**I said：**

Now I'm going to ask some questions about this paper, please answer in detail

**ChatGPT siad：**

Awesome—ready when you are! To make your questions easier to explore, here’s a clear, student-friendly map of **Ansor** (so you have the “lay of the land” before we dive into specifics).

**What problem Ansor solves (in plain English)**

Deep-learning operators (like conv2d, matmul) run fast only if their **low-level loops** (tile sizes, vectorization, parallelism, caching…) are arranged just right for each **hardware** (CPU/GPU/ARM). Hand-tuned libraries cover common cases, but new operators and odd shapes lag behind. Prior auto-tuning either (a) relies on **manual templates** (limited shapes/ideas), or (b) builds programs **step-by-step** and prunes too early using rough estimates. Ansor aims to **search a much bigger space** *and* explore it efficiently to find faster kernels automatically.

**The three big ideas (mental model)**

**1) A two-level search space: “sketch + annotations.”**

* **Sketch** = the *high-level* loop structure (e.g., multi-level tiling, fuse a consumer, add a cache stage).
* **Annotations** = the *low-level* details (tile sizes, which loops to parallelize/vectorize/unroll, etc.).  
  Ansor **generates sketches automatically** for a subgraph, then **samples full programs** by filling in annotations at random—giving every full program a chance to be tried.

**2) Fine-tuning with evolutionary search + a learned cost model.**

* Start from those random full programs; then **mutate/crossover** them (e.g., tweak tile sizes, change parallel granularity, move a computation) while a **cost model** quickly scores thousands of variants; only a few get measured on hardware and used to re-train the model. This loop converges toward high-performance kernels.

**3) A task scheduler for end-to-end models.**  
A whole network is partitioned into subgraphs (e.g., conv+relu blocks). Ansor’s scheduler uses a **gradient-descent–style rule** to spend more tuning time where it most improves end-to-end latency (rather than wasting time on unimportant parts).

**Why this is different from older approaches**

* **Not just fixed templates.** Templates lock you into a small set of loop shapes; Ansor’s **sketch rules** can derive many structures (tiling + fusion + caching + reduction factorization), then freely explore the low-level choices.
* **No early pruning of half-built programs.** Earlier “sequential construction” methods keep only top-k partial programs based on unreliable estimates. Ansor evaluates **complete programs**, then evolves them—so it avoids throwing away designs that would have ended up fast.

**Results (what to remember)**

Across full networks on Intel CPU, ARM CPU, and NVIDIA GPU, Ansor matched or beat strong baselines, improving throughput by **up to 3.8× (Intel), 2.6× (ARM), and 1.7× (GPU)**. Compared to the popular AutoTVM tuner, it reached equal/better performance and often with **less search time**, thanks to the scheduler.

**Quick glossary (helpful for upcoming questions)**

* **Tiling:** Break loops into blocks to improve cache/SM usage.
* **Fusion:** Compute two ops together to reduce memory traffic.
* **Sketch:** High-level loop structure (e.g., multi-level tiling “SSRSRS” on CPU).
* **Annotation:** Concrete choices (tile sizes, which loop is parallel/vectorized/unrolled).
* **Evolutionary search:** Mutations/crossovers on programs guided by a cost model; a few are actually benchmarked to keep the model honest.

That’s the gist. Fire away with any questions—algorithm details, how sketches are built, what mutations do, why the scheduler uses a gradient-style rule, or how to interpret the speedup graphs—and I’ll unpack each in beginner-friendly steps using the paper’s specifics.

**I said：**

The paper talks a lot about sketches and annotations for programs. Can you explain in simple terms why splitting the search space this way is better than just having one big set of possible programs?

**ChatGPT said：**

Great question 👌 — this “sketch + annotations” split is **the core trick** that makes Ansor practical. Let’s build the intuition step by step.

**1. Imagine searching without sketches**

Suppose you want to optimize a convolution operator.

* Every decision (tile sizes, cache levels, loop order, unrolling, vectorization, fusion…) could be made in **many different combinations**.
* If you try to search this “flat” space — one giant set of all possible programs — it’s astronomically large, like trying to guess a good password by brute force.
* Worse, most random combinations would be nonsense (illegal tilings, redundant caches) or very slow, so the search wastes tons of effort.

**2. What “sketches” do**

A **sketch** is like a **blueprint** for the *big structural decisions* in a program.

* Example: “Apply two levels of tiling (outer + inner), then fuse the consumer op, then add a cache write.”
* This doesn’t yet say *how big the tiles are* or *which exact loops to parallelize* — it just fixes the high-level structure.
* Think of it like **choosing the recipe** before you worry about the ingredient amounts.

So instead of wandering aimlessly through all possible code, Ansor generates **a set of promising sketches** automatically using rules (tiling, fusion, caching, factorization).

**3. What “annotations” do**

Once a sketch is chosen, **annotations fill in the details**:

* Tile sizes (e.g. tile by 32 vs tile by 16).
* Which loop to parallelize.
* Which loops to unroll or vectorize.

These are the **fine-grained knobs** that still matter, but now they’re applied to a **valid, structured base program**.

**4. Why this split is better**

* **Shrinks the useless space:** By fixing the *macro-structure* with sketches, Ansor avoids exploring obviously bad designs (like fusing everything into one giant loop or tiling in meaningless ways).
* **Keeps flexibility:** Unlike hand-written templates, sketches are generated by rules, so Ansor can still cover a wide variety of loop structures.
* **Better use of compute:** The evolutionary search now only mutates within a sketch’s valid region (annotations), so you get much higher chance of finding good kernels quickly.

In short:  
👉 Without sketches: **needle in a haystack** search.  
👉 With sketches: **choose a haystack that’s known to have needles**, then fine-tune within it.

**5. Analogy**

Think of designing a house:

* A **sketch** is the floor plan (2 floors, open kitchen, 3 bedrooms).
* The **annotations** are the specific choices (exact room dimensions, paint colors, window sizes).
* If you don’t pick a sketch first, you’d be mixing every possible wall placement + color + window size at once → chaos.
* By splitting the search, you’re always designing a *valid house*, then adjusting details.

✅ **In short:**  
Splitting into sketches + annotations lets Ansor explore **only meaningful program structures** first, then tune the fine details. This makes the search both broader (more structural diversity than fixed templates) and more efficient (fewer wasted candidates than flat random search).

Would you like me to also walk through a **concrete example from the paper** (say, a conv2d operator) showing what a sketch looks like vs what its annotations look like? That way you’ll see the split in action.

**I said：**

Ansor avoids pruning incomplete programs early, unlike Halide’s approach. Why is it so important to only evaluate complete programs. Does that make the cost model much more reliable?

**ChatGPT said：**

You’ve spotted one of the most important design contrasts! Let’s unpack it slowly.

**1. What Halide’s “sequential construction” does**

* In Halide-style auto-schedulers, a program is built step by step: first pick a tiling, then a loop reordering, then a parallelization, etc.
* At each step, you have a **partial program** (incomplete schedule).
* The system uses a **cost model estimate** to decide which partial programs look promising, and prunes the rest.
* Only the “top-k” partials survive to the next step.

👉 Problem: These cost model estimates are **very noisy for incomplete programs**, because many key decisions (tile sizes, caching, unrolling) haven’t been made yet. A partial plan that *looks bad early* might have become a winner once completed — but it gets thrown away too soon.

**2. What Ansor does differently**

* Ansor skips pruning mid-way.
* It **always generates complete programs** (with all tiling, caching, parallelization, unrolling decisions specified).
* Then the cost model can evaluate the *actual full program* with much more context.

👉 Result: The cost model’s predictions are **far more reliable**, because they’re about a complete, runnable program.

**3. Why reliability matters so much**

* With noisy early predictions, you risk **eliminating all the best candidates** before they even finish being built.
* This leads to **local optima**: you explore only a narrow slice of the space.
* By sticking to complete programs, Ansor gives the cost model **better training data** (realistic feature patterns) and reduces the chance of premature pruning.

It’s the difference between:

* **Halide style:** judging a half-built house blueprint by looking at just one wall and guessing how good the whole house will be.
* **Ansor style:** only rating *finished houses* — you can see the floor plan, materials, windows, etc., so your judgement is much more trustworthy.

**4. Does this make search slower?**

Surprisingly, no.

* Even though Ansor evaluates only full programs, it uses **sketch rules** to make the space manageable (not every crazy program is considered).
* And it uses **evolutionary search + cost model** to avoid exhaustively measuring everything.
* The combination keeps search efficient *and* robust.

✅ **In short:**  
Yes — evaluating only **complete programs** makes the cost model much more reliable, because it sees the whole set of choices rather than noisy fragments. This avoids throwing away promising directions too early, which was a major weakness in Halide’s pruning approach.

Do you want me to also explain **what features the cost model actually uses** (like loop counts, memory reuse, arithmetic intensity) to make those predictions more accurate on full programs?

**I said：**

Since Ansor tries such a huge search space, does that make it slower in practice, or does the evolutionary search and cost model keep it competitive?

**ChatGPT said：**

Great question 👍 — at first glance, Ansor’s **giant search space** sounds like it should be a nightmare: more possibilities = more time, right? But the cleverness is in how Ansor *navigates* that space. Let’s break it down.

**1. Why the search space is huge**

* A convolution or matmul kernel has tons of design knobs: loop orderings, tiling factors, fusion options, caching strategies, unrolling, vectorization…
* Combine them all and you easily get **billions of legal programs**.
* Ansor deliberately keeps this space **very large** (bigger than Halide templates or AutoTVM), because that’s where the best kernels hide.

**2. How Ansor makes it tractable**

Ansor doesn’t brute-force. It layers on two techniques:

**a. Evolutionary search**

* Starts with random complete programs sampled from sketches.
* Mutates them (change a tile size, parallelize a different loop, move a computation).
* Crosses over good candidates (mix-and-match parts of schedules).
* Keeps improving populations across generations.  
  👉 This avoids blind random sampling — it “evolves” better kernels steadily.

**b. Learned cost model**

* You can’t run every candidate on hardware — too slow.
* Instead, a cost model (trained on features of loop nests, memory access, etc.) quickly predicts performance for thousands of candidates.
* Only a few promising ones are actually measured, and the results are fed back to retrain the model.  
  👉 This gives the search a “compass,” so most time is spent on good regions of the space.

**3. The scheduling trick for full networks**

Ansor also has a **task scheduler**:

* It looks at the whole model graph, not just one op.
* Spends more tuning time on the ops that bottleneck end-to-end latency (like big convolutions), less on minor ops.  
  👉 This avoids wasting effort on kernels that barely affect runtime.

**4. So what happens in practice?**

Even with a massive space:

* Ansor found **faster kernels** than hand-tuned templates and older auto-tuners.
* On full networks, it sped up throughput by **up to 3.8× on Intel CPUs, 2.6× on ARM CPUs, and 1.7× on NVIDIA GPUs** over strong baselines.
* And it often did this in *comparable or less tuning time* than AutoTVM, thanks to the smarter search + scheduler.

**5. Analogy**

Think of the search space as a massive library:

* Brute force = reading every book (impossible).
* Halide’s pruning = reading only a few books based on shaky early guesses (you might miss the gems).
* Ansor = using a librarian (cost model) + book reviews (measurements) + family trees of recommendations (evolutionary search) to quickly zero in on the best shelves and best books.

✅ **In short:**  
Yes, the search space is enormous — but **evolutionary search + cost modeling + smart scheduling** keep Ansor competitive in tuning time, while still exploring broadly enough to uncover kernels that older, smaller search spaces would miss.

Would you like me to also explain **what features the cost model looks at** (like memory reuse, loop counts, parallelism) that make it fast and accurate in guiding this huge search?

**I said：**

In the benchmarks, Ansor often finds programs outside the search space of AutoTVM or FlexTensor. Could you give a simple example of an optimization that Ansor can discover but those other systems cannot?

**ChatGPT said：**

Great one — this really gets at why Ansor’s design is different from earlier auto-tuners like AutoTVM and FlexTensor. Let’s step through it in plain terms.

**1. AutoTVM / FlexTensor limitations**

* **AutoTVM**: relies on **manual templates**. A template might say:

“Tile the convolution into 2 levels, unroll inner loops, vectorize along channels.”  
That template defines the entire search space. If the best kernel needs *three-level tiling* or a *different fusion pattern*, it’s simply unreachable.

* **FlexTensor**: tries to generate search spaces automatically, but it still builds programs **sequentially** and prunes early based on noisy estimates. That biases it toward a narrow set of loop structures.

So both tend to stay inside “expected” patterns.

**2. What Ansor adds**

* **Sketch rules** can generate *many more loop structures*:
  + Multi-level tiling (beyond 2 levels).
  + Non-trivial **fusion** of producers/consumers.
  + Explicit **cache stages** for reads or writes.
  + **Reduction factorization** (splitting a reduction like sum into parallel partial sums).
* **Evolutionary search** then explores low-level details (tile sizes, unrolling, vectorization) within these sketches.

This makes Ansor’s search space **strictly larger**.

**3. A concrete example (from conv2d)**

Imagine a 2D convolution on CPU.

* **AutoTVM template space**: maybe only supports a simple “SSRSRS” tiling pattern (two levels of split loops for spatial axes, two for reduction axes).
* **Ansor**: could generate a sketch that:
  1. Applies **3-level tiling** on the spatial dimension (outer, middle, inner).
  2. **Factors the reduction loop** to compute partial sums in parallel, then merge them.
  3. **Caches writes** to local memory to reduce redundant global writes.

That combination (triple tiling + reduction factorization + write caching) is **not even expressible** in AutoTVM’s hand-written template. FlexTensor’s pruning would likely discard it before completion. But Ansor keeps it, evaluates the complete program, and often finds it runs faster.

**4. Why this matters in benchmarks**

That’s why the paper reports that Ansor often finds kernels “outside” the search spaces of AutoTVM/FlexTensor and achieves up to **3.8× CPU speedup, 2.6× ARM speedup, and 1.7× GPU speedup** over baselines.

Ansor

**5. Analogy**

Think of it like cooking:

* AutoTVM has a recipe template: “always roast chicken with two spices and one sauce.” You can try many spice/sauce combos, but that’s it.
* Ansor says: “Why not try three spices, braising instead of roasting, and adding a side dish?” — suddenly the space of meals is much larger, and you might discover a tastier one.

✅ **In short:**  
Ansor can discover optimizations like **multi-level tiling, reduction factorization, and cache insertion combinations** that AutoTVM’s fixed templates and FlexTensor’s early pruning cannot. This broader reach is exactly why Ansor beats them in practice.

Would you like me to **sketch a simple loop-nest before vs. after Ansor’s transformations** (like a toy conv2d) so you can *see* the kind of optimization AutoTVM misses but Ansor finds?

**I said：**

If a researcher comes up with a totally new operator, how easy is it to plug it into Ansor? Do they need to write templates, or just give the math definition?

**ChatGPT said：**

Excellent question — this is one of the **main selling points of Ansor** compared to older tuners like AutoTVM. Let’s unpack it.

**1. How it worked before (AutoTVM, etc.)**

* In **AutoTVM**, if you invented a new operator (say a novel attention kernel), you had to **hand-write a template** for it.
  + A template is a parameterized schedule (loop nest structure) that defines the search space.
  + Writing one requires deep hardware knowledge: which loops to tile, how to fuse them, which loops to parallelize/vectorize.
* If your template missed a clever optimization (say multi-level tiling or caching), the tuner would never find it.  
  👉 This meant a *lot* of manual engineering per new operator.

**2. How Ansor changes things**

With Ansor, you don’t write templates. Instead:

* You just provide the **pure mathematical definition** of the operator in TVM’s tensor expression language (like writing the formula for convolution, attention, etc.).
* Ansor then:
  1. **Generates sketches** automatically using generic rules (tiling, fusion, caching, reduction factorization).
  2. **Samples full schedules** by adding annotations.
  3. Evolves and tunes them using its cost-model–guided search.

So the researcher does **not** need to encode any scheduling expertise — the system explores it.

**3. Why this matters**

* **Lower barrier:** A researcher working on novel ops can try Ansor without being a GPU/CPU kernel wizard.
* **Broader exploration:** Because sketches cover a wide space, Ansor may find optimizations you wouldn’t think to include in a template.
* **Reusability:** The same sketch rules apply across operators, so you don’t reinvent the wheel for each one.

**4. A simple example**

Say you invent an operator my\_op(A, B) = sum\_i (A[i] \* B[i]^2).

* In AutoTVM: you’d have to write a schedule template describing tiling and reduction order.
* In Ansor: you just define my\_op as a reduction in tensor expressions. Ansor’s rules automatically see:
  + “Oh, this is a reduction” → try reduction factorization, parallel reduction, caching partial sums.
  + “Oh, elementwise multiplication + square” → try fusing them.
  + Then it explores all low-level details via annotations.

**5. So how easy is it?**

👉 Plugging in a new operator = just give the **math definition**. No templates needed. Ansor takes it from there.

✅ **In short:**  
Ansor eliminates the need to write templates for new operators. A researcher only needs to supply the **mathematical definition** (in TVM’s tensor expression), and Ansor’s sketch/annotation system automatically generates and tunes possible implementations. This is a big step forward in usability and coverage compared to older tuners.

Would you like me to also explain **what kinds of operators Ansor handles best** (e.g., compute-heavy vs memory-heavy) and where writing extra scheduling hints might *still* be helpful?