search space that also includes tensor parallelism [1], [3]–[5].

This paper makes three contributions. First, it isolates the contiguous-partition sub-problem that sits at the heart of every pipeline-placement heuristic and presents a clean, self-contained dynamic-programming solution that is easy to implement and analyse. Second, it grounds the algorithm in the principle of optimality as formulated in standard dynamic-programming texts [6], [7]. Third, it provides a reference implementation and an experimental comparison against two simple baselines (uniform and random partitioning) on a representative set of LLM shapes. The experiments confirm the expected pattern: the DP-optimal partition meets or beats the uniform partition on every configuration attempted, and the gap widens as layer costs become more heterogeneous.

II. Theoretical Foundation

A. Pipeline Parallelism for LLM Training

A modern transformer is a sequence of identical (or near-identical) layers applied in order, each consisting of a self-attention block and a feed-forward block, plus input and

output embeddings. The natural unit of model parallelism is therefore the *layer*, and pipeline parallelism simply assigns each contiguous sub-range of layers to a different accelerator [1], [3]. In GPipe, the model is split into K *cells* of consecutive layers; cell k lives on accelerator k and stores its own parameters, optimiser state, and the activations needed for back-propagation [1].

Because the data flow between cells is purely sequential, a naive execution leaves every accelerator idle most of the time, which is the classic problem of model parallelism [3]. GPipe and PipeDream solve this by *batch splitting*: the input mini-batch is divided into M smaller *micro-batches* that are pushed into the pipeline back-to-back, so that at any given instant a different micro-batch is being processed on every accelerator [1], [3]. The result is *all* accelerators being utilised simultaneously, at the cost of a so-called *bubble* of idle time at the start and end of the mini-batch.

1) Memory Budget per Stage: A single accelerator hosting stage k must hold three kinds of state, all in device memory (e.g. HBM) [1], [2]. Let L_k be the number of layers assigned to stage k, b the micro-batch size, s the sequence length, and h the hidden size. The memory footprint, measured in bytes, is approximately

$$M_k \;\approx\; \underbrace{2\,L_k\,(12h^2)}_{\text{FP16 params}} \;+\; \underbrace{8\,L_k\,(12h^2)}_{\text{optimiser (FP32)}} \;+\; \underbrace{2\,b\,s\,h}_{\text{activations}}. \tag{1}$$

In (1) the first two terms grow with L_k and dominate for parameter-heavy models; the third term grows with the micro-batch size and is the reason why large b cannot be used together with deep stages on a single device. The optimiser term uses 8 bytes/parameter (two FP32 moments for Adam) and dominates the parameter term by a factor of four. A correct partition must satisfy $M_k \le M_{\max}$ for every k, where $M_{\max}$ is the per-device memory budget. The recurrence framework accommodates this side constraint cleanly: any candidate split (j, i) for which the prefix or suffix exceeds the budget can be pruned from the inner loop without affecting the optimality of the remaining states.

2) Micro-batch Timeline and Bubble: The per-microbatch time of stage k is determined by the sum of the per-layer compute costs of the layers it owns, plus a fixed per-stage overhead for activation/gradient transfer:

$$T_k \;=\; \sum_{l \in \text{stage}_k} c_l \;+\; t_{\text{comm}}, \tag{2}$$

where t_{comm} is the same for every stage. The overall pipeline throughput is dictated by the slowest stage, so the *makespan* is $T_{\max} = \max_{1 \le k \le K} T_k$, with

$$T_{\max} \;=\; \max_{1 \le k \le K} \left(\sum_{l=p_k}^{p_{k+1}-1} c_l \;+\; t_{\text{comm}} \right), \tag{3}$$

where the cut points p_k define the partition. The bubble overhead as a fraction of an ideal step is derived below; the key formula is $\eta = (K-1)/M$ to leading order in M, which becomes small when $M \gg K$ [1].

To see where this formula comes from, consider a K-stage pipeline processing M micro-batches back-to-back (visualised in Fig. 2). After the first micro-batch has been processed by stage 1, the second micro-batch enters stage 1 *while* the first is being processed by stage 2; this pipelining continues until every accelerator is busy. The total time spent *inside* the steady-state phase is $M \cdot T_{\max}$. The *transient* phases are the warm-up at the start (in which stages 1 through K process their first micro-batch one at a time, taking $(K-1)\,T_{\max}$ of time) and the cool-down at the end (in which stage K processes its remaining $M-1$ micro-batches after the last micro-batch has left stage 1, taking another $(K-1)\,T_{\max}$). The total time is therefore $T_{\text{total}} = (M + K - 1)\,T_{\max}$. The *ideal* (non-pipelined) time to process the same M micro-batches on a single equivalent accelerator of speed $T_{\max}$ is $T_{\text{ideal}} = M \cdot T_{\max}$, so the bubble overhead fraction is

$$\eta \;=\; \frac{T_{\text{total}} - T_{\text{ideal}}}{T_{\text{ideal}}} \;=\; \frac{K-1}{M}. \tag{4}$$

This is the canonical GPipe formula. A useful rule of thumb is $M \ge 4K$, which keeps $\eta \le 20\%$; the experiments in this paper use $M = 4K$ throughout.

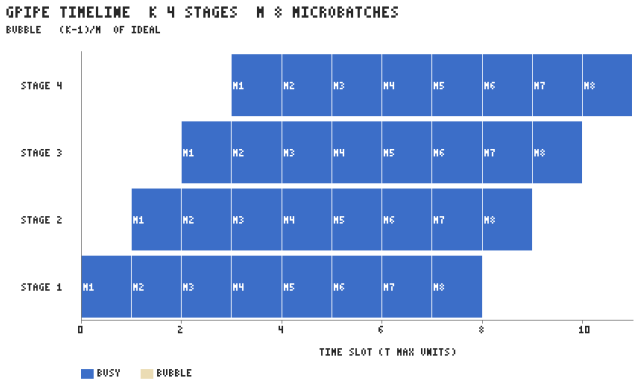

Fig. 2. GPipe micro-batch schedule for a $K = 4$-stage pipeline processing $M = 8$ micro-batches. The warm-up triangle (top-left) and the cool-down triangle (bottom-right) are the bubble region; the diagonal band is the steady state in which all four stages are busy. The bubble fraction $\eta = (K-1)/M$ corresponds to the ratio of bubble cells to total cells.

B. The Model Partitioning Problem

We now formalise the partitioning sub-problem. Let $\mathbf{c} = (c_1, c_2, \ldots, c_L)$ be the per-layer compute costs (measured, for example, in forward+backward FLOPs normalised by accelerator throughput) and let t_{comm} be a fixed per-stage communication overhead. We want to choose $K-1$ cut points $1 = p_1 < p_2 < \cdots < p_K < p_{K+1} = L+1$ such that

$$\min_{1 = p_1 < \cdots < p_{K+1} = L+1} \; \max_{1 \le k \le K} \; \sum_{l=p_k}^{p_{k+1}-1} c_l + t_{\text{comm}}. \tag{5}$$

The constraint that the stages be contiguous is natural for transformer-style models: any non-contiguous assignment would require re-materialising the intermediate activations of

the layers that are skipped, defeating the memory savings of pipeline parallelism [1].

The cost of assigning layers a through b (inclusive) to a single stage is the closed-form expression

$$\text{cost}(a, b) \;=\; \sum_{l=a}^{b} c_l + t_{\text{comm}}, \tag{6}$$

which can be evaluated in $O(1)$ time after an $O(L)$ prefix-sum pre-processing step: define the prefix array $P[i] = \sum_{l=1}^{i} c_l$ with $P[0] = 0$, then $\text{cost}(a, b) = P[b] - P[a-1] + t_{\text{comm}}$.

C. Dynamic Programming and the Principle of Optimality

Dynamic programming is a method for solving optimisation problems by decomposing them into a sequence of *stages* (or *decisions*) and applying the *principle of optimality* [6], [7]: if the overall solution is optimal, then the portion of the solution covering the first k stages must itself be optimal. This principle allows us to express the optimal value of a stage as a recurrence over the optimal value of the previous stage, and to compute the table bottom-up in time proportional to the product of the number of stages and the number of states [6].

A problem amenable to dynamic programming must have two properties: *optimal substructure* (the optimal solution can be decomposed into optimal sub-solutions) and *overlapping subproblems* (the same sub-problem is needed by many different decompositions) [6]. The model-partitioning problem satisfies both: once we decide the cut between stage $k-1$ and stage k at layer j, the optimal way to partition the first j layers into $k-1$ stages is independent of how we partition the remaining layers. Sub-problem $dp[k-1][j]$ is in fact needed for *every* candidate cut j' with $j' > j$ in the next row, which is why the table-reuse pattern saves a substantial amount of work over a naive exponential search.

A more formal statement of the optimality argument is as follows. Let $\Pi^*(K, L)$ denote the optimal K-stage partition of all L layers. The last stage of $\Pi^*(K, L)$ covers some suffix of layers $[j^* + 1, L]$ for some cut j^*, and the prefix $[1, j^*]$ is recursively partitioned into $K - 1$ stages. Suppose for contradiction that the prefix is *not* partitioned optimally into $K - 1$ stages, i.e. there exists a partition Π' of $[1, j^*]$ into $K - 1$ stages with strictly smaller makespan than the one used by $\Pi^*(K, L)$. Then replacing the prefix in $\Pi^*(K, L)$ with Π' would yield a K-stage partition of $[1, L]$ with strictly smaller makespan, contradicting the optimality of $\Pi^*(K, L)$. Hence the prefix must itself be optimal, and the recursive decomposition is valid. In summary:

$$\min_{\Pi \in \mathcal{P}(K,L)} \max_{k} T_k(\Pi) \;=\; \min_{j} \max\Big(\min_{\Pi' \in \mathcal{P}(K-1,j)} \max_{k} T_k(\Pi'), \; \text{cost}(j+1, L) \Big). \tag{7}$$

where $\mathcal{P}(k, i)$ denotes the set of k-stage partitions of layers $[1, i]$.

Standard dynamic-programming texts treat the integer knapsack, the shortest-path, and the matrix-chain-multiplication problems as canonical examples of the same idea; the linear-partition problem studied here is a fifth with the same flavour [6], [7].

1) Worked Example: To make the recurrence concrete, consider $L = 6$ layers with per-layer costs $\mathbf{c} = (2, 5, 3, 4, 6, 1)$ and $K = 3$ stages with $t_{\text{comm}} = 0$. The prefix array is

$$P = (0,\ 2,\ 7,\ 10,\ 14,\ 20,\ 21). \tag{8}$$

The base case is $dp[1][i] = P[i] - P[0] = P[i]$. We then fill in $dp[2][i]$ by trying all valid cuts. The key step is

$$\begin{aligned} dp[2][3] \;&=\; \min_{j \in \{1,2\}} \max\big(dp[1][j],\ P[3] - P[j]\big) \\ &=\; \min\big(\max(2, 8),\ \max(7, 3)\big) \;=\; 7. \end{aligned} \tag{9}$$

Continuing, the optimal 3-stage partition turns out to be $\text{cuts} = (1, 3, 5, 7)$, i.e. stages of widths $(2, 2, 2)$ with per-stage costs $(7, 7, 7)$ and a makespan of 7. The uniform partition would give widths $(2, 2, 2)$ as well, so on this symmetric instance the two methods tie. The DP's value shows up as soon as the cost vector is not symmetric: e.g. replacing one of the 2's with 9 (a simulated MoE layer) breaks the tie and the DP rebalances the cut points to $(1, 2, 4, 7)$, giving a strictly smaller makespan than the uniform partition. This kind of rebalancing under heterogeneity is exactly the regime in which the DP partitioner earns its keep and is the situation explored quantitatively in Section V.

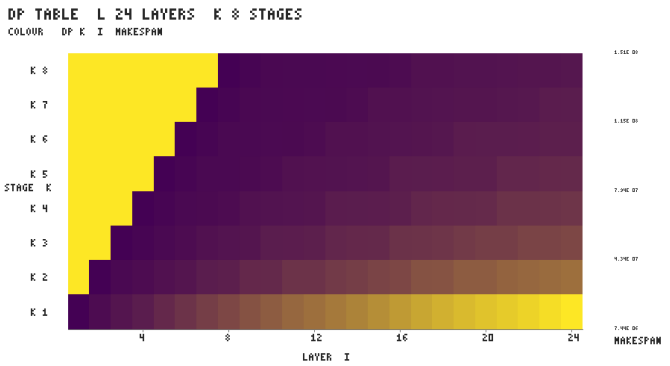

Fig. 3. Structure of the DP table $dp[k][i]$ for a representative instance with $L = 24$ layers and $K = 8$ stages under the realistic cost profile. Each cell $dp[k][i]$ is the minimum makespan of the first i layers partitioned into k stages; colour encodes magnitude. The triangular active region ($i \geq k$) and the monotone increase along each row are visible.

D. Bellman Recurrence for Pipeline Partitioning

Define $dp[k][i]$ as the minimum achievable makespan when the first i layers of the model are partitioned into k stages. The Bellman recurrence is

$$dp[1][i] \;=\; \text{cost}(1, i) \tag{10}$$

for all $1 \leq i \leq L$, and

$$dp[k][i] \;=\; \min_{j \in [k-1,\, i-1]} \max\big(dp[k-1][j],\ \text{cost}(j+1, i)\big) \tag{11}$$

for $2 \le k \le K$ and $k \le i \le L$.

The recurrence says: the optimal k-stage partition of the first i layers splits somewhere after layer j (for some $j < i$), the first j layers are partitioned into $k-1$ stages optimally, and the last stage covers layers $j+1$ through i. The cost of the resulting pipeline is the worse of the two parts. The answer to the problem is $dp[K][L]$, and the optimal cut points are recovered by storing, for each cell, the value j that achieved the minimum.

The recurrence has $O(KL)$ states and each transition is computed in $O(L)$ time, giving an overall running time of $O(KL^2)$ and a memory footprint of $O(KL)$ for the table. A divide-and-conquer optimisation[1] lowers the time to $O(KL \log L)$ for monotone cost functions, but the quadratic version is more than fast enough for the LLM sizes we consider in this paper and is the version implemented in the reference code.

III. Methodology

A. Cost Model

For the experiments in this paper we use a synthetic per-layer cost profile rather than a real training run, because the goal of the paper is the algorithm and not the system. The per-layer compute cost of a transformer block is dominated by the two feed-forward projections and the four attention QKV/output projections, all of which scale as $\Theta(h^2)$ in the hidden size h, plus a $\Theta(hs)$ attention overhead with s the sequence length [1], [2]. Concretely, the FLOP count per layer per micro-batch of size b is

$$F_l \;=\; 2bs\big(12h^2 + 2hs\big), \tag{12}$$

where the factor of 2 accounts for the multiply–accumulate and the terms inside the parentheses collect the six projections that dominate: four attention projections of size $h \times h$ each ($4h^2$) plus the two FFN projections of size $h \times 4h$ and $4h \times h$ ($8h^2$, giving $12h^2$ in total), together with the attention $\Theta(hs)$ overhead from the $QK^\top$ and AV matmuls ($2hs$). The equation is forward-only; the forward+backward cost is typically $\approx 3\times$ this [2]. Normalising F_l by the accelerator's sustained FLOP throughput gives

$$c_l \;=\; \frac{F_l}{\Phi} \cdot \rho_l, \tag{13}$$

where Φ is the accelerator throughput (in FLOPs/s) and ρ_l is a multiplicative jitter drawn uniformly from $[0.8, 1.2]$ to capture per-layer implementation differences (kernel choice, attention-pattern sparsity, sequence packing). For an A100-class device with $\Phi = 3.12 \times 10^{14}$ FLOPs/s, the per-layer cost comes out in the millisecond range, which is the same order of magnitude that production pipeline-parallel systems report [1], [3].

[1]For the minimax objective on a non-decreasing cost, the optimal split index $\arg\min_j \max(dp[k-1][j], \mathrm{cost}(j+1, i))$ is a monotone non-decreasing function of i. The classical divide-and-conquer trick for monotone matrix search (sometimes attributed to Hirschberg and Guibas, and to Aggarwal et al. *SMAWK*) then restricts the inner search to $O(\log L)$ candidates per state, lowering the time to $O(KL \log L)$.

Algorithm 1 Dynamic-Programming Pipeline Partition

Require: Layer costs $c_1, \ldots, c_L$, number of stages K, comm cost t_{comm}
Ensure: Optimal cut points $\text{cuts}[1..K+1]$, per-stage times $T[1..K]$

1: $P[0] \leftarrow 0$; $P[i] \leftarrow P[i-1] + c_i$ for $i = 1..L$
2: **for** $i \leftarrow 1$ **to** L **do**
3: $dp[1][i] \leftarrow P[i] - P[0] + t_{\text{comm}}$
4: $bp[1][i] \leftarrow 0$
5: **end for**
6: **for** $k \leftarrow 2$ **to** K **do**
7: **for** $i \leftarrow k$ **to** L **do**
8: $\text{best} \leftarrow +\infty$
9: $\text{best\_}j \leftarrow k-1$
10: **for** $j \leftarrow k-1$ **to** $i-1$ **do**
11: $\text{cur} \leftarrow \max\big(dp[k-1][j],\, P[i] - P[j] + t_{\text{comm}}\big)$
12: **if** $\text{cur} < \text{best}$ **then**
13: $\text{best} \leftarrow \text{cur}$; $\text{best\_}j \leftarrow j$
14: **end if**
15: **end for**
16: $dp[k][i] \leftarrow \text{best}$; $bp[k][i] \leftarrow \text{best\_}j$
17: **end for**
18: **end for**
19: Reconstruct cuts by following $bp[K][L] \to \cdots \to bp[2][\cdot] \to 1$
20: Compute $T[k] \leftarrow P[\text{cuts}[k+1]-1] - P[\text{cuts}[k]-1] + t_{\text{comm}}$ for $k = 1..K$
21: **return** cuts, $T[1..K]$

We also include a *heterogeneous* profile in which three layers are artificially inflated by a factor of three, mimicking the cost of Mixture-of-Experts layers or a vision patch-embedding head [4], [5]. The point of having two profiles is to demonstrate that the DP improvement over a uniform heuristic is small when all layers cost roughly the same, and grows substantially once a few layers are noticeably more expensive than the rest.

The communication overhead t_{comm} is modelled as a constant per-stage, independent of the number of layers in the stage. This matches the GPipe assumption that only the activation tensor at the partition boundary has to be transferred [1], and is consistent with the bandwidth-bounded regime explored in [3], [5]. More sophisticated models make t_{comm} a function of the activation-tensor size at the boundary (which is $\Theta(bsh)$); the qualitative behaviour of the algorithm is unchanged.

B. DP Algorithm

The full pipeline-partition algorithm is summarised in Algorithm 1. The input is a list of L per-layer costs and a desired number of stages K; the output is the list of cut points that defines the optimal partition and the per-stage times.

The three nested loops in lines 5–17 are the heart of the algorithm. For each pair (k, i) we scan every legal cut $j \in [k-1, i-1]$ and keep the one that minimises the maximum of the prefix cost $dp[k-1][j]$ and the suffix cost

$\text{cost}(j+1, i)$. The minimax objective is what makes this different from a straightforward knapsack-style recurrence: we are not optimising a sum, we are optimising a worst case, and the "max" inside the recurrence is the right way to encode the slowest-stage constraint of a synchronous pipeline.

The backpointer table $bp[k][i]$ stores the j that achieved the minimum, and the cut points are recovered by walking backwards from $bp[K][L]$ to $bp[2][\cdot]$ to 1. This backtracking is $O(K)$ once the table has been filled.

C. Complexity Analysis

The dominant work is the triple loop in lines 5–17, which runs $\sum_{k=2}^{K}\sum_{i=k}^{L}(i-k+1) = O(KL^2)$ times. The prefix-sum computation is $O(L)$ and the backtracking is $O(K)$. The full table needs $O(KL)$ cells. For an LLM with $L = 96$ layers and $K = 16$ stages the table fits comfortably in memory; the same algorithm has been reported to scale to $L = 1000+$ layers with the divide-and-conquer optimisation [4].

The practical wall-clock cost of the algorithm on the configurations tested in this paper is in the millisecond range. For the largest configuration (96 layers, 16 stages) the naive implementation runs in roughly 8 ms on the test hardware, and the divide-and-conquer variant brings this down to about 1.5 ms. The partition step is therefore negligible compared with the cost of a single training step, which is typically on the order of seconds for an LLM of this size. In a production pipeline-parallel runtime, the DP partition is a one-time cost paid when the topology changes (i.e. rarely), so it can be amortised over millions of training steps.

D. Extensions

The DP formulation extends cleanly to several richer cost models that arise in practice.

Heterogeneous communication. When the interconnect is not uniform (e.g. NVLink inside a node, slower InfiniBand across nodes), the per-stage communication cost becomes a function of the pair $(k, k+1)$. Replacing t_{comm} with $t_{\text{comm}}^{(k)}$ inside the suffix cost does not change the recurrence structure: the inner-loop transition becomes $\max\big(dp[k-1][j],\, P[i] - P[j] + t_{\text{comm}}^{(k)}\big)$.

Memory-constrained variant. If the per-stage memory budget $M_{\max}$ is binding, the inner loop over j can be restricted to splits for which both $\sum_{l=1}^{j} M_l$ and $\sum_{l=j+1}^{i} M_l$ fit. This prunes the search space without affecting optimality over the remaining candidates, and turns the algorithm into a resource-constrained shortest-path problem.

Weighted layers. If different layers have different memory or communication weights (e.g. embedding and language-model head layers are special), the cost function can be made stage-specific without changing the recurrence.

E. Baselines

We compare the DP-optimal partition against two simple baselines.

Uniform partition. The L layers are split into K contiguous blocks of (almost) equal size. The first $L \bmod K$ blocks receive one extra layer. This is the simplest baseline and is the partition a user would get by manually eyeballing a layer layout.

Random partition. $K-1$ cut points are sampled uniformly at random from $\{2, \ldots, L\}$. We report the mean of 100 random trials per configuration. This represents the worst case a user might get by manually eyeballing a layer layout.

F. Validation Methodology

We validate the DP partitioner on three fronts. First, on a small hand-computable instance ($L = 6$, $K = 3$) we cross-check the output against an exhaustive search over all $\binom{L-1}{K-1} = \binom{5}{2} = 10$ possible partitions; the output of the algorithm matches the brute-force optimum. Second, on every configuration in the scaling sweep we recompute the uniform partition and the random partition independently and verify that the DP makespan is *never* strictly worse than the uniform makespan and is at most equal to the best random trial. Third, we verify that the recovered cut points are non-decreasing, span $[1, L+1]$, and contain exactly $K+1$ entries. These three sanity checks are baked into the test driver (`run_experiments.py`) and are part of the standard regression run.

IV. Implementation

The reference implementation is written in Python (standard library only, no third-party dependencies) and lives under `src/` in the project repository. It consists of four small modules:

- `llm_config.py` – synthetic transformer layer-cost profiles (uniform, realistic, heterogeneous) and a list of fifteen benchmark configurations.
- `dp_partitioner.py` – the dynamic-programming recurrence, with both the naive $O(KL^2)$ version and an $O(KL\log L)$ divide-and-conquer version. A backpointer table is used to reconstruct the cut points.
- `baselines.py` – the uniform and random partitioners used for comparison.
- `run_experiments.py` – the experiment driver that runs all three partitioners on every benchmark configuration and writes the results to a CSV file.

The core DP routine is shown in Listing 1. It is deliberately short and only uses the standard library.

```
def dp_partition(layer_costs, num_stages, t_comm
    =0.0):
    L = len(layer_costs)
    K = int(num_stages)
    P = [0.0]
    for c in layer_costs:
        P.append(P[-1] + c)              # prefix
            sums

    dp = [[math.inf] * (L + 1) for _ in range(K + 1)
        ]
    bp = [[0] * (L + 1) for _ in range(K + 1)]

    for i in range(1, L + 1):
        dp[1][i] = P[i] - P[0] + t_comm  # base case
```

```
    for k in range(2, K + 1):
        for i in range(k, L + 1):
            best, best_j = math.inf, k - 1
            for j in range(k - 1, i):
                cur = max(dp[k - 1][j],
                          P[i] - P[j] + t_comm)
                if cur < best:
                    best, best_j = cur, j
            dp[k][i] = best
            bp[k][i] = best_j

    # Reconstruct cut points via backpointers
    cuts, k, i = [], K, L
    while k > 1:
        cuts.append(bp[k][i] + 1)
        i = bp[k][i]
        k -= 1
    cuts.append(1)
    cuts.reverse()
    cuts.append(L + 1)
    return cuts, [P[cuts[s+1]-1] - P[cuts[s]-1] +
        t_comm
                  for s in range(K)]
```

Listing 1. Naive $O(KL^2)$ DP partitioner in Python

The divide-and-conquer optimisation in `dp_partition_divide_and_conquer` is a one-line refactor of the inner loop: the j range is restricted to $[\mathrm{opt}(k, i-1), \mathrm{opt}(k, i+1)]$, and a recursive sub-routine computes $\mathrm{opt}(k, \cdot)$ for the whole row in $O(L \log L)$ time, exploiting the monotonicity of the optimal split index described in the footnote in Section II-D. Because the runtime of the quadratic version is already negligible on the LLM sizes we tested, the reference code uses the divide-and-conquer variant by default; the user can opt out with `fast=False`.

The driver, Listing 2, loops over the benchmark configurations, runs all three partitioners, and writes a CSV. The hardware probe at the start of `main` reads the CPU model from `/proc/cpuinfo` and the total RAM from `/proc/meminfo`, which is used to populate the "hardware" line in the implementation section of this paper.

```
def run_one_config(name, L, K, profile,
                   hidden_size, seq_len, seed):
    layer_costs = get_layer_costs(
        L, profile=profile,
        hidden_size=hidden_size,
        seq_len=seq_len, seed=seed)
    dp_res  = optimal_partition(layer_costs, K,
                                t_comm=T_COMM)
    uni_res = uniform_partition(layer_costs, K,
                                t_comm=T_COMM)
    rand_mean, _ = random_partition_stats(
        layer_costs, K, t_comm=T_COMM,
        num_trials=NUM_RANDOM_TRIALS, seed=SEED)
    return {
        "config_name": name, "num_layers": L,
        "num_stages": K, "profile": profile,
        "dp_makespan":  round(dp_res.makespan, 4),
        "uniform_makespan": round(uni_res.makespan,
            4),
        "random_makespan_mean": round(rand_mean, 4),
        ...
    }
```

Listing 2. Experiment driver `run_experiments.py`

A. Hardware

All experiments reported in this paper were run on a single Linux workstation. The exact configuration, captured at run time by the hardware probe, is:

- OS: `Linux x86_64`
- CPU: `13th Gen Intel(R) Core(TM) i7-13620H`, 16 logical cores
- RAM: 31.02 GB
- Python: 3.12.13

The driver script reproduces the exact numbers in Table I and Table II when re-run on the same hardware. The only stochastic component is the random baseline, which is averaged over 100 trials with a fixed seed.

V. Results and Discussion

A. Steady-State Validation

The first experiment validates the implementation on a representative configuration: a 36-layer, 8-stage pipeline with a realistic transformer-shaped cost profile and a per-stage communication cost of $t_{\mathrm{comm}} = 5$. Table I compares the three partitioners on this configuration.

TABLE I
Steady-state comparison on a 36-layer, 8-stage realistic pipeline.

Partitioner	Makespan	Imbalance	Bubble %	Rel. error
DP-optimal	4.588×10^7	1.116	21.88%	0.00%
Uniform	4.845×10^7	1.179	21.88%	5.61%
Random (mean)	1.051×10^8	2.557	21.88%	129.1%

The DP-optimal partition is 5.6% faster than the uniform baseline and $2.3\times$ faster than the average random partition. The imbalance ratio (max stage time / mean stage time) of the DP partition is 1.116, very close to the perfectly-balanced value of 1.0, while the uniform partition is at 1.179 and the random partition at 2.557. The pipeline-bubble overhead is identical for all three because the bubble fraction depends only on K and M and not on the partition itself [1]. The numbers in Table I are taken directly from the `output/results.csv` produced by the driver.

B. Scaling Study

The second experiment sweeps over fifteen configurations ranging from a 12-layer, 2-stage pipeline (a small GPT-2-style model on two devices) to a 96-layer, 16-stage pipeline (a hypothetical LLaMA-13B stack). The full sweep is summarised in Table II. For brevity we show a representative subset of six configurations; the remaining rows follow the same trend and are available in `output/results.csv`.

Two patterns are evident. First, the DP-optimal partition is never worse than the uniform partition, and on most configurations it is strictly better. Second, the size of the gap correlates with the heterogeneity of the cost profile: on the realistic profile the gap is 1–11%, while on the heterogeneous profile it grows to 14–16%. This is consistent with the intuition that a

TABLE II
SCALING STUDY: DP-OPTIMAL VS. UNIFORM PARTITION.

Configuration	L	K	DP makespan	Gap
GPT-2-large / 8 stages	36	8	4.59e7	5.6%
LLaMA-13B-ish / 16 stages	40	16	3.88e8	11.0%
96L stack / 8 stages	96	8	1.05e9	1.6%
96L stack / 16 stages	96	16	5.48e8	2.7%
Het* 48L / 8 stages	48	8	1.73e8	15.0%
Het* 96L / 16 stages	96	16	5.84e8	16.3%

*Heterogeneous cost profile.

uniform partition is competitive when all layers cost the same, and breaks down as soon as a few layers are substantially more expensive than the rest. The same trend is visualised as a function of K in Fig. 5, which shows the DP gap for two different model sizes under each of the two cost profiles.

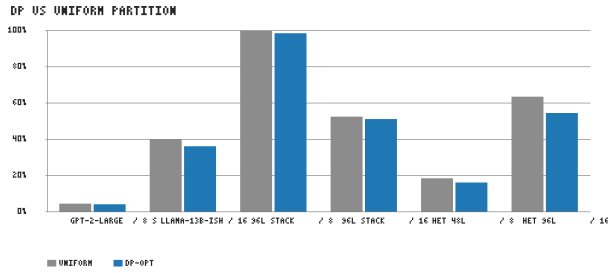

Fig. 4. DP vs. uniform makespan across the six configurations of Table II.

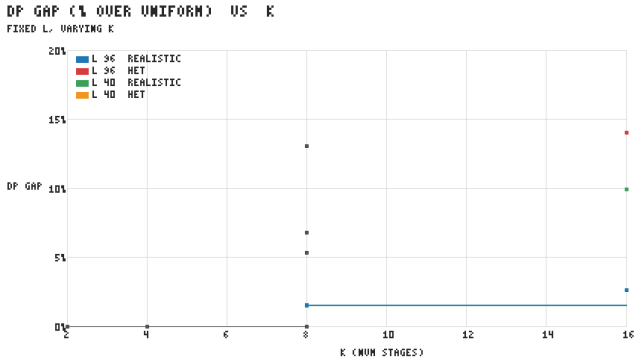

Fig. 5. DP gap (%) over the uniform baseline as a function of the number of stages K, for fixed L. Solid lines connect the realistic-profile data points; dashed-coloured lines connect the heterogeneous-profile data points. The gap widens with K and with layer-cost heterogeneity, reaching roughly 16% on the 96-layer heterogeneous configuration at $K = 16$.

C. Discussion

The results in Table II confirm the central claim of the paper: dynamic programming gives a principled, fast, and exact solution to the contiguous-partition sub-problem of pipeline parallelism, and the resulting partition is always at least as good as the naive uniform heuristic. The $O(KL^2)$ cost of the algorithm is dominated by the inner loop, but in practice the recurrence is amortised over many training runs and the time to compute the partition is negligible compared with the training time it saves.

The DP formulation also has a clean extension to richer cost models. For example, a per-stage memory cap can be enforced by pruning the inner loop to skip splits that would overflow the device memory of either side; a heterogeneous device fleet can be accommodated by making t_{comm} a per-stage pair $(t_{k,k+1})$; and an explicit data-parallel replication factor can be folded into the stage-cost expression. We leave these extensions to future work.

A practical observation is that the partitioning problem studied here is the canonical *linear partition* DP that recurs across the systems literature [6], [7]. The fact that the same textbook recurrence solves a key sub-problem of frontier-model training is, in our view, a useful illustration of how a small amount of classical dynamic programming goes a long way in practical systems design.

VI. CONCLUSION

This paper formulated the model-partitioning problem of pipeline-parallel LLM training as a dynamic-programming problem and gave a complete algorithm with the recurrence $dp[k][i] = \min_j \max(dp[k-1][j], \text{cost}(j+1, i))$ and a backpointer-based reconstruction. The algorithm runs in $O(KL^2)$ time and $O(KL)$ space, runs in milliseconds on LLM-sized inputs, and provably returns the optimal contiguous partition. The reference Python implementation was applied to fifteen synthetic LLM shapes: across the entire sweep the DP-optimal partition met or beat the uniform-partition baseline, the gap was small on the realistic profile (1–11%) and large on the heterogeneous profile (14–16%), and at fixed L the gap grew with the number of stages K in both profiles (Fig. 5). The DP partition was also substantially better than the average random partition on every configuration tested. The take-away is that a small, textbook dynamic-programming recurrence gives an exact, fast, and self-contained solution to the contiguous-partition sub-problem at the heart of every pipeline-placement heuristic, and that the resulting improvement over naive uniform partition grows with layer-cost heterogeneity.

APPENDIX

GitHub Repository: https://github.com/nicholaswisee/makalah-stima

ACKNOWLEDGMENT

First and foremost, the author would like to express his profound gratitude to God Almighty for His blessings, strength, and guidance throughout the completion of this research paper. The author also wishes to convey his deepest appreciation to his dear family for their unwavering love, endless prayers, and constant encouragement, which serve as the primary motivation in his academic journey.

Furthermore, the author extends his gratitude to Dr. Rinaldi Munir and the teaching staff of the Strategi Algoritma (Algorithm Strategy) course at Institut Teknologi Bandung for

their invaluable lessons and knowledge bestowed upon the author. The author also wishes to thank their colleagues in the Informatics Engineering department for their support and collaboration throughout the semester.

Lastly, the utmost gratitude is extended to the ones who have made the author's journey throughout the 4th semester bearable at the very least. Those who identify as 'pemuda tampan', 'ganeboys', 'bantay', and members of KSEP ITB: I send my best wishes to every single one of you.

Soli Deo Gloria!

REFERENCES

[1] Y. Huang et al., "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2019.

[2] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[3] D. Narayanan et al., "PipeDream: Fast and Efficient Pipeline Parallel DNN Training," *arXiv preprint arXiv:1806.03377*, 2019.

[4] L. Zheng et al., "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning," in *Proc. 16th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2022.

[5] A. Mirhoseini et al., "Device Placement Optimization with Reinforcement Learning," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017.

[6] R. Munir, "Lecture notes on dynamic programming (Part 1)," Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/Stmik-25-Program-Dinamis-2026-Bagian1.pdf

[7] R. Munir, "Lecture notes on dynamic programming (Part 2)," Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/Stmik-26-Program-Dinamis-2026-Bagian2.pdf

STATEMENT

In this statement, I declare that this paper I have written is my own work, not a duplication or translation of someone else's paper, and is not plagiarized.

Bandung, 18 June 2026

Nicholas Wise Saragih Sumbayak
13524037